CS-107 : Mini-project 2
Games on grid :: ”ICRogue”

S. Abuzakuk, J. Sam, B. Jobstmann

Version 1.7

Table des matières

1 Presentation

This document uses colors and contains clickable links (underlined text). It is best viewed in digital format.

Over the past few weeks, you have familiarized yourself with the fundamentals of a small ad hoc game engine (see tutorial) allowing you to create two-dimensional grid games . The simple draft obtained is similar to what can be found in an RPG type game. The goal of this mini-project is to take advantage of it to create concrete versions of another type of game. The basic game that you will be asked to create is strongly inspired by famous Roguelike games. Figure 1 shows an example basic outline[1] which you can later add to as you wish, according to your fantasy and imagination.
In addition to its fun aspect, this mini-project will allow you to put into practice the fundamental concepts of object-orientation in a natural way. It will allow you to experience the fact that a design located at an adequate level of abstraction makes it possible to produce programs that are easily extensible and adaptable to different contexts.
You will have to concretely complexify, step by step, the desired functionalities as well as the interactions between components.

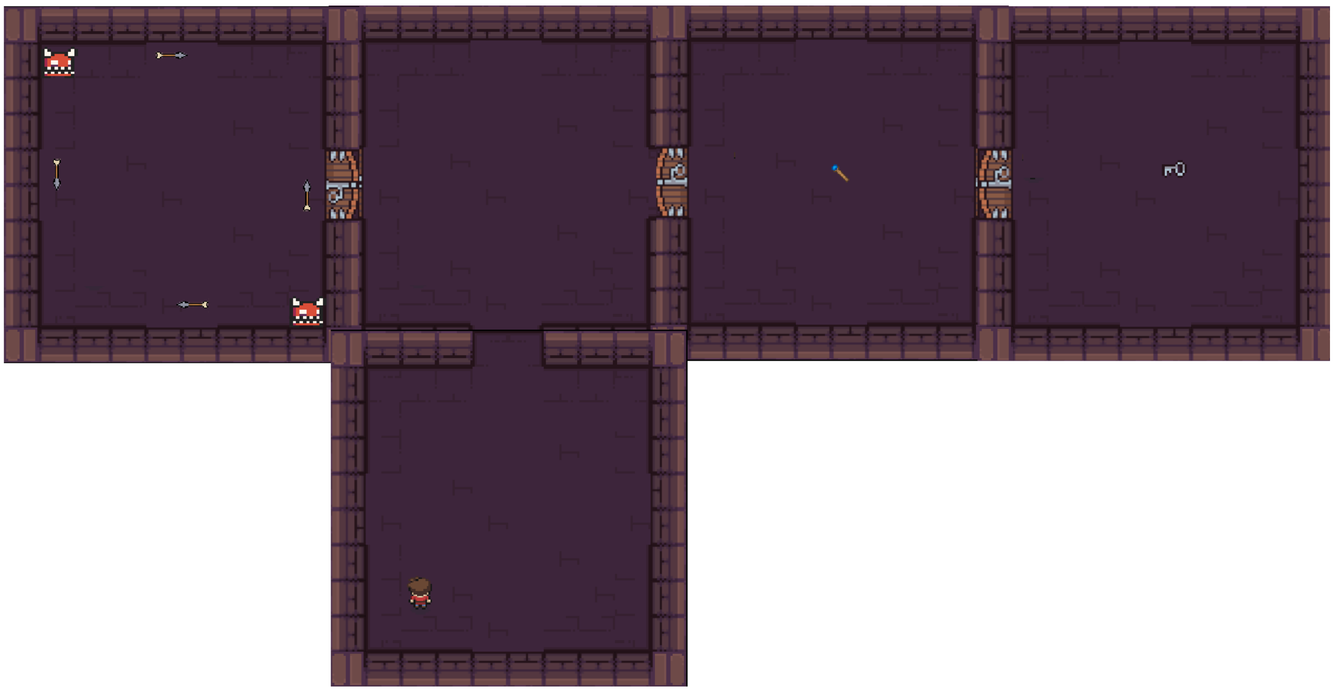


FIG. 1 : Example of a game level with five rooms. The "player" moves from room to room (and only the current room is visible). The goal is to meet the challenges associated with each room and to reach and solve the ultimate room (here the one at the top left). To meet the challenges and access the various rooms, the player will have to collect objects and defend themselves against enemies.

