OGP Assignment 2015-2016: The Hillbillies (Part III)

This text describes the third part of the assignment for the course *Object-oriented Programming* (OGP). There is no exam for this course. Therefore, if you worked out part two of the project with some partner, you must work out this final part of the project with the same partner. You are not allowed to start working with a new partner for part three. As before, if conflict arise within the team, you must report them to ogp-inschrijven@cs.kuleuven.be before the 1st of May 2016. Both members of the team must then finish the project on their own. There are some reductions in the third part of the assignment that apply to all students that are working individually on the project.

In the course of the assignment, we will create a simple game that is loosely based on *War Craft* and *Dwarf Fortress* – real-time strategy games that involve combat as well as manipulation of the game world. Note that several aspects of the assignment will not correspond to any of the original games. 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 this 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 not impose to use nominal programming, total programming or 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. The goal is not to hand it the best possible real-time strategy game. Therefore, the grades for this assignment do not depend on correctly implementing functional requirements only. We will pay attention to documentation, accurate specifications, re-usability and adaptability. After handing in your solution to the first part of the assignment, you received 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

This assignment aims to create a simulation video game that is loosely based on *War Craft* or *Dwarf Fortress*. In The Hillbillies, the player (mostly) indirectly controls a number of hillbilly `Unit`s. The goal of the game is to maintain the safety of these hillbillies in a hostile three-dimensional game world, destroying hostile `Unit`s in combat or avoiding them by means of clever manipulation of the world. In the first part of the assignment we focused on a single class `Unit` that implements a basic type of in-game character with the ability to move around, interact with other such characters and manipulate the game world. The second part of the assignment emphasised on associations, adding a number of new classes and interactions to the game. This third part introduces a `Scheduler` class and a simple programming language to control `Unit`s of a faction. While this part of the assignment explicitly focuses on inheritance, generics and functional programming in Java, you may have used some of these concepts in previous iterations already. If not, you are encouraged to re-factor your solution to improve on readability

and re-usability of your code. As in the previous iteration, we have indicated new and changed requirements in blue.

Of course, your solution may contain additional helper classes beyond the ones specified in this text,(in particular classes marked *@Value*). In the remainder of this section, we describe the classes `Unit`, `World` and `Scheduler` in more detail. Unless explicitly stated otherwise, all aspects of your implementation of the classes `Unit`, `Scheduler` and `Task` shall be specified both formally and informally. Classes other than `Unit`, `Scheduler` and `Task`, that are specified in this document must be documented informally only. For your support, you will be provided a JAR file containing the user interface for the game together with some helper classes.

1.1 The Class `World`: A Game World

The Hillbillies is played in a cubical game `World` that is composed of X times Y times Z adjointly positioned, non-overlapping *cubes*. The dimensions of a game `World` do not change during the lifetime of this `World`. Each cube is located at a fixed position, denoted by a triple of integer values (x_c, y_c, z_c). The position of the bottom-left-back cube of the game world shall be $(0, 0, 0)$. The position of the top-right-front cube of the game world shall be $(X - 1, Y - 1, Z - 1)$. For the purpose of calculating locations, distances and velocities of game objects, each cube shall be assumed to have a side length $l_c = 1\ m$.

1.1.1 Terrain

A game world shall have geological features, including passable terrain (air, workshop) and impassable (i.e., solid) terrain (rock and wood), that are associated with entire cubes. If a cube of the game world is not assigned a feature explicitly, "air" should be used as the default. Importantly, geological features of the game world may change during the lifetime of the `World`.

Solid terrain cubes must always be attached on (at least) one of their side planes to another solid cube's side plane or to the borders of the `World`. More specifically, a solid cube C at (x_c, y_c, z_c) can be attached to another solid cube at *(a)* $(x_c \pm 1, y_c, z_c)$, *(b)* $(x_c, y_c \pm 1, z_c)$ or *(c)* $(x_c, y_c, z_c \pm 1)$. We call the six cube that match the above specification the *directly adjacent* cubes of C. Groups of directly adjacent cubes may include (at least) one cube that connects to the borders of the game world (i.e., $(0|X-1, ?, ?)$, $(?, 0|Y-1, ?)$, $(?, ?, 0|Z-1)$). Every cube of solid terrain from which there is no path of directly adjacent solid terrain cubes leading to a cube at the borders of the game world, shall cave-in within at most 5 s of game time.

When a solid cube collapses – either as a result of a cave-in or in consequence of a `Unit` manipulating the `World`, the geological property of that cube shall immediately change to "air" and, with a probability of $P = 0.25$, a `Boulder` (rock cube) or a `Log` (wood cube) shall be created at the centre of the collapsed cube.

An implementation of an algorithm to efficiently check if a cube is anchored at the borders of the `World` is provided[1]. You may (but do not have to) use this implementation in your code.

1.1.2 Game Objects

A `World` may contain an unlimited number of such `Boulder`s and `Log`s. A `World` may further contain up to 100 `Unit`s (cf. Sec. 1.2). Each `Unit` must always belong to a faction and there must be no more than 50 units belonging to the same faction. There shall be no more than 5 different active (i.e., non-empty) factions in a `World`. The `World` shall silently reject requests to add new units or factions while the respective limits are reached.

