OGP Assignment 2017-2018: **Worms** (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 need not implement some (small) parts of the assignment, as indicated in the following section. In principle, you should work out 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 March 31, 2018. 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 three parts of this 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 8 or higher and follow the rules described in this document. The previous part of the assignment focussed on a single class *Worm*. In the second part, we extend *Worm*, and add additional classes and relationships between them. Note that certain aspects of the class `Worm` described in Part I change in this second part of the assignment.

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 the Git version control system. Instructions on how to obtain a private repository on GitHub, already populated with the provided GUI code (see section 4), as well as a short tutorial are posted on Toledo.

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 or higher, 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 Prof. Steegmans or Prof. Jacobs.

This text extends the assignment for the first part of the project. Portions of the original assignment that have not been changed are colored blue. Portions of the original assignment that have been changed are colored red. New parts are simply printed in black.

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 game released in 1995 by Team17 Digital.

In this second part of the assignment, we extend the class `Worm` from part one, and introduce new classes such as `World` and `Food`. However, 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 formal way only. 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 `0.0` to `Double.MAX_VALUE` (both inclusive) for all worlds.

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. The association between a world and its game objects must be worked out in a bi-directional way. For now, the game world only contains worms and portions of food. Other kinds of game objects will be introduced in the 3rd part of the assignment. All current and future game objects will be circular entities. If a game object is located in a world, its circle must lie fully within the bounds of that world. As soon as an entity is partially or completely outside the boundaries of its world, that entity shall be removed from that world. The class `World` shall provide methods for adding and removing worms and portions of food.

Those methods must be worked out defensively. The class must also offer facilities to check whether a given world contains a given worm, respectively a given portion of food. These methods must operate on the collection of game objects stored in the world, and must return their result in constant time. Finally, it must be possible to ask a world all the worms, respectively all the portions of food it contains.

Worms and portions of food may overlap with other game objects. However, game objects cannot be located partially or completely on impassable terrain of their world. Moreover, at stable times, worms and portions of food shall be located adjacent to impassable terrain. A circular area with radius σ is adjacent to impassable terrain if the area itself is passable and the distance from the area's boundary to at least one impassable location is less than or equal to $\sigma \cdot 0.1$. The class `World` shall further provide methods to determine whether a circular area with given center and given radius is passable, respectively adjacent to impassable terrain. These methods must be worked out in a total manner.

Finally, `World` must provide methods to start and to finish the game. When a game is started, a first worm starts its turn and performs player-controlled actions such as moving, jumping or falling. 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. As long as a game is played in a game world, it is not permitted to add further worms or portions of food to that game world. A game is won as soon as only worms that all belong to the same team are left in the game world, or as soon as only a single, stand-alone worm is left in the game world. The class `World` must provide a method to ask a world for the winner. No formal or informal documentation is required for all the methods related to the game. All these methods must be worked out in a total way.

The documentation for the class `World` must be worked out only in a formal way. In fact, this applies to all classes in the second part of the project, except for the class `Worm`, whose documentation must be worked out both formally and informally. Be careful not to use effect-clauses in the documentation of mutators and constructors, if a more declarative specification by means of postconditions is possible. As an example, the documentation of a method to move a worm will not reveal details concerning how worms move; it will only describe the net result of the method.

1.2 Worms

Each worm is located at a certain location (x, y) in a two-dimensional space. Both x and y are expressed in metres (m). If a worm is not located in a

