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

This text describes the third part of the assignment for the course *Object-oriented Programming*. The groups that were formed during the first and second part should preferably work together on the third part. 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.

A number of teaching assistants (TAs) will advise the students and answer their questions. In this phase of the assignment, each team has up to four slots of thirty minutes in which 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 three parts of this assignment, we create a simple game that is loosely based on the artillery strategy game *Worms*. Note that several aspects of the assignment do not correspond to the original game. Your solution should be implemented in Java 6 or 7 and follow the rules described in this document. The first part of the assignment focussed on a single class *Worm*. In the second part, we extended *Worm*, added additional classes and relationships between these classes. In this third part, we extend the program developed in part II, focussing particularly on inheritance. Modifications of the assignment with respect to part II are highlighted in blue.

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 their 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 this third part of the assignment, we extend the game from part two with computer-controlled opponents. Each such opponent executes a program which determines its worm's actions (turning, firing, etc.) throughout the game. As in the previous parts, your solution may contain additional helper classes (in particular classes marked *@Value*). In the remainder of this section, we describe the main classes in more detail. All aspects of the class `Worm` must be specified both formally and informally. All aspects of classes other than `Worm` must be documented in a formally way only. It is not necessary to provide documentation for the classes for representing programs. Note that if the assignment does not specify how to work out a certain aspect of the game, select the option you prefer. You may also use inheritance as you see fit.

1.1 Game World

Worms live in a rectangular two-dimensional underground landscape with slopes and obstacles. Each game world has a particular size, described by a finite *width* and *height* expressed in metres (m). The size of a world cannot change after construction. Both the width and height must be 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 bound.

Geological features of the game world shall be extracted from an image file such as the one shown in Fig. 1: Scaled to the dimensions of the game world, coloured pixels in the image represent impassable and indestructible terrain, while transparent pixels are passable by game objects such as worms or projectiles. More specifically, each pixel of an image that is x pixels wide and y pixels high shall be used to mark a rectangular area of $width/x \times height/y$ of the game world as either passable or impassable. These areas must be located at the same relative locations in the game world that are held by the pixels in the image file, respectively. The code to load image files and to compute game maps from these images is provided with this assignment.

A game world contains game objects, i.e. worms, worm food and at most one live projectile. At all times, each game object is located in at most one world. No world contains the same game object twice. All game objects 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 may overlap with other entities. If an entity leaves the boundaries of a world, that entity shall be removed from that world.

At the start of a game, a world may only contain worms and worm food. These worms and the worm food shall be placed at random passable locations adjacent to impassable terrain. A location is adjacent to impassable terrain if the location itself is passable and the location's distance to an impassable location is smaller than the game object's radius $\sigma \cdot 0.1$. It is suggested to determine such locations by selecting a perimeter location at random, and then iteratively explore terrain in the direction of the centre of the map until a valid initial position is found. However, other strategies may be used in the future. In fact, the actual strategy for finding a location that is adjacent to impassable terrain is of no relevance outside the class of game worlds. If no valid position can be found, the object will not be placed in the world. Worms may further be assigned to a `Team` at the moment at which they enter a world.

The class `World` shall provide methods for adding and removing worms and worm food. Those methods must be worked out defensively. The initial position of worms and worm food that are added to a game world is to be determined by that game world. All other attributes of game objects, including team membership, may be assigned arbitrarily. It should be possible to ask a world what worms, worm food and projectiles it contains. In addition to returning all the objects in the world, it must be possible to restrict the result to all objects of a specific type (e.g., all worms or all worm food). The class `World` shall further provide methods to determine whether a given location is passable, impassable or adjacent to passable terrain. These methods must be worked out in a total manner.