[1]see demonstration video

The project has four mandatory stages :

- Step 1 (“Basic ICRogue”) : At the end of this step you will have created, using the provided game engine tools, a basic game instance with an actor moving through a room and able to collect objects and throw projectiles ;
- Stage 2 (”Level Exploration”) : During this stage the player will become able to explore a *game level* consisting of several rooms ;
- Step 3 (”Challenges and Enemies”) : This will involve modeling *challenges* that must be met to move from one room to another. Along the way, you will have to collect useful objects to meet challenges or unlock doors. The goal is to reach the ultimate “Boss” room. Defeating the latter is what will condition the passage to a next level or winning.
- Step 4 (”Random Level Generation”) : One of the specificities of ”RogueLike” type games is the procedural generation of game levels. This step will be dedicated to this task.
- Step 5 (Extensions, optional) : During this step, you will be offered to implement various extensions and enrich the game created in the previous step or create another one in your own way.

Coding a few extensions (your choice) allows you to earn bonus points and/or promote your project to participate in the competition.

Here are the main guidelines/pointers to observe for project coding :

1. The project will be coded with standard Java tools (import starting with `java.` or `javax.`). If you have doubts about the use of such or such library, ask us the question and above all pay attention to the alternatives that your IDE offers you to import on your machine. The project notably uses the `Color` class. You must use the `java.awt.Color` version and not other implementations from various alternative packages.
2. Your methods will be documented according to standard javadoc (inspire yourself from the provided code). Your code should respect the usual naming conventions and be well **modularized and encapsulated**. In particular, intrusive publicly accessible getters on modifiable objects should be avoided.
3. The guidelines (this document) can sometimes be very detailed. **That does not mean that they are exhaustive**. The methods and attributes necessary to carry out the desired processing are obviously not all described and it will be up to you to introduce them according to what seems relevant to you and respecting good encapsulation.
4. Your project **must not be stored on a public repository** (like github). For those of you familiar with git, we recommend using GitLab : https ://-gitlab.epfl.ch/, but any type of deposit is acceptable as long as it is private.

sectionBasic ICRogue (step 1)
The goal of this step is to start creating your own little "ICRogue" game in a modest lineage of "RogueLike" type games.
This basic version will contain a main character able to walk around a room and collect objects. The collection of objects will be done according to the more general mechanism of *interactions between actors*, as described in tutorial 3.
This game will therefore feature :

- a main character ;
- collectables that the latter can pick up by passing over them (”contact interactions”) or by summoning them (”range interaction”) ;
- projectiles it can throw.

You will be working in the provided `game.icrogue` subpackage.

1.1 Preparing `ICroguegame`

Prepare an `ICrogue` inspired by `Tuto2` (in the provided sub-package `game.tutosSolution`). The latter will consist of :

- The class `ICRoguePlayer` which models the main character, to be placed in `game.icrogue.actor` ; leave this class empty for the moment, we will come back to it a little later.
- The `ICrogue` class, equivalent to `Tuto2` to be placed in the package `game.icrogue` ; this class will have an `ICRoguePlayer` as a character. Don't forget to adapt the `getTitle()` method which will return a name of your choice (e.g. *"ICrogue"*).
- The class `ICrogueRoom` equivalent to `Tuto2Area`, to be placed in the sub-package `game.icrogue.area`.
- The class `Level0Room` inheriting from `ICRogueRoom`, to be placed in a subpackage `game.icrogue.area.level0.rooms` (equivalent to class `Ferme` or `Village` of `game.tutosSolution.area.tuto2`).
- The class `ICrogueBehavior` analogous to `Tuto2Behavior` to place in `game.icrogue` and which will contain a public class `ICRogueCell` equivalent of `Tuto2Cell`.