The game world shall have a method to spawn a new unit with random initial attributes at a random position of the game world. Initial positions shall be chosen such that a new unit is occupying a passable cube at (x_c, y_c, z_c), where $(x_c, y_c, z_c - 1)$ must be a solid cube or $z_c = 0$. When a new unit is created, it starts a new faction as long as the maximum number of active factions is not reached; otherwise, the unit joins the active faction with the smallest number of units.

Game objects such as `Unit`s, `Boulder`s or `Log`s shall be removed from the game world upon death or as they are consumed in certain activities.

1.1.3 Game Over

The game is never over. Players are encouraged to define victory conditions for themselves. Options for such conditions may include

- Control 50 units in the first faction.
- Defeat all enemy factions for 30 minutes of game time.
- Control 5 units with maximum strength and agility.

Victory conditions do *not* have to be implemented in the game.

[1] The algorithm is implemented in class `hillbillies.util.ConnectedToBorder`.

1.1.4 Implementation

The game `World` shall provide methods to inspect each individual cube, listing the terrain type and objects currently occupying that cube. The game world shall further provide methods to list all factions, `Units` (individually or per faction), `Boulders` and `Logs` currently present in the game world. A method `advanceTime` must be implemented by the game `World`. This method shall control timed behaviour of the `World` and invoke the `advanceTime` methods of all game objects that are part of the `World`. All aspects of the class `World` shall be worked out defensively.

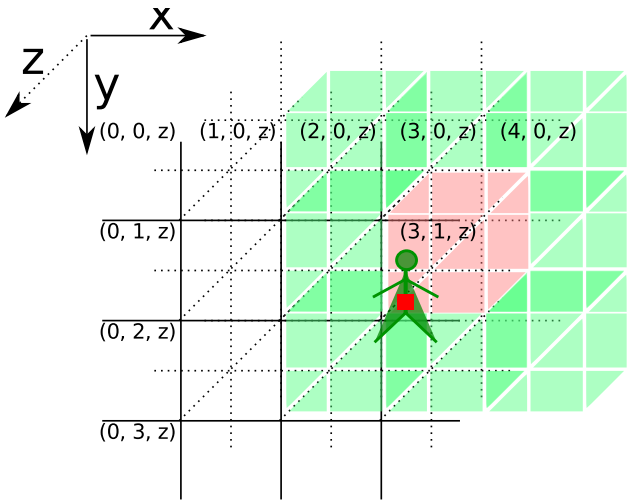

Figure 1: A section of the game world in The Hillbillies and a `Unit`. The given section represents a top-down view on one z-level of the world. The `Unit` sits (or lies) on the edge of cube $3, 1, z$ and we say that it is occupying this cube (shaded in red). The actual position of the `Unit` is indicated by the red square and could, for example, be $3.1, 1.1, z.9$. The `Unit` can move (cf. Sec. 1.2.2) to neighbouring cubes on the same z-level (shaded in green). The `Unit` can further move to neighbouring cubes at $z \pm 1$. These cubes are not highlighted in the figure. They comprise of cubes with the same x and y coordinates as the red and green cubes at directly overlying and underlying z-levels. The boundaries of the game world restrict a `Unit`'s movement options.

1.2 The Class `Unit`

Hillbilly `Units` are considered to be cubical objects that occupy a position (x, y, z) in the game world. Each of them has a name, and a certain weight, strength, agility and toughness. A `Unit` must always belong to a certain faction.

While the game world is using integer coordinates, the position of `Unit`s shall be treated as double precision floating-point numbers. That is, use Java's primitive type `double` to compute and store these values. Intuitively, a `Unit`'s position (x, y, z) denotes the location of the centre of the `Unit`. Rounding x, y and z down to integer numbers shall yield the position (x_c, y_c, z_c) of a cube of the game world that is said to be occupied by the `Unit`. These concepts are illustrated in Fig. 1. A `Unit` shall never be positioned outside the game world and the components of a `Unit`'s position must at all times be valid numbers. Furthermore, the game `World` cube at the position of the `Unit` must feature a passable (i.e., not solid) terrain type. The class `Unit` shall implement methods to inspect a `Unit`'s position and the position of the game world cube occupied by the unit. Aspects of `Unit` that are concerned with a `Unit`'s position in the game world shall be worked out defensively.

A `Unit`'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. However, letters, quotes and spaces will always be legal characters. All aspects related to a `Unit`'s name must be worked out defensively.

A `Unit`'s weight, strength, agility and toughness influence how fast that `Unit` can move, work, and how it behaves in combat. All four attributes may change during the `Unit`'s lifetime and shall be worked out using total programming and integer numbers with values ranging from 1 to 200, inclusively. Importantly, the initial values (i.e., when a new `Unit` object is created) of these attributes shall be in the range of 25 to 100, inclusively, and the weight of a `Unit` must at all times be at least $\frac{strength+agility}{2}$.