Finally, `World` must provide a method to start the game. That is, a first worm starts its turn and performs player-controlled actions such as moving,

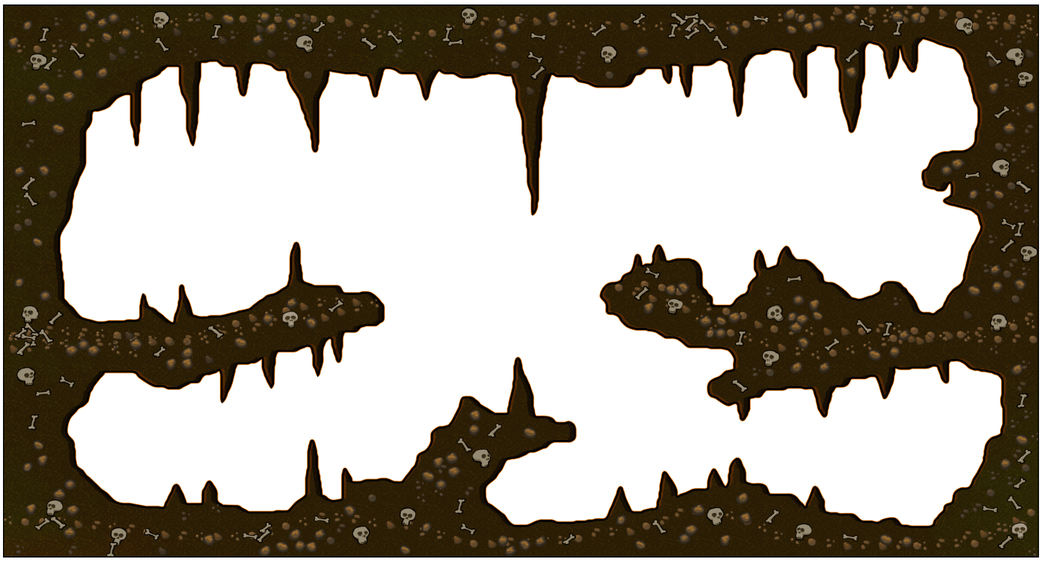

Figure 1: The game world. Source: `http://gna.org/projects/warmux/`

jumping or shooting. All worms in a game world shall then, one by one and in a cyclic order, perform player-controlled actions as specified in the following sections. A game world may not contain any live projectiles at the beginning of a worm's turn. Once a game has been started, it is not permitted to add further worms or worm food to that game world. The game is finished once only one worm or worms that belong to the same team are left in the game world and all other worms have been removed from that game world. No formal or informal documentation is required for the method to start the game, nor for any auxiliary method it may use. The class `World` must provide a method to ask a world for the winning worms.

1.2 Worms

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 and a current number of action points and hit points, which shall be represented by integer values. The maximum number of action points and hit 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, while the current number of points remain unchanged. As explained in Sections 1.3 and 1.5, the current number of action points and hit points may change during the program's execution. Yet, the current value of a worm's action points and hit points must always be less than or equal to the respective maximum value, but it must never be less than zero. At the start of a worm's turn, that worm's action points are assigned the maximum action points, and the worm's hit points are increased by 10. The worm's turn ends when either action points or hit points are decremented to zero. If the worm's hit points are decremented to zero, that worm is removed from the game world. All aspects related to action points and hit 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. Names can only use letters (both uppercase and lowercase), quotes (both single and double), numbers and spaces. "James o'Hara 007" is an example of a well-formed name. 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, action points and hit points of a worm.

1.3 Turning and Moving

A worm can move and turn. 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.

The class `Worm` shall further provide a method `move` to change the position of the worm based on the current position, orientation, and the terrain. Worms move from any location of the game world to another location that is adjacent to impassable terrain, following the slope s of that terrain in the direction of θ. Movement always occurs in steps. The distance d covered in one step shall not be greater than the radius σ of the worm.

More specifically, a worm at position (x, y) that is commanded to move one step in the direction θ will end up in a location (x', y') that is passable and adjacent to impassable terrain. The worm shall aim to maximise the distance d while minimising the divergence $|\theta - s|$, where $s = arctan((y - y')/(x - x')) = \theta \pm 0.7875$ and $d = \sqrt{(x - x')^2 + (y - y')^2}$ with $0.1m \leq d \leq \sigma$. Candidate divergences may be sampled with a precision (step size) of 0.0175 rad. This behaviour is illustrated in Fig. 2. If no such location exist because all locations in the direction of $\theta \pm 0.7875$ are impassable, the worm shall remain at (x, y). Otherwise, if locations in the direction of θ are passable but not adjacent to impassable terrain, the worm shall move there and then drop passively to impassable terrain as explained below. As the method `move` affects the position of the worm, it must be worked out defensively.