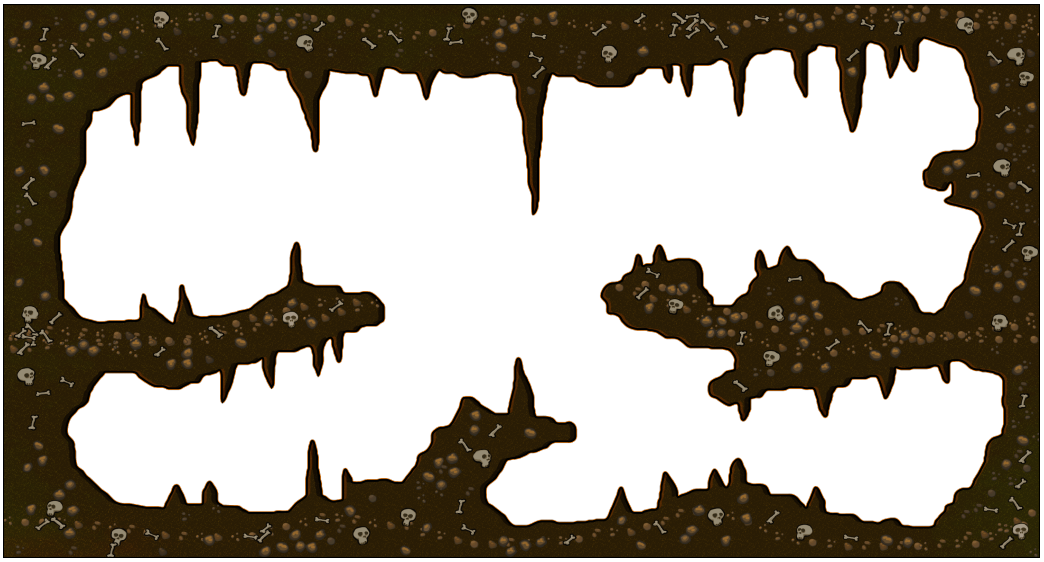

Figure 1: A game world.

world, it is located in a two-dimensional space that is unbounded in both directions, and that furthermore includes special areas where one or both coordinates are infinite. That is, the set of possible locations for a worm is $(\mathbb{R} \cup \{-\infty, +\infty\}, \mathbb{R} \cup \{-\infty, +\infty\})$. All aspects related to the location of a worm shall be worked out *defensively*.

Each worm faces a certain direction. The orientation of a worm is expressed as an angle θ in radians. For example, the orientation of a worm facing right is 0, a worm facing up has $\pi/2$ as its orientation, a worm facing left has an orientation equal to π and a worm facing down has $3\pi/2$ as its orientation. The orientation of a worm will always be in the range $0 \ldots 2\pi$, the latter value not included. 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 location. 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. All aspects related to a worm's radius must be worked out *defensively*.

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: $m = p \cdot (4/3 \cdot \pi\sigma^3)$.

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 using the predefined method `round` in `java.lang.Math`. If the mass of a worm changes, the maximum number of action points must be adjusted accordingly. 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 than zero. Whenever a worm is created, its current number of action points will have the maximum value. All aspects related to action points must be worked out in a total manner.

Each worm also has a number of hit points, which shall be represented by integer values. The number of hit points for a worm cannot be negative. If a worm's hit points are decremented to zero (or lower), that worm dies and must be removed from the game world in which it is located, if any. There is no upper bound on the number of hit points a worm can have. Whenever a worm is created, its hit points will be between 1000 and 2000. All aspects related to hit points must be worked out in a total manner.

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 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, world, location, direction, radius, mass, action points, hit points, and team of a worm.

1.3 Turning

The class `Worm` must provide a method `turn` to change the orientation of the worm by adding a given angle to the current orientation. As this method affects the orientation, it must be worked out nominally. This means that the given angle must be such that the resulting angle is in the specified range for the orientation of a worm. Active turning 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.

1.4 Moving

The class `Worm` shall further provide a method `move` to change the location of the worm based on the current location, orientation and 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 worm's direction 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 location (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 $\mathsf{div}(\theta, s)$ [1], where s is the direction in which the worm actually moves. More formally, $s = \text{atan2}(y' - y, x' - x)$ and $d = \sqrt{(x - x')^2 + (y - y')^2}$, with $\theta - 0.7875 \leq s \leq \theta + 0.7875$ and $0.1m \leq d \leq \sigma$. More in particular, worms will step in such a way that the ratio of the travelled distance d over the divergence $\mathsf{div}(\theta, s)$ will be as large as possible. [2] 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 adjacent to impassable terrain exists, the worm shall remain at (x, y). If, on the other hand, 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[3]. As the method `move` affects the location of the worm, it must be worked out defensively.

[1] In this text, the divergence $\mathsf{div}(\alpha, \beta)$ between two angles α and β must be understood as the smallest positive angle between them, i.e., $\min(\{|\alpha - \beta + k \cdot 2\pi| \mid k \in \mathbb{Z}\})$.