However, several small changes will be necessary to adapt to the spirit of the new game. It is advised to be meticulous in the implementation of these adaptations (to start on the right foot ;-)).

1.1.1 Adaptation of `ICRogueRoom` and `Level0Room`

The rooms are actually going to be part of a "map" (which we will introduce when modeling the levels), as shown in figure2. An `ICRogueRoom` will therefore naturally have as an additional characteristic its coordinates in this map (of type `DiscreteCoordinates`). These coordinates (as well as the name of the associated grid) will be parameters of the constructor. A `Level0Room` therefore inherits from `ICrogueRoom` a position `[x][y]` on a map. In order to be able to identify a `Level0Room` more meaningfully in this context, you would modify its definition of `getTitle` so that it returns the string *"icrogue/level0"* to which we

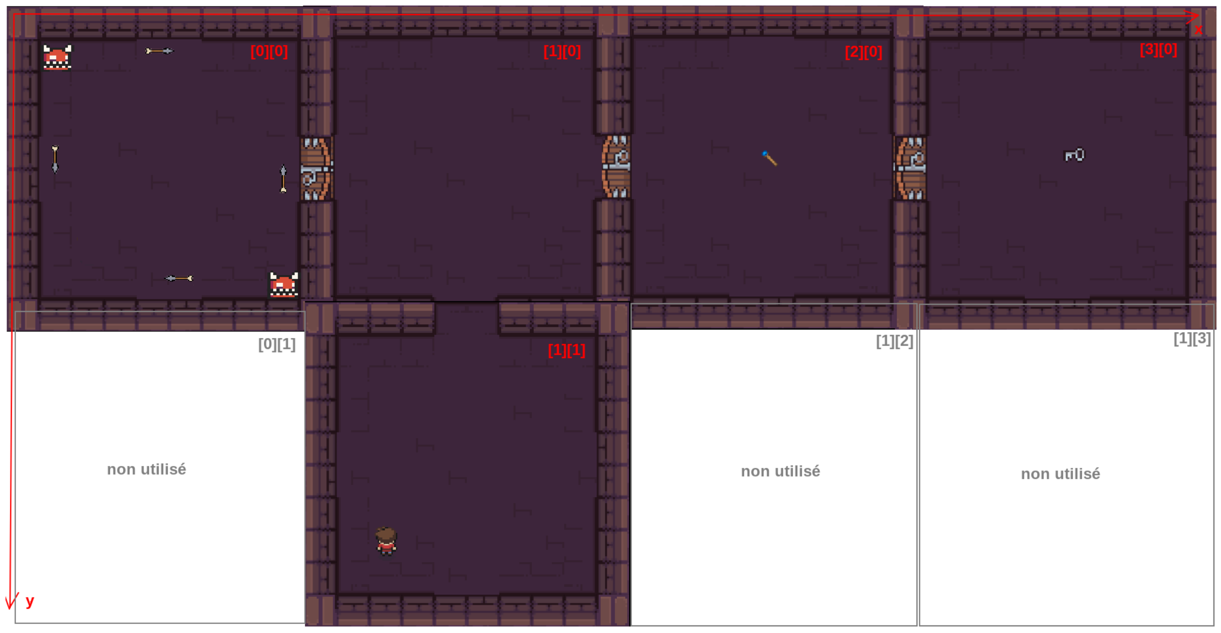


FIG. 2 : Example of "map" of rooms in a game level

concatenate the coordinates `x` and `y`. For example, if the position of the `Level0Room` on the map is `[0][1]`, then the title will be *`"icrogue/level001"`*. The title of the room will thus depend on its position. On the other hand, the grid associated with a Level0Room ! will always be the same, *whatever its position*. The name of the grid ("behavior") will be, for example, "icrogue/Level0Room" ! for *all* the `Level0Room` Contrary to what was the case in `Tuto2`, there is a distinction to be made here between the title of the area and the name of the associated gridthe title of the area is what identifies it in all the areas of a game, the `behaviorName` is what identifies the associated grid ; this grid is always the same for any area of a certain type. To manage this distinction, you will endow `ICRogueRoom` with an attribute allowing to store the name of the grid (`behaviorName`) and a constructor allowing to initialize the two specific attributes :

```
ICRogueRoom(String behaviorName, DiscreteCoordinates roomCoordinates)
```

`Level0Room` will be equipped with a constructor :

```
Level0Room(DiscreteCoordinates roomCoordinates)
```

which will call the constructor of its super class and give it *`"icrogue/Level0Room"`* as a value for the attribute `behaviorName`.