Based on their primary attributes, `Unit`s have a maximum number of hitpoints and a maximum number of stamina points, both being determined as the unit's $200 \cdot \frac{weight}{100} \cdot \frac{toughness}{100}$, rounded up to the next integer. Hitpoints and stamina points can be consumed and regained by means of certain activities. Thus, each `Unit` further has current number of hitpoints and a current number of stamina points which shall always be greater or equal to zero and less or equal to the respective maximum number. Aspects of `Unit` that are concerned with hitpoints and stamina points must be worked out using nominal programming.

A `Unit` shall also have an orientation θ, i.e., a direction the `Unit` is facing. This orientation shall be given as an angle in radians and defaults to $\theta_0 = \frac{\pi}{2}$. The orientation must be updated when the `Unit` is moving or executing other activities (cf. Sec. 1.2.2 ff.). Your implementation of `Unit` must implement a

method to inspect the current orientation of a unit. All aspects of `Unit` that concern the orientation attribute shall be worked out using floating-point numbers and total programming.

`Units` can conduct a number of activities, interacting with the game world and other units. In particular, `Units` can move around, work on the cube they are standing on, rest, and attack other units. In the following we will describe these activities in detail. Importantly, the execution of these activities is dependent on *game time*, which is measured in seconds. The class `Unit` shall provide a method `advanceTime` to update the position and activity status of a `Unit`, based on that `Unit`'s current position, attributes and a given duration Δt in seconds of game time. This duration Δt shall never be less than zero and must always be smaller that 0.2 s. Unless explicitly specified in the following sections, `advanceTime` and other methods related to the interaction of `Units` with the game world shall be worked out defensively. It is not required to provide formal documentation for the method `advanceTime`.

As `Units` conduct activities, they shall be able to gain experience points, which are to be treated as integer values. For every 10 experience points a `Unit` gains, *one* of that `Unit`'s strength, agility or toughness attributes shall be increased by one.

1.2.1 (Not) Falling

`Units` shall be positioned in the game `World` such that the `Unit` always occupies a passable terrain cube. `Units` further need to stand on (or hang on) solid ground. `Units` are remarkable climbers and are perfectly able to move along a vertical (or even an overhanging) cliff as long as the `Unit` has solid rock or a tree in reach to hold on to. That is, for a `Unit` currently occupying a passable cube with the coordinates (x_c, y_c, z_c), there must be a *neighbouring cube* $(x'_c, y'_c, z'_c) = (x_c \pm 0..1, y_c \pm 0..1, z_c \pm 0..1)$ of impassable (solid) terrain or the `Unit` will fall.

Falling is a special case of basic movement as specified in Sec. 1.2.2: The falling `Unit` shall move with a constant velocity of $\vec{v} = \langle 0, 0, -3 \rangle$ towards the centre of the cube at $(x_c, y_c, z_c - 1)$ until the `Unit` reaches a position that is directly above a solid cube (i.e., for a falling `Unit` currently occupying a non-solid cube at (x_c, y_c, z_c), the cube at $(x_c, y_c, z_c - 1)$ must be solid for the unit to stop falling), or until $z_c = 0$. `Units` cannot start sprinting while falling.

Falling `Units` shall lose 10 hitpoints per Z-level they fall.

1.2.2 Basic Movement

`Unit`s can move from their current position (x, y, z) to the centre of any neighbouring cube (x', y', z'). As depicted in Fig. 1, for a `Unit` currently occupying a cube with the coordinates (x_c, y_c, z_c), the target cube $(x'_c, y'_c, z'_c) = (x_c \pm 0..1, y_c \pm 0..1, z_c \pm 0..1)$, and $x' = \frac{l_c}{2} + x'_c$, $y' = \frac{l_c}{2} + y'_c$ and $z' = \frac{l_c}{2} + z'_c$, provided that (x', y', z') is within the boundaries of the game world. Here, l_c denotes the length of any side of a cube of the game world, as defined in Sec. 1.1. Importantly, `Unit`s only move through cubes that feature a passable terrain type and that are neighbouring cubes of solid terrain.

A `Unit`'s movement speed is determined by the relative position of the target cube and the `Unit`'s weight, strength and agility. We compute a `Unit`'s base speed in m/s as $v_b = 1.5\frac{strengh+agility}{200\frac{weight}{100}}$. For a `Unit` "walking" from some (x, y, z) to a target position (x', y', z') we determine that `Unit`'s walking speed v_w as follows:

$$v_w = \begin{cases} 0.5v_b, & \text{if } z - z' < 0 \\ 1.2v_b, & \text{if } z - z' > 0 \\ v_b, & \text{otherwise} \end{cases}$$

A `Unit` may also choose to do a sprint and move at $v_s = 2v_w$. Sprinting, however, exhausts units quickly, reducing their current stamina by 1 points for each 0.1 second of game time they run. A `Unit` may only start sprinting if that `Unit` is currently moving and the `Unit`'s current stamina is greater than zero. A sprinting `Unit` must stop sprinting as that `Unit`'s stamina reaches zero; the `Unit` may stop sprinting at any time. A `Unit` that stops sprinting before it has reached it's target position (x', y', z') will continue walking towards (x', y', z') with v_w.