[2] Notice that worms will only deviate from their orientation if they can not reach a proper location in the direction set out by their orientation.

[3] For students working alone, worms will simply not move if they cannot reach a location adjacent to impassable terrain.

If a move of a worm ends in a location adjacent to impassable terrain, and the worm's area partially or fully overlaps with some other worm, a number of points will be subtracted from the hit points of both worms. The total number N of subtracted hit points will be a random number in the range 1..10. The smallest worm will lose a number of hit points N_1 equal to $N/(\sigma_2/(\sigma_1 + \sigma_2))$ rounded to the nearest integer, in which σ_2 is the radius of the largest worm, and σ_1 is the radius of the smallest worm. The largest worm will lose $N - N_1$ hit points. If the moving worm's area overlaps with several worms, the above formula is applied separately to this worm and each of the overlapping worms. If a worm moves to a new location that is not adjacent to impassable terrain, that move will not have an impact on overlapping worms that are positioned near that new location.

The cost of movement shall be proportional to the horizontal and vertical component of the trajectory such that a horizontal step of 1 meter is at the expense of 1 action point, while a vertical step of 1 meter incurs a cost of 4 action points. The total cost of a step of 1 meter in the direction θ 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 shall not move if they do not have enough action points to cover the move.

In order to manage the complexity of the method `move`, you will introduce a number of auxiliary methods. In general, no method body should exceed 20 lines of code (more or less). One step to manage the complexity of moving is to introduce a method `getFurthestLocationInDirection`. That method will return the location the farthest away from the current location of a worm to which that worm can move in the world in which it is positioned, following a given direction and not exceeding a given maximum distance. On its road to the resulting location, the given worm will always be positioned on passable terrain, but not necessarily on terrain adjacent to impassable terrain. The resulting location may also be not adjacent to impassable terrain. You must work out this auxiliary method, and preferably others that you will introduce yourself.

If you want to get a score of 18 or more for this course, you must work out a *formal documentation* of the method `move` and of all auxiliary methods that you have introduced to manage the process of moving a worm, including the method `getFurthestLocationInDirection`. You are not allowed to use effect clauses in the documentation of those methods. You may use quantifications over intervals of real numbers as in `for each x in [a,b]: p(x)`. Those intervals can be closed (`[a,b]`), half-open (`[a,b[` or `]a,b]`) or open (`]a,b[`). If you do not want to go for the highest possible score, an informal documentation of all the methods related to moving is more than

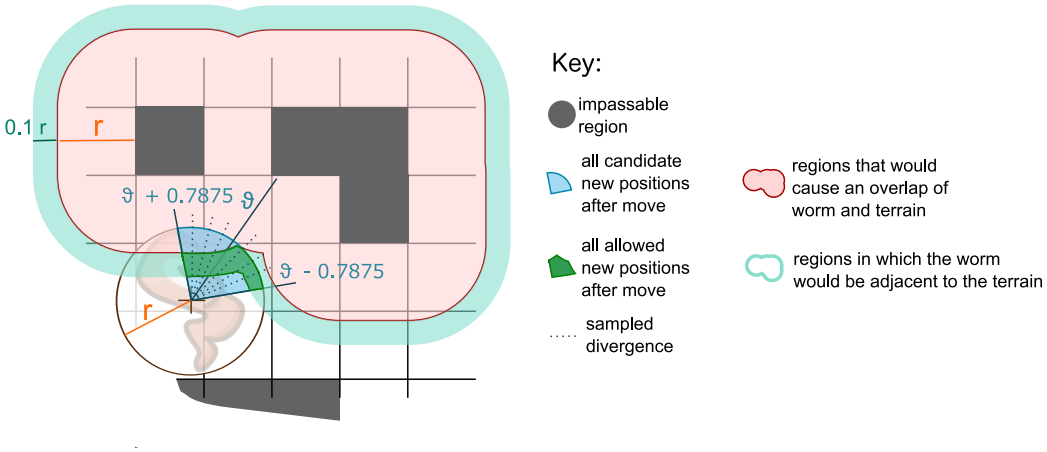

Figure 2: Illustration of a worm's movement.

good enough.