The actual way that worms move may change in future versions of the game. Regardless of the strategy, after a move, a worm will either have left the world or be on a location adjacent to impassable terrain. Moreover, if the worm has changed its position, its action points may have been diminished.

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. 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 that follows the slope of the terrain is computed as $|cos\, s| + |4sin\, s|$. Likewise 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.

Worms may also move passively, e.g. fall down a chasm. The class `Worm` shall implement a method `fall` to change the position of the worm as the

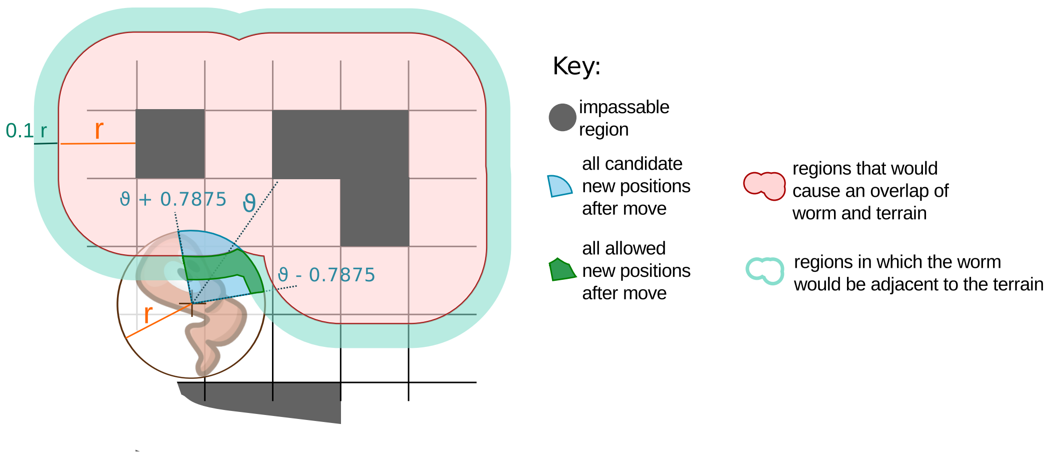

Figure 2: Illustration of a a worm's movement.

result of a free fall from the current position. Specifically, if a worm is not located adjacently to impassable terrain, it will fall straight down to the next location that is adjacent to impassable terrain. If there is no impassable terrain underneath the worm, that worm will fall out of the game world. Passive movement does not incur a decrease of the worm's action points but a decrease of 3 hit points for every meter (rounded down) travelled falling.

Students that aim at a score of 16 or more for the course must work out a formal documentation of the methods for moving and falling worms. Otherwise, an informal specification is good enough.

1.4 Jumping

Worms can 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, the geological features of the game world, 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 action 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 $v_0 = (F/m) \cdot 0.5\ s$.

As illustrated in Fig. 3, jumping worms always travel along a trajectory through passable areas of the map. The jump is finished when the worm