The class `Unit` shall provide a method `moveToAdjacent` to initiate movement to a neighbouring cube. Once a unit started moving, subsequent invocations of `advanceTime(`Δt`)` shall lead to updates of that `Unit`'s position. To compute intermediate positions of a `Unit`, we first need to determine that `Unit`'s velocity:

$$d = \sqrt{(x' - x)^2 + (y' - y)^2 + (z' - z)^2}$$

$$\vec{v} = \langle v_x, v_y, v_z \rangle = \langle v_? \frac{(x' - x)}{d}, v_? \frac{(y' - y)}{d}, v_? \frac{(z' - z)}{d} \rangle$$

Here, $v_?$ must be replaced with v_w or v_s, depending on whether the `Unit` is enjoying a quiet stroll in the countryside or hastes towards its destiny. Now the `Unit`'s new position after moving some Δt seconds can be computed as

$$(x, y, z)_{new} = (x, y, z)_{current} + (\vec{v} \cdot \Delta t)$$

$$= ((x_{current} + v_x \Delta t), (y_{current} + v_y \Delta t), (z_{current} + v_z \Delta t))$$

A `Unit` shall stop moving as soon as it reaches or surpasses (x, y, z) within the current time step Δt; the `Unit`'s position shall then be set to (x', y', z'), exactly.

Based on the `Unit`'s velocity, we can also update that `Unit`'s orientation: the orientation attribute of a `Unit` shall be set to $\theta = atan2(v_y, v_x)$, using Java's built-in arctangent function with two arguments.

Movement to a neighbouring cube, unless the `Unit` is currently falling, is only interrupted if the `Unit` is attacked. In that case the `Unit` shall stop moving and execute its defense behaviour (cf. Sec. 1.2.5). The movement of a falling `Unit` cannot be interrupted, neither by fighting nor by any other action. Invocations of `moveToAdjacent` of a `Unit` that is already moving, as well as triggering any other behaviour, shall have no effect. `Unit` shall provide methods to start and stop sprinting and to inspect a `Unit`'s current movement status.

`Unit`s shall gain 1 experience point for every completed movement step. No experience points shall be awarded if the movement was interrupted.

1.2.3 Extended Movement and Path Finding

`Unit`s typically move to a specific target cube that might be further away than directly neighbouring cubes. Therefore, `Unit` shall provide a method `moveTo` that allows for initiating a complex movement activity that spans multiple `moveToAdjacent`-steps. For a `Unit` that aims to move from some current cube position (x_c, y_c, z_c) to an arbitrary cube (x'_c, y'_c, z'_c), we propose an algorithm in Listing 1. However, students are free to choose and implement other (i.e., faster or more reliable) path finding algorithms such as A^* or Dijkstra's.

Listing 1: Pseudocode of a simple path finding algorithm.

```
let Q be an empty Queue;

function search((position c_0, int n_0))
   let l be a List of cube positions such that each (c ∈ l)
      is neighbouring c_0 ∧
      is of passable terrain ∧
      is neighbouring solid terrain ∧
      (c, n|n ≤ n_0) ∉ Q;
   for each (c ∈ l) Q.append((c, n_0 + 1));

// main loop: path needs to be recomputed to account for
// terrain changes; this can certainly be optimised.
while ((x_c, y_c, z_c) ≠ (x'_c, y'_c, z'_c)) do
   Q.append(((x'_c, y'_c, z'_c), 0));
   while ((x_c, y_c, z_c) ∉ Q ∧ Q.has_next()) do
      (c_0, n_0) = Q.get_next();
      search((c_0, n_0));
   done
```

```
  if ((x_c, y_c, z_c) ∈ Q) then
    let (next, n) ∈ Q such that
      next is neighbouring (x_c, y_c, z_c) ∧
      for all (c, m) ∈ Q where c is neighbouring (x_c, y_c, z_c), n ≤ m;
    moveToAdjacent(next);
  else
    terminate pathing;
  fi
done
```

Importantly, pathing to a distant target location can be interrupted by other activities, such as a `Unit`'s need to rest or enemy interaction. In this case, the interrupted `Unit` shall resume pathing to the target position as soon as the interrupting activity is finished. Pathing may be terminated if the target position is not reachable. Invocations of `moveTo` of a `Unit` that is already pathing to some location shall update that `Unit`'s target location with a new target cube.

1.2.4 Work

`Unit` can conduct activities such as digging in the ground, chopping wood, carry objects around, or operating workshops. The class `Unit` shall provide a method `work` to conduct a generic labour at a specified cube position, which must be the unit's current position or a neighbouring cube. The game time needed for finishing a work order depends on the `Unit`'s strength: the `Unit` shall be busy for $\frac{500}{strength}$ s. Floating-point numbers must be used for computing the duration and progress of work activities. Work activities can be interrupted by assigning new tasks to a unit, by fighting or by the unit's requirement to rest. If an activity is interrupted, the interrupted activity shall have no effect on the working `Unit` (other than the game time that passed) or the game `World`.