~~1.5 Falling~~

~~*This part of the assignment is only mandatory for teams of two students.*~~

~~Worms may also move passively, e.g. fall down a chasm. The class `Worm` shall implement a method `fall` to change the location of the worm as the result of a free fall from the current location. 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.~~

~~If a fall of a worm ends in a location adjacent to impassable terrain, and the worm's area partially or fully overlaps with some other worms, the falling worm will get half the hit points of each worm that it has fallen upon.~~

~~Note that falling should not happen automatically after moving, but must be invoked explicitly.~~

1.6 Jumping

Worms may also jump along ballistic trajectories. The class `Worm` shall provide a method `jump` to change the location of the worm as the result of a jump from the current location (x, y) and with respect to the worm's orienta-

tion θ and the number of remaining action points APs. All methods related to jumping 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 standard acceleration in the game world which is equal to 5.0 m/s^2. [4] From this, we can compute the initial velocity of the worm as $v_0 = (F/m) \cdot 0.5\ s$. The formula to calculate the initial velocity may change in the future. However, the resulting value will always be nonnegative and finite.

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 reaches a location that is adjacent to impassable terrain and at least a distance equal to the worm's radius away from (x, y), or when the worm leaves the map. A jump is executed in small steps. You may assume that worms do not move through impassable terrain during such a step. Jumping consumes all remaining action points of a worm. A worm that has no action points left or that is oriented downwards must not jump.

If a jump of a worm ends in a new location adjacent to impassable terrain, and the worm's area partially or fully overlaps with some other worm, both worms will have a fight. In that fight one of both worms will hit the other worm exactly once. A coin will be tossed to decide which worm will be the attacker and which one will be the attacked worm. The victim will lose a random number of hit points in the range $1..N$, in which N is 10 times the ratio (rounded up to an integer) of the radius of the attacking worm over the radius of the attacked worm. If the jumping worm's area overlaps with several worms after a jump, the jumping worm will fight each of them in turn. If a worm is still at the same location after having attempted to jump, or if the jumping worm has left its world, no fights will take place.

The class `Worm` shall provide a method `jumpTime` that returns the effective time of a potential jump from the worm's current location. The class will also offer a method `jumpStep` that computes in-flight locations $(x_{\Delta t}, y_{\Delta t})$ of a jumping worm at any Δt seconds after launch. $(x_{\Delta t}, y_{\Delta t})$ may be computed as follows:

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

[4] We use a fictitious value for the standard acceleration instead of the standard acceleration on earth to make testing simpler.

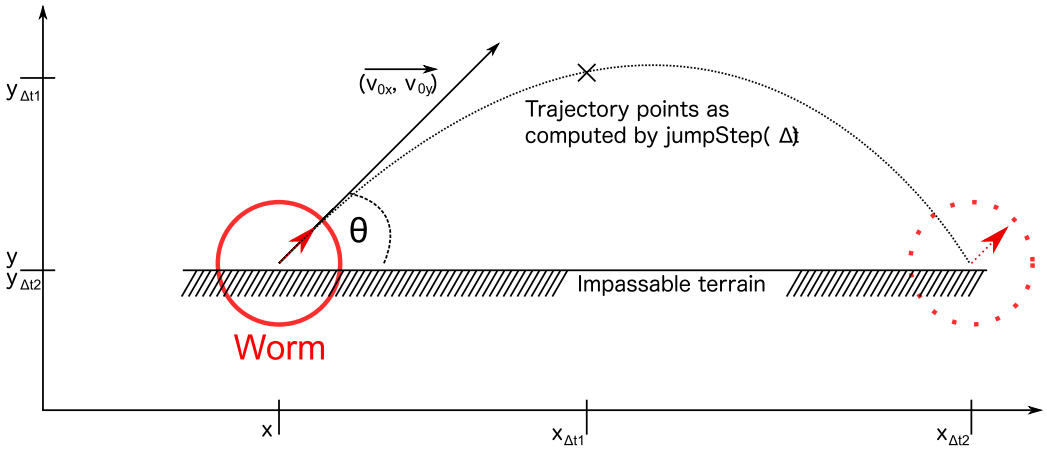

Figure 3: Illustration of a jumping worm's trajectory.

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.