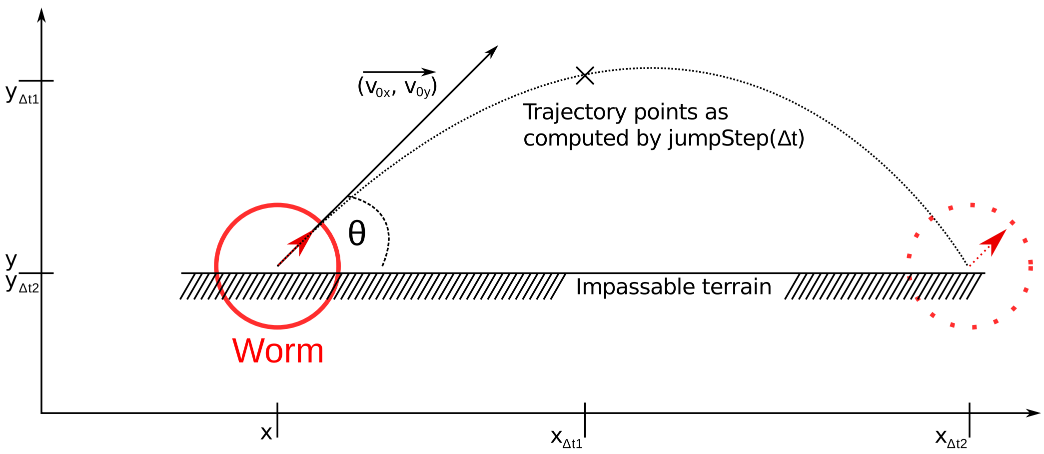

Figure 3: Illustration of a jumping worm's trajectory.

reaches a location that is adjacent to impassable terrain and at least σ metres away from (x, y), or when the worm leaves the map. Jumping consumes all remaining action points of a worm. A worm that has no action points left or that is located on impassable terrain must not jump.

The class `Worm` shall also provide a method `jumpTime` that returns the effective time of that jump until an obstacle is hit or the worm has left the game world, and a method `jumpStep` that computes in-flight positions $(x_{\Delta t}, y_{\Delta t})$ of a jumping worm at any Δt seconds 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.

1.5 Shooting and Projectiles

All worms are initially equipped with two weapons, a rifle and a Bazooka, and an unlimited supply of ammunition for these two weapons. In the future, worms may acquire further weapons, tools and ammunition. The class `Worm` shall provide methods to select a weapon and and to fire that weapon with a given propulsion yield p. The propulsion yield is an integer value that must be in the range of $0 \leq p \leq 100$. Firing a weapon costs a weapon-specific number of action points. A worm that has no action points left or that is located on impassable terrain must not shoot any of its weapons.

The behaviour of projectiles is similar to that of jumping worms. Hence, a class `Projectile` shall implement the methods `jump`, `jumpTime` and `jumpStep` as described in Sec. 1.4. Each projectile has an initial position (x, y) and orientation θ. A projectile's (x, y) shall be located just outside the worm's perimeter in the direction of the shooting worm's orientation. The projectile's θ shall be identical with the current orientation of the shooting worm. Projectiles also have a mass m, and are propelled in the direction of θ with force a F that is exerted for 0.5 s on the projectile. The projectile's radius σ can be derived by assuming that the projectile has is a spherical object with a homogeneous density of 7800 kg/m^3.

The projectile will move along a trajectory as explained in Sec. 1.4 until it hits impassable terrain or a worm, or leaves the game world. At that moment, the projectile is destroyed and removed from the game world. If a worm is hit, i.e. the projectile partially overlaps with the worm, a specific number of hit points is deduced from that worm's current number of hit points.

If not stated otherwise, all numeric characteristics of a projectile 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 projectile 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).

As in the class `Worm`, program aspects concerning a projectile's position, radius and mass shall be worked out defensively, while aspects concerning a projectile's orientation are to be implemented nominally.

The Rifle Rifle projectiles have a mass of 10 g and are propelled with a force of 1.5 N, unaffected by the propulsion yield. If a worm is hit, 20 hit points shall be deduced from that worm's current number of hit points. Shooting the rifle costs 10 action points.

The Bazooka Bazooka projectiles have a mass of 300 g and are propelled with a force of 2.5 N to 9.5 N, depending on the propulsion yield. If a worm is hit, 80 hit points shall be deduced from that worm's current number of hit points. Shooting the Bazooka costs 50 action points.

1.6 Worm Food

This part of the assignment is not mandatory for student groups that consist of less than two students.

Worms may consume worm food to grow in size. Each food ration 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 worm food shall be worked out defensively.

The shape of a food ration is a circle with finite radius σ (expressed in metres) centred on the food ration's position. The radius of a food ration shall be 0.20 m.

Worms automatically `eat` food rations if their body partially overlaps with a food object at the worm's final location after a `move`, `jump` or `fall` action. As the result of eating a food ration, the worm's radius σ shall increase by 10% and the food object shall be destroyed and removed from the game world. A worm may, as a result of consuming a food ration, be placed at impassable terrain.