Depending on the the features an objects present on the target cube, completing a `work` order shall yield results as listed below. Conditions shall be checked upon completion of the work order and in order of appearance: only the first activity for which all conditions are met shall be executed (i.e., `switch-case` semantics). If a work order is completed and none of the conditions below is met, the completed job shall have no effect on the `Unit` or the `World`.

`Unit` carries a `Boulder` or `Log`: The `Boulder` or `Log` is dropped at the centre of the cube targeted by the labour action.

Target cube is *Workshop* and one `Boulder` and one `Log` are available on that cube: The `Unit` will improve their equipment, consuming one `Boulder` and one `Log` (from the workshop cube), and increasing the `Unit`'s weight and toughness.

`Boulder` **present on target cube:** The `Unit` shall pick up the `Boulder`.

`Log` **present on target cube:** The `Unit` shall pick up the `Log`.

Target cube is *Wood*: The cube collapses as described in Sec. 1.1.1, leaving a `Log`.

Target cube is *Rock*: The cube collapses as described in Sec. 1.1.1, leaving a `Boulder`.

Importantly, a `Unit` that picks up a `Boulder` or `Log` shall, for the time it carries that `Boulder` or `Log` have an increased weight computed as the `Unit`'s weight plus the weight of the carried object. The temporary weight of a unit may exceed the maximum weight specified earlier.

`Units` shall gain 10 experience points for every completed work order. No experience points shall be awarded for interrupted activities.

1.2.5 Fighting

A `Unit` A can attack other `Units` that occupy the same or a neighbouring cube of the game world. To do so, A and a defending `Unit` D must belong to different factions. D has a chance to either dodge or block the attack, depending on both `Unit`'s strength and agility attributes. If D fails to either dodge or block the attack, D will suffer damage proportional to A's strength. Conducting an attack shall last 1 s of game time. Defensive actions are instantaneous responses to an attack and require no game time to conduct. A defending `Unit` shall perform its defensive actions immediately when it is attacked and be available to defend against multiple simultaneous ongoing attacks. The class `Unit` shall implement the specified behaviour in two methods `attack` and `defend`. `Units` shall gain 20 experience points for every successful attempt at dodging, blocking or attacking. No experience points shall be awarded for unsuccessful attempts.

Dodging. An attacked `Unit` D will always first try and evade the attack by jumping away. The probability for successfully dodging an attack shall be computed as $P_d = 0.20\frac{agility_D}{agility_A}$. If D succeeds in dodging the attack, D shall suffer no damage and shall be moved instantaneously to a random position $(x'_D, y'_D, z'_D) \neq (x_D, y_D, z_D) = (x_D \pm 0..1, y_D \pm 0..1, z_D)$. (x'_D, y'_D, z'_D) must be a valid position featuring passable terrain within the boundaries of the game world. Note that (x'_D, y'_D, z'_D) refers to D's double-precision position. Thus, some $x_D \pm 0..1$ can refer to a new x-position on the same or a neighbouring cube.

Blocking. If an attacked `Unit` D fails to dodge a blow, it will next try to parry the attack. The probability for successfully blocking an attack shall be computed as $P_b = 0.25\frac{strength_D + agility_D}{strength_A + agility_A}$. If D succeeds in blocking the attack, D shall suffer no damage.

Taking Damage. If an attacked `Unit` D fails to dodge or block the attack, D shall suffer damage in terms of a reduction of D's hitpoints by $\frac{strength_A}{10}$.

A's and D's Orientation. Two `Units` A and D fighting each other must also update their orientation θ so that they are facing each other. We can compute and update θ based on the x and y components of the `Unit`'s positions:

$$\theta_A = atan2((y_D - y_A), (x_D - x_A))$$
$$\theta_D = atan2((y_A - y_D), (x_A - x_D))$$

1.2.6 Resting

As `Units` get exhausted or injured, they can rest to recover hitpoints and stamina. A resting unit will recover $\frac{toughness}{200}$ hitpoints or $\frac{toughness}{100}$ stamina points per 0.2 s of game time it spends resting. More specifically, when resting a `Unit` shall always first recover hitpoints until it has reached the `Unit`'s maximum number of hitpoints, and then recover stamina points. If a `Unit` starts resting, it will always rest for at least as long as it takes that `Unit` to recover one hitpoint. This initial recovery period is only interrupted if the `Unit` is fighting; in that case, neither hitpoints nor stamina are recovered. After the initial period, the `Unit` shall continue resting until it has recovered all hitpoints and stamina points, or until the `Unit` is assigned a new task. `Units` shall automatically rest once every three minutes of game time. The class `Unit` shall implement a method `rest` that initiates resting.

1.2.7 Death

`Units` die if their current number of hitpoints reaches zero. Death units can no longer conduct any activities and all ongoing activities of such a unit shall be interrupted or terminated immediately. If the `Unit` was carrying any objects, these objects shall be dropped at the `Unit`'s current position.