Attention : The method `begin` of `ICRogueRoom` at the time of creating the grid `ICRogueBehavior` will use as a parameter `behaviorName` and no longer the title of the area. Similarly, the actor `Background` of `Level0Room` will be constructed using the `behaviorName` (not the title) :

```
registerActor(new Background(this, behaviorName)); // better
  with a getter!
```

Note that there is no actor `Foreground` for the `Level0Room` Finally, don't forget to redefine `getCameraScaleFactor` in `ICrogueRoom` (with the value of 11 for example).

1.1.2 Adaptation de `ICRogue`

A game of type `ICrogue` is made up of levels and each level is called to contain several rooms.
Instead of having as attribute the titles of each of the possible rooms (as in `Tuto2`), the game `ICrogue` will have instead as attribute a *level* . For the moment, this concept is not introduced and we will simply assimilate a level to a single room. Instead of a level, there will be in `ICRogue` an attribute *current room* of type `Level0Room`.
The method `createAreas()` is to be replaced by a method `initLevel()`, in charge of creating the "level" and whose role is :

- to assign to the current room a `Level0Room` with coordinates (0,0) ;
- to add this room to the set of areas in the game ;
- to designate the current room as the current area of the game (`setCurrentArea`) ;
- to create a `ICRoguePlayer` (expect this call with a comment, we'll come back to it later) ;
- and to make it enter the current area (instruction also to be planned using a comment).

Note that it is no longer necessary to center the camera on the character because you want to have an overall view of the whole room (if you do nothing the camera is centered on the center of the current area).

1.1.3 Adaptation of `ICRogueBehavior`

In the spirit, `ICRogueBehavior` and `ICRogueCell` are equivalent to `Tuto2Behavior` and `Tuto2Cell`
The enumerated type describing the cell types and their "traversability" could be described as follows :

```
NONE(0,false), // Should never be used except in the toType method
GROUND(-16777216, true ), // traversable
WALL(-14112955, false), // not traversable HOLE(-65536, true);
```

However, the nature of the "scenery" will no longer be the only element that will condition the movement of the character. In the case of this new type of game, the presence of another actor who would not allow himself to be "stepped on" will also hinder the movement of the character. An object can be walked on (is traversable) if its method `takeCellSpace()` returns `false`.
Concretely, two entities for which the method `takeCellSpace()` returns true cannot coexist both in the same cell. The `canEnter` of `ICRogueCell` must also guarantee this. Make the adaptations suggested above in the `ICRogueBehavior` class.

1.1.4 Task

You are asked to code the concepts described above according to the given specifications and constraints. Start your game `ICRogue` You will verify that the empty room in figure 3 is displayed.

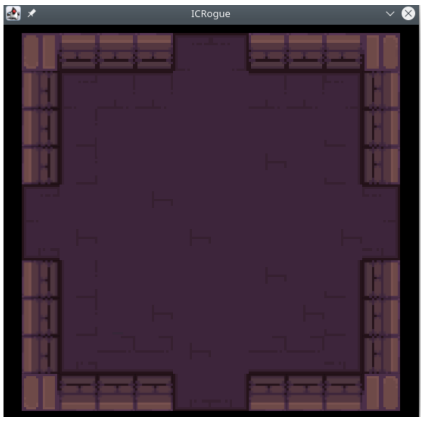


Fig. 3 : First room in the ICRogue game

1.2 Actors in "ICrogue"

Now that the basics are in place, you are asked to model some key players in `ICRogue` type games. They will be encoded in the `game.icrogue.actor` subpackage.
You will consider that all the actors involved in an `ICRogue` game, the `ICRogueActor`, are able to move on a grid (`MoveableAreaEntity`) and that at this level of abstraction they do not evolve in a specific way. Their area of belonging, their orientation and their starting position are given to the constructor. The cells they occupy are defined as for the `GhostPlayer` (method `getCurrentCells()`), but by default they are traversable (`takeCellSpace` returning `false`).
In terms of functionality, any `ICRogueActor` is able to enter a given area at a given position and leave the area it occupies. As `Interactable`, by default it can be subject to contact-only interactions (at this level of abstraction).
At this point in the project, two more specific categories of `RogueLikeActor` are to be introduced : the *main character* and *projectiles*. We will limit ourselves for the moment to the main character.