1.7 Teams

This part of the assignment is not mandatory for student groups that consist of less than two students.

Worms may join a team at the moment at which they join a game world. Worms fighting together in a team may jointly carry victory after the game has been started and when only worms belonging to the same team remain in a world. A game world may contain up 10 teams and teamed worms may co-exist with individual worms that do not belong to any team. Worms in a team may still damage and destroy each other.

Each team shall have a name that is at least two characters long and must start with an uppercase letter. Names can only use letters (both uppercase and lowercase). The class `Team` shall provide methods to add a worm to a team and to query the live worms of a team. A valid implementation would be to always add new worms to the last team that was created. All aspects of the class `Team` must be worked out defensively.

1.8 Programs

A worm can store a program that determines its actions. If a worm is supposed to run such a program, that program must be passed to the worm at construction time. Re-loading or updating programs at runtime is not required. If a worm is assigned a program, this program starts executing as soon as the worm starts its turn. The worm's turn is finished when the program terminates or is interrupted. By default, a worm is not associated with a program.

A program typically starts with the declaration of global variables, followed by single statement which we refer to as the *main statement*. For example, consider the program shown below.

```
double d := 0.01;
bool b := true;
entity w;

while(b) {
  w = searchobj(0);
  if (isworm(w)) {
    fire 100;
  } else {
    turn d;
    d := d + 0.01;
  }
}
```

This program declares three global variables named `d`, and `b` `w`. The former variables are of type `double` and `bool`, while the latter is of type `entity`. The declaration of a variable starts with one of these three types, followed by the name of the variable. A declaration may involve an initial assignment denoted by `:=` followed by an expression that determines the initial value for the variable at stake. All the variables that are used in the statement that follows must be declared up front. In the example, the main statement is `while`-loop. The body of this loop contains an `if-else` statement, of which the if-branch contains a `fire` statement, while the else-branch contains a sequence of statements: a turn statement and an assignment.

The program associated with a Worm executes when that worm starts its turn in the game. A program can order its Worm to perform as many actions as permitted by the worm's action points. When the program associated with a worm runs, it executes until there are either no statements left, or until execution reaches an action statement (turning, jumping, shooting, ...) for which there are not enough action points to perform the action, or until 1000 statements (c.f. `s` in the BNF listing in Sec. 1.8.1) have been executed. If the previous execution of a program was not completely finished, the next execution continues at the point at which the previous execution was stopped. In that case, all global variables keep their value. Otherwise, execution of a program starts at the beginning with the processing of global variable initialisation.

As an example, consider one possible scenario of executing the program shown above. The first time this program runs, it may execute the else-branch of the `if-else` statement n times, encountering the turn statement and the assignment. The program will be interrupted once the worm runs out of action points. By then, the worm will have turned $n-1$ times for `d` radians and `d` will hold the value $0.1 + ((n-1) \cdot 0.1)$. The second execution of the program continues right at the `while` loop: the variable `d` still holds the value $0.1 + ((n-1) \cdot 0.1)$ from the previous execution, making the worm turn in bigger steps until its action points are again exhausted. The third time the program runs, it continues again at the `while` loop. This time, the worm may see another worm in the game world and start shooting its default weapon until that worm worm disappears or the worm runs out of action points.

When executing programs, you may assume that the program is completely correct (c.f. Sec. 1.8.5). This means among others that it does not contain statements such as `true + self` that violate typing rules. Illegal operations performed by a program should be handled in a total manner. That is, if a program performs an illegal operation (e.g. the program evaluates `getx null` or and division by zero, causing an exception to be thrown) execution of the program stops. Subsequent runs of the program immediately return.

1.8.1 Statements

The syntax of statements `s` in Backus Normal Form (BNF) notation is as follows:

```
s ::=
  x := e;
| while(e) { s }
| foreach(kind, x) { s }
| if (e) { s } else { s }
| print e;
| s*
| action;

action ::=
  turn e
| move
| jump
| toggleweap
| fire e
| skip

kind ::=
  worm
| food
| any
```

That is, a statement is either an assignment, a while loop, a for-each loop, an if-then-else, a print statement, a sequence of zero or more statements or an action statement. There are six different kinds of action statements: turning, moving, jumping, switching weapons, shooting and doing nothing (skip). The statement `print e` shall output the result of evaluating the expression `e`.