1.2.8 Default Behaviour

`Units` shall have a default behaviour of picking and executing the highest priority task from the `Unit`'s faction's `Scheduler`. If no such task is available, `Units` shall choose activities at random, as specified below. When a `Unit` is

not currently conducting an activity, that `Unit` may arbitrarily choose to *(a)* move to a random position within the game world, *(b)* conduct a work task, *(c)* fight potential enemies, or *(d)* rest until it has fully recovered hitpoints and stamina points If a `Unit` is currently moving, it may choose to sprint until it is exhausted.

The default behaviour can be activated and deactivated for each `Unit` individually. To facilitate this, the class `Unit` shall implement two methods `startDefaultBehaviour` and `stopDefaultBehaviour`.

1.3 The Classes `Boulder` and `Log`

`Boulder`s and `Log`s are raw materials that are introduced into the game `World` by cave-ins or `Unit`s manipulating the world. As such, `Boulder`s and `Log`s occupy a double-precision position (x, y, z) in the game world. They further have an integer weight between $10 and 50$, inclusively. The weight of a `Boulder` or `Log` shall be assigned at random upon creation of the object; the weight shall never change during the lifetime of the object.

This position must always be located on a passable (non-solid) cube at (x_c, y_c, z_c), for which either $z_c = 0$ or which is located directly above a solid cube (at position $(x_c, y_c, z_c - 1)$). `Boulder`s and `Log`s do not actively interact with the game world. They do, however, fall if the above conditions are not met. Once a `Boulder` or `Log` starts falling, it behaves as specified in Sec. 1.2.1 (without losing hitpoints, of course). Thus, the classes `Boulder` and `Log` must implement a method `advanceTime`.

`Boulder`s and `Log`s can further be carried around by `Unit`s. For this purpose, the weight of the carried `Boulder` or `Log` shall be added to the weight of the carrying `Unit`, affecting that `Unit`'s movement. `Boulder`s or `Log`s shall not be considered as present in the game `World` while they are carried around. Thus, no second `Unit` can attempt to carry a `Boulder` or `Log` that is already being carried. If a `Boulder` or `Log` is dropped it shall inherit the carrying `Unit`'s position and become part of the game `World` again.

All aspects of the classes `Boulder` and `Log` shall be worked out defensively.

1.4 The Classes `Scheduler`

A `Scheduler` shall be associated with each faction. `Scheduler`s manage a list of prioritised tasks to be executed by `Unit`s. The class Scheduler shall provide methods to add and remove one or more tasks. It must also offer a method to replace a task by some other task. If the task to be replaced is currently being executed, that task shall first stop executing. The class `Scheduler` shall also implement methods to check whether one or more tasks are all

part of that `Scheduler`, to return the highest priority task that is currently not being executed, to return all the tasks currently managed by a scheduler, as well as a more general method to return all the tasks that satisfy some condition (e.g. positive priority, being executing). The class must further offer an iterator that delivers tasks managed by a scheduler in descending order of their priority. Finally, the class Scheduler shall provide a method to mark a task as scheduled for execution by a specific `Unit`, and to reset this marking.

Importantly, a single task can be part of several schedulers at the same time. However, a task can only be assigned for execution to a single unit.[2] All aspects of the `Scheduler` shall be worked out using total or defensive programming.

Intuitively, an idle `Unit` will pick the highest-priority task that is not yet assigned to any other `Unit`. This task shall then be marked as assigned to the `Unit` that selected the task, and that `Unit` starts executing the task's individual activities. If execution of an activity is interrupted, the task shall again be marked as available and the task's priority shall be reduced. As `Unit`s become idle, they may select and re-execute the task. Activities can be interrupted by the the mechanisms specified for the class `Unit` (e.g., the unit starts to fight or needs to rest) but also if required materials are unavailable or locations become inaccessible. Once a `Unit` succeeds in executing all activities of a task, that task is to be removed from the all `Scheduler`'s lists of tasks.

A task always has a name (any string) and a priority (integer), and consists of a non-empty list of activities. These activities are to be executed, starting with the first element of the list. Activities can have arguments, e.g. a target position, associated with them and can be guarded by conditionals. A simple programming language consisting of a small number of statements and expressions is to be used to define tasks. Examples for tasks are:

```
name: "goto (10, 10, 10)"
priority: 5
activities: moveTo (10, 10, 10);

name: "operate workshop"
priority: -100
activities: w := workshop;
  moveTo boulder; work here;
  moveTo w; work here;
  moveTo log; work here;
  moveTo w; work here; work here;

name: "dig tunnel from (11, 10, 10) to (14, 10, 10)"
priority: 1000
```

[2]Some of the above functionality, such as tasks being a member of multiple schedulers, will not be supported or used by the GUI in this iteration. However, all functionality must be thoroughly covered (like all other public methods) by test cases.

```
activities: moveTo (10, 10, 10); work (11, 10, 10);
  moveTo (11, 10, 10); work (12, 10, 10);
  moveTo (12, 10, 10); work (13, 10, 10);
  moveTo (13, 10, 10); work (14, 10, 10);
```

In our second example we declare and assign a global `w` with some position, which we later use to perform actions. Global variables can be declared at any point in the program, however they must first be assigned before being used in an expression. Variables are not explicitly assigned a type (c.f. Sec. 1.4.3) but take the type of the first assignment. All further assignments of the same variable must be of the same type.