1.2.1 The main character

The class `ICRoguePlayer`, mentioned above, will be coded for the moment in the same spirit as `GhostPlayer`. You need to revise the inheritance relations though as it is obviously also an `ICRogueActor`. The image that will be used to draw it will depend on its orientation. Here is the code to extract the `Sprite` needed depending on orientation :

```
//bottom
new Sprite("zelda/player", .75f, 1.5f, this, new RegionOfInterest(0,
   0, 16, 32), new Vector( .15f, -.15f));
// right
new Sprite("zelda/player", .75f, 1.5f, this, new RegionOfInterest(0,
   32, 16, 32), new Vector(.15f, -.15f));
// top
new Sprite("zelda/player", .75f, 1.5f, this, new RegionOfInterest(0,
   64, 16, 32), new Vector(.15f, -.15f));
// left
```

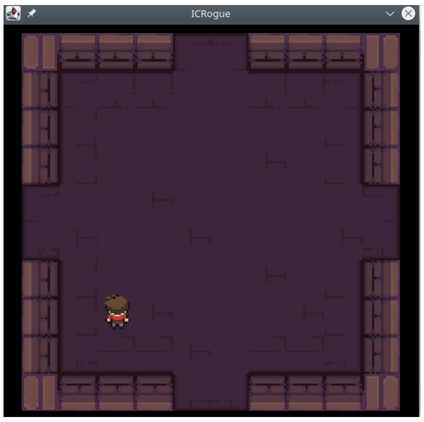


Fig. 4 : Placement of the character

```
new Sprite("zelda/player", .75f, 1.5f, this, new RegionOfInterest(0,
   96, 16, 32), new Vector(.15f, -.15f));
```

(see section 6.6 of the tutorial if you want to understand the details of this code).
The `ICRoguePlayer` behaves (updates) like `ICRogueActor` but in addition :

- it will be able to move using the directional arrows like the `GhostPlayer` coded in the solution of the tutorial ;
- pressing the `X` key makes him throw *fireballs* : a fireball, having the same orientation as him, must be created at the position he occupies each time the `X` is pressed. For the moment, just provide this treatment as a comment.

Finally, an `ICRoguePlayer` will not be traversable.
To be able to implement the fireball, you will be asked to introduce more generally the notion of *projectile*, which will keep us busy in the next section. Once `ICRoguePlayer` is coded, you can finalize the `begin` method of `ICRogue` and start testing your game. At launch, a player of type `ICRoguePlayer` will be created at the position intended for it in the starting area (take $(2, 2)$ in `Level0Room` for example). It will be oriented to `UP`. To facilitate the tests, you will endow `ICRogue` with the following control : the `R` key must make it possible to make a "reset" of the game i.e. to restart it under the same conditions as the very first time that one launches it (think of the modularization of the treatments here).

1.2.2 Task

You are asked to code the concepts described above according to the given specifications and constraints. Launch your game "ICRogue". You will check :

1. that the game starts by displaying the `ICRoguePlayer` according to the figure 4 ;
2. that this actor can be moved freely on the whole area by means of the directional arrows but that it must not be able to leave it ;
3. that its graphic representation fits well with its orientation ;
4. that he cannot walk on the walls of the edge ;
5. and that the key `R` allows to reset the game according to the specification described above.