A for-each loop iterates over all entities of a given kind, i.e., all food objects, all worms or all game objects. As an example, consider the program shown below:

```
// Worm that prints the distance to the nearest other worm, turns and fires.

double angle;
entity nearestWorm;
double distanceToNearestWorm;
double x;
double y;
double r;
double wx;
double wy;
double wr;
double distance;
entity w;

while(true) do {
  x := getx self;
  y := gety self;
  r := getradius self;
  nearestWorm := null;
  foreach(worm, w) do {
    if (w != self) {
      wx := getx w;
      wy := gety w;
      wr := getradius w;
      distance := sqrt((((x - wx) * (x - wx)) + ((y - wy) * (y - wy))));
      if(nearestWorm == null) then {
        nearestWorm := w;
        distanceToNearestWorm := distance;
      } else {
        if(distance < distanceToNearestWorm) then {
          nearestWorm := w;
          distanceToNearestWorm := distance;
        }
      }
    }
  }
  if(nearestWorm != null) then {
    print distanceToNearestWorm;
  }
  turn (0 + -0.2);
  toggleweap;
  if (! sameteam(nearestWorm)) {
    fire 100;
  }
}
```

This program iterates over all worms to find the one that is closest to the worm executing the program. The variable `w` in the for-each loop gets assigned a different value in each iteration. The total number of iterations is equal to the number of worms in the world that contains the worm executing the program. Note that the variable used in a for-each loop (here `w`) must be an *existing* global variable. The body of a for-each loop should not contain action statements.

Students working alone do not have to support for-each loops.

1.8.2 Expressions

The syntax of expressions `e` in BNF notation is as follows:

```
e ::=
  x
| c
| true
| false
| null
| self
| e + e
| e - e
| e * e
| e / e
| sqrt(e)
| sin(e)
| cos(e)
| e && e
| e || e
| ! e
| e < e
| e <= e
| e > e
| e >= e
| e == e
| e != e
| getx e
| gety e
| getradius e
| getdir e
| getap e
| getmaxap e
| gethp e
| getmaxhp e
| sameteam e
| searchobj e
| isworm e
| isfood e
```

An expression is either a variable, a double constant, true, false, null, self (i.e. the worm that executes the program), an addition, a subtraction, a multiplication, a division, a square root, a sine, a cosine, a conjunction, a disjunction, a negation, or a comparison (less than, less than or equal to, greater than, greater than or equal to, equal to or different from). More interestingly, expressions may also employ inspectors on a given game entity, such as query the position or the remaining hit points of an entity.

The expressions `getx e`, `gety e`, `getradius e`, `getdir e`, as well as `getap e`, `getmaxap e`, `gethp e`, `getmaxhp e` and `sameteam e` respectively compute the x-coordinate, y-coordinate, radius, orientation, action points, hit points and whether an entity belongs to the same team as the executing worm, for the entity expression `e`. The expression `searchobj e` returns the next game object to be seen in a direct line from the executing worm in the direction of $\theta + e$. The expressions `isworm e` and `isfood e` can be used to determine the type of an entity expression `e`.

Students working alone do not have to support conjunction, disjunction, negation, sine and cosine expressions. If your group did not implement *Teams* and *Worm Food*, you do not have to implement the `sameteam e` and `isfood e` expressions.

1.8.3 Types

Expressions, global variables and values can have three possible types: `double`, `bool` and `entity`. A variable of type `entity` is either `null` or a reference to a worm or worm food.

The type `double` is the only numeric type used within programs that control a worm's behaviour. Yet, as some methods of the class `Worm` return or expect integer arguments, `double` values have to be converted to integer values "on the fly". Your implementation shall always round `double` values towards zero when such a conversion is necessary. Potential overflows in `double` to integer conversion shall be handler in a total manner.