As can be seen, our language uses a number of "wildcard" expressions such as `workshop`, `boulder` or `log` which refer to any such object in the game world. The task "operate workshop" can be interrupted if either no workshop, no boulder or no log exists in the game `World`, or if the selected objects become inaccessible. Likewise, the task "dig tunnel" shall be interrupted if any of the target locations become inaccessible or if the work orders cannot be carried out (i.e., if the target cube is not solid and does not contain any items). Note that, in particular the third task, is not guaranteed to actually "dig tunnel from (11, 10, 10) to (14, 10, 10)": the `Unit` executing the task may just pick up and drop objects or operate workshops, depending on the current features of the game `World`. Using conditionals we can rewrite the task as:

```
name: "dig tunnel from (11, 10, 10) to (14, 10, 10)"
priority: 1000
activities: if (carries_item(this)) then work here; fi
  if (is_solid(11, 10, 10)) then
    moveTo (10, 10, 10); work (11, 10, 10); fi
  if (is_solid(12, 10, 10)) then
    moveTo (11, 10, 10); work (12, 10, 10); fi
  if (is_solid(13, 10, 10)) then
    moveTo (12, 10, 10); work (13, 10, 10); fi
  if (is_solid(14, 10, 10)) then
    moveTo (13, 10, 10); work (14, 10, 10); fi
```

1.4.1 Statements

More formally, the syntax of statements `s` in Extended Backus Normal Form (EBNF) notation is as follows:

```
s ::=
| x := e;
| while e do s done
| if e then s [ else s ] fi
| break;
| print e;
| ACTION;
| {s}

ACTION ::=
  moveTo e
```

```
| work e
| follow e
| attack e
```

That is, a statement is either an assignment, a while loop, an if-then-else, `break` (immediately terminates loops), a print statement (for debug output only), an action statement or a sequence of zero or more statements. There are four different kinds of action statements: `moveTo` (as specified in Sec. 1.2.3), `work` (c.f. Sec. 1.2.4), `follow` (to follow a target unit, continuously moving to the – possibly changing – position of the followed unit until that position – or a neighbouring position – is reached or the followed unit dies) and `attack` (c.f. Sec. 1.2.5). Several statement types require argument expressions to be evaluated, which are defined below. Students working alone must not support the break statement.

1.4.2 Expressions

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

```
e ::=
  x
| POSITION
| UNIT
| true
| false
| "(" e ")"
| ! e
| e || e
| e && e
| is_solid e
| is_passable e
| is_friend e
| is_enemy e
| is_alive e
| carries_item e

POSITION ::=
  here
| log
| boulder
| workshop
| (x, y, z)
| next_to e
| position_of e
| selected

UNIT ::=
  this
| friend
| enemy
| any
```

As can be seen, an expression `e` always evaluates to a cube position, a unit reference or to a boolean type. `e` can be a variable, a cube position, or a boolean constant. Parentheses can be used to express precedence, `!`, `||` and

`&&` denote negation, disjunction and conjunction, and the `is_*` expressions return true or false depending on the properties of the cube position or unit parameters. Positions can be assigned using the concrete (integer) coordinates of a cube, or by using the terms, `here`, `log`, `boulder` or `workshop`, which shall be implemented to return the position of the cube currently occupied by the `Unit` executing the activity, or a cube that contains a `Log`, `Boulder` or a workshop. The `next_to POSITION` expression shall evaluate to an accessible position that directly neighbours the target cube identified by `e`. The evaluation of `log`, `boulder` or `workshop`, `next_to` may be implemented to yield non-deterministic results. That is, two consecutive evaluations of `next_to here` may evaluate to different absolute positions, provided that there are multiple accessible position that directly neighbouring `here`.

The `position_of UNIT` shall return the position of a `Unit` reference, such that `position_of this` and `here` yield the same position. A special position is denoted by the `selected` keyword, which means that the task must be instantiated with concrete positions from the GUI once the task is loaded. Similarly, unit specifications can use `this`, `friend`, `enemy` or `any`, which shall evaluate to the `Unit` executing the activity, or a unit of the same faction, a unit of another faction, or any unit other than `this`, respectively. Similar to the `POSITION` wildcards, evaluation of `friend`, `enemy` or `any` may be non-deterministic. All the methods of the classes Scheduler and Task must be documented in a formal way. On the other hand, no documentation is required for the classes related to statements and expressions.

Students working in pairs shall implement the `log`, `boulder`, `workshop`, `friend`, `enemy` and `any` expressions as referring to the nearest accessible object of the specified type. Students working alone may choose an arbitrary object of the specified type. In addition, they must not work out the special position `selected`. Functional aspects of Java 8 shall be used to implement these expressions as you see fit.