1.7 Portions of Food

Worms may consume portions of food to grow in size. If a portion of food is not located in a world, it is located in a two-dimensional space that is unbounded in both directions, and that furthermore includes special areas where one or both coordinates are infinite. If a portion of food is located in a world, its circle must lie fully within the bounds of that world. All aspects related to the location of portions of food shall be worked out defensively.

The shape of a portion of food is a circle with finite radius σ (expressed in metres) centred on the portion's location. The radius of a food ration shall always be 0.20 m. Its mass m expressed in kilograms is derived from σ, assuming that a food portion has a spherical body and a homogeneous density p of 150 kg/m^3: $m = p \cdot (4/3 \cdot \pi\sigma^3)$.

Worms can eat portions of food if the worm's body partially overlaps with that portion. For that purpose, the class of worms will introduce a method `eat()` to make a worm eat a single portion of food. If the worm's body against which that method is applied overlaps with at least one portion of food, the worm's radius shall increase by 10% and one of the overlapping portions of food shall be destroyed and removed from the game world. Because of its increased size, a worm that has eaten some food will reposition itself such

that it is still adjacent to the same impassable terrain as before. If no such position can be found, the worm will explode. This is for instance the case if the worm is sandwiched between impassable terrain at opposite sides of the worm. If because of a repositioning, the worm's circular area is no longer fully included in the worm's world, the worm will leave that world. You must only work out an implementation of the method `eat()` and of any auxiliary methods that you may introduce to support eating food. No formal nor informal specification is required.

1.8 Teams

This part of the assignment is only mandatory for teams of two students.

Worms may be part of a team. A team may group worms from different worlds. All worms in a team must be alive. Moreover, at the time of joining a team, a worm's mass cannot be less than half the mass of any worm already in the team, nor can it be larger than twice the mass of each member of the team. The members of a team shall be ordered alphabetically, such that methods to add a worm to a team, to remove a worm from a team and to check whether some worm is part of a team all have logarithmic execution time. There should also be a method that returns an alphabetically sorted list of all members of a given team in linear time.

Worms fighting together in a team jointly carry victory after the game has been started and when only worms belonging to the same team remain in a world. Worms in a team may still damage and destroy each other. Teamed worms may co-exist with individual worms that do not belong to any team. A game world may contain up to 10 teams. The world, if any, to which a team belongs is set at the time the team is created, and cannot be changed as long as the team is alive. The association between worlds and teams may only be supported in the direction from a world to its teams. In other words, teams themselves have no knowledge of the world in which they are in.

Each team shall have a name. The spelling rules for names of teams are identical to those for names of worms, and will always stay identical. Contrary to worms, however, the name of a team cannot change. Moreover, all the worms in a team must have a different name.

The class `Team` shall provide methods to add a series of worms to a team and to remove a series of worms from a team. The class will also offer a method to merge two teams, meaning that all the worms of the one team become member of the other team. Finally, the class will offer an inspector that returns a set of all the worms in a team.

All aspects of the class `Team` must be worked out defensively. If you want to get a score of 18 or more for this course, you must work out a *formal*

~~*documentation* of all the methods applicable to teams. You are not allowed to use effect clauses in the documentation you write for this part. If you do not want to go for the highest possible score, an informal documentation of all the methods related to teams is more than good enough.~~

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[5]

$$0.2000000000000000011102230246251565404236316680908203125$$

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:

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

[5] You can check this by running `System.out.println(new BigDecimal(1.0/5.0))`.

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

You must work out a proper set of tests for the methods to move a worm and for the auxiliary method to find the furthest possible location in some direction. Include that 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. Along with the assignment for the final part of the project, we will distribute an extended set of tests that covers the entire project. That suite will yield a final score that will be integrated in your final score for the course.

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 worm to the latest created team, F to add portion of food to the world, and S to start the game. The in-game command keys

are Tab ⇥ for switching worms (finishes a worm's turn), ← and → arrow key (followed by pressing ⏎) to turn, ↑ to move forward, n to change the worm's name, j to jump, and e to eat. 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.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the classes mentioned 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 `Facade`. As described in the documentation of `IFacade`, the methods of your `Facade` class 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 April 2018 at 11:59 PM. Follow the instructions on Toledo (under `Project:Assignment`) to submit a proper JAR.