All global variables are initialised to default values (based on their type) before the first execution of the program. If the variable is explicitly initialised in its declaration an assignment statement is created and executed so that the variable is re-initialised with the value resulting from the evaluation of the initialising expression.

When implementing your `double` and `bool` types, you are allowed to either implement your own type classes or to refer to `java.lang.Double` and `java.lang.Boolean` directly. You should be able to justify your decision.

1.8.4 Well-formed programs

We already mentioned that for-each statements may not directly or indirectly (via if statements, while statements or sequences in their body) contain action statements. You must write a method that checks whether a program satisfies that condition. That method must be worked out in a total way.

Students that make the project on their own must not work out this part of the project.

1.8.5 Type-Checking

Expressions expect operands of the proper type and yield values of a specific type. As an example, an addition expects two double values and yields a double value, a comparison expects two double values and yields a boolean value, the built-in function `getx` expects an entity and returns a double value, etc. Intuitively, checking whether each expression has operands of the proper type, and that `if` statements and `while` statements have boolean expression to control their execution, is known as type-checking.

We have already mentioned that in executing programs, you may assume that programs are completely correct. However, if you want to score 17 or more on this project, you must make it impossible to construct programs that violate the typing rules. More in particular, the Java compiler must complain if we try to build a program that has typing errors. An example of such a program would be `print self + 1.45`. We only expect those messages if we create programs directly by means of constructors in your hierarchy. Note that we do not expect you to prohibit Java programmers to build incorrect programs by means of reflection or by means of raw types underlying generic classes. You are not allowed to change the signature of the methods in `ProgramFactory` interface.

Of course, some typing errors cannot be detected by the Java compiler itself. An example of such a program is

```
entity x;
x := ... # some food entity
getap x  # fails because x is not a worm
```

Such errors must be detected at runtime, i.e. during the execution of the program. As explained before, runtime errors should be handled in a total manner. That is, execution of the program stops and subsequent runs of the program return immediately.

1.8.6 Parsing

The assignment comes with a number of example programs stored in text files. Reading a text file containing a program and converting it from its textual representation into a number of objects that represent the program in-memory is called *parsing*.

The assignment includes a parser. To parse a `String` object, instantiate the class `ProgramParser` and call its `parse` method. This method constructs an in-memory representation of the program by calling methods in the `ProgramFactory` interface. You should provide a class that implements this interface.

Performing actions by a worm shall not be implemented by invoking the respective action methods of the worm (e.g. `jump`) directly. Instead, your implementation of an action statement must call the corresponding method of the given action handler `IActionHandler`. This performs the action as if a human player has initiated it, eventually calling the corresponding method on the `Facade`.

The parser was generated using the ANTLR parser generator based on the file `WormsParser.g4`. It is not necessary to understand or modify this file.

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:

```java
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 a JUnit test suite for the classes `Worm` and `World` that tests each public method. Include this test suite in your submission. Obviously, we recommend you to build test suites for other classes as well, but this is not required as part of the project.

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. Initially, you may press `T` to create an empty team, `W` to add a player-controlled worm, `C` to add a program-controlled worm, `F` to add worm food, and `S` to start the game. The in-game command keys are `Tab` for switching worms (finishes a worm's turn), `left` and `right` arrow key (followed by pressing `return`) to turn, `up` to move forward, `n` to change the worm's name, and `j` to jump. A worm shoots when `s` is pressed. The active weapon may be toggled by pressing `w`, and the propulsion yield is increase by pressing `+` and decreased by pressing `-`, respectively. `Esc` terminates the program. The GUI displays the entire game world scaled to the dimensions of the screen or the displayed window. Full-screen display can be switched of by using the `-window` command-line option.

Note, that running a game with a game world that contains computer-controlled worms only, may be difficult to debug as you may not be able to interact or terminate that game. For interactive debugging we advise that your game worlds contain at least one worm that is player controlled, i.e., one worm that does not have a program assigned. 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 22nd of May 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.

Your JAR file must include the source code of all the classes in your project. Before you submit your JAR, check whether it indeed contains everything that is needed, by importing it in a new project and executing the main program. When submitting via Toledo, make sure to press `OK` until your solution is submitted!