1.4.3 Type Safety and Well-Formedness

Tasks – "programs" defined in the above syntax – employ three different types of expressions or variables: booleans, `POSITION` references and `UNIT` references. The above syntax does not narrowly define when each type may be used. Instead, your implementation of programs must make sure that only programs using valid types in their expressions can be constructed. For example, programs that would compute the conjunction or disjunction of `POSITION`s or `UNIT`s shall be rejected. Similarly the use of position arguments to `carries_item` or a boolean argument to `is_friend` makes little sense. Students who do the project on their own must not work out this part of the project.

If you want to score 17 or more on this project, you must not work out a method to check whether programs are well typed. Instead, you must make it impossible to construct programs that violate the above typing rules. You must use generic classes and/or interfaces for that purpose. More in particular, the Java compiler must complain if we try to build a program that has typing errors. 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 further not allowed to change the signature of the methods in interfaces that we provide.

In addition to Type Safety, `break` statements have a semantics that is only defined within the execution of a loop body. Therefore, `break` may only appear within while-bodies. Also, variables must be assigned once before they are being used in expressions.

You must write a method that checks whether a program satisfies these conditions, i.e., guarantee well-formedness of a program. That method must be worked out in a total way. Students who do the project on their own must not work out this part of the project.

1.4.4 Task Creation, Evaluation and Execution

Tasks are created by loading a text file containing exactly one task descriptions through the GUI. You are provided with the infrastructure to parse these text files (c.f. Sec. 1.4.5) but you must implement the classes to represent and execute a specific task.

Importantly, when creating tasks that make use of the `selected` keyword, these tasks have to be instantiated for each cube of the game `World` selected in the GUI. The user interface provides a list of selected cube coordinates as a parameter in the `createTasks` method of `ITaskFactory` (c.f. Sec. 1.4.5). For each instance of such a task, all occurrences of `selected` shall refer to the same in-game position. The `selected` keyword is particularly useful for tasks such as digging, which may affect a large number of cubes that become accessible once some other cubes are dug out:

```
name: "dig"
priority: 8
activities: if (carries_item(this)) then work here; fi
  if (is_solid(selected)) then
    moveTo (next_to(selected)); work (selected); fi
```

Once a task is assigned to a `Unit`, that `Unit` starts executing the statements included in the task in order of appearance and starting from the first statement. Specifically, one statement (or loop iteration) may be executed per 0.001 s of game-time. For time advancements $\Delta t < 0.001$, exactly one

statements shall be executed. A program can order its `Unit` to perform as many actions as permitted by Δt. Once Δt is exhausted, the program is put on hold. If execution of an action extends beyond Δt (e.g. movement, work or fighting), program execution is put on hold until the activity is complete. Yet, the program's state is preserved so that execution can continue in a following time advancement at the statement that follows upon the statement that was last executed. Once all statements have been executed and control reaches the end of the program, the program is terminated and the task is complete. Alternatively, if an action is interrupted or cannot be executed, the program is terminated and the task becomes available again, albeit with a reduced priority.

1.4.5 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. Technically, to parse a `String` object, instantiate the class `TaskParser` and call its `parse` method. This method constructs an in-memory representation of the program by calling methods in the `ITaskFactory` interface. You should provide a class that implements this interface. For your convenience, the methods `TaskParser.parseTasksFromString` and `TaskParser.parseTasksFromFile` are provided to facilitate parsing of input strings or tasks.

The parser was generated using the ANTLR parser generator based on the file `HillbilliesTaskLang.g`. 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:

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

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

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

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

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

3 Testing

Write JUnit test suites for the classes `World`, `Unit`, `Boulder`, `Log`, `Scheduler` and `TaskFactory` that tests each public method. Include these test suites in your submission.

4 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on `Unit`. The user interface is included in the assignment as a JAR file. When importing this JAR file you will find a folder `src-provided` that contains the source code of the user interface, the `Util` class 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, write a class `Facade` in package `hillbillies.part3.facade` that implements the provided interface `IFacade` from package `hillbillies.part3.facade`. `IFacade.java` contains additional instructions on how to implement the required methods. Read this documentation carefully.

To start the program, you may execute the `main` method in the class `hillbillies.part3.Part3`. After starting the program, you can press keys to modify the state of the program. Generally your commands will affect

`Units` of the first faction of the game world only. Units that belong to other factions act autonomously. Commands are issued by pressing `c` to create a new `Unit`, `Tab` to switch between existing `Units`, `Shift+Tab` to switch between factions, and `w`, `r`, `a` to make the currently selected unit work, rest or attack. Movement is controlled by pressing `y`, `u`, `i` to move North (NW, N, NE), `h` and `k` to move West or East, and `b`, `n`, `m` to move South (SW, S, SE). The modifiers `Ctrl` and `Alt` together with the above keys control movement along the z-axis; `Ctrl+j` and `Alt+j` result in moving straight up or down. `Esc` terminates the program.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Unit`. 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 20th of May 2016 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` to confirm the submission!