1.2.3 Projectiles

The projectiles will be modeled in a subpackage `game.icrogueactor.projectiles`. Projectiles (abstract class `Projectile`) burn out under certain conditions (for example the fireball eventually goes out). They are `ICRogueActors` that can be drawn using a `Sprite`. They moving and are characterized by :

- The number of "frames" used for the calculation of the displacement ;
- the damage points they can inflict (an integer) ;
- a state indicating whether they have been consumed or not (`isConsumed`, a boolean).

To make them move, simply invoke the `move` method (in the update method), passing the number of frames as a parameter.

The area to which a projectile belongs, its orientation, its starting position in the area, the number of damage points it inflicts and the number of "frames" used for its movement are given during construction. It must also be possible not to specify the last two values explicitly at construction. In this case, they will default to the values `DEFAULT_DAMAGE` (equal to 1 for example) and `DEFAULT_MOVE_DURATION` (equal to 10 for example) respectively.

The associated `Sprite` cannot have any concrete value at this level of abstraction. It can also potentially take several values for the same projectile (for example a ball of fire could become more and more red as it moves). Therefore, a "setter" is more suitable for initializing its value and the concept of projectile is therefore abstract.

The cells occupied by a projectile are defined as for the `ICRoguePlayer` (method `getCurrentCells()`), and by default they are traversable (`takeCellSpace` returns `false`).

Functionally, a projectile behaves like an object that burns out. The provided interface `Consumable` (package `game.icrogue.actor.projectiles`) allows to model this.

A `Consumable` object will typically offer the `void consume()` describing what happens when the object is consumed and a method `boolean isConsumed()` to test if the object is consumed. The `void consume()` method of a projectile will simply consist at this stage of assigning the value `true` to the `isConsumed` attribute.

All kinds of concrete projectiles can be considered. You are asked for this part to code a "fireball" projectile (`Fire`) whose characteristics are as follows :

- number of damage points : 1
- number of " frames" for movement : 5

The `Sprite` associated can be extracted from the provided resources by means of

```
new Sprite("zelda/fire", 1f, 1f, this, new
   RegionOfInterest(0, 0, 16, 16), new Vector(0, 0) ));
```

A fireball disappears when it is consumed (no longer exists as an actor). Once the fireball is coded, you can complete the behavior of ICRoguePlayer ! so that the 'X' key creates a fireball at the character's position
(`getCurrentMainCellCoordinates`) and withthe same orientation.

1.2.4 Task

You are asked to code the concepts described above according to the given specifications and constraints.

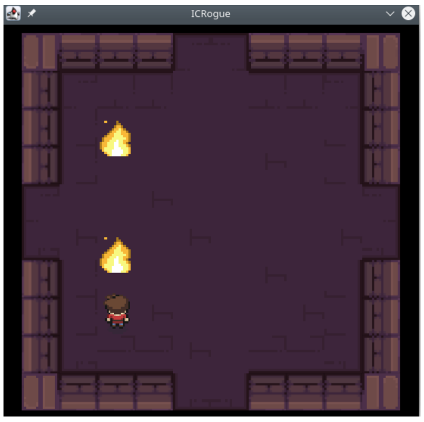


FIG. 5 : Launching fireballs with the `X` key.

Launch your `ICRogue` game. You will verify :

1. that the `X` allows the main character to throw fireballs in front of him ;

At this point the fireballs do not disappear yet (they will when you have coded the interactions, see below).

1.3 Collectible objects

It is now a question of modeling objects that the main character will be able to collect, which will impact the course of the game. Collectible items will be implemented in the `game.icrogue.actor.items` subpackage.

You are asked to introduce an abstract class `Item` modeling "collectable" objects and inheriting from class `CollectableAreaEntity` provided in the model (package `areagame.actor`).

An `Item` is a traversable object by default. It is drawn using a `Sprite` which characterizes it. However, it only draws if it has not yet been collected (method `isCollected()` of `CollectableAreaEntity`).

As `Interactable`, by default it can be subject to contact-only interactions[2].

Two concrete types of `Item` are to be coded at this stage : cherries (`Cherry`) and staves (`Staff`).

A staff is an `Item` which accepts remote interactions (the character can summon it if it is close enough to it). Its area to which it belongs, its orientation and its starting position are given as construction parameters. The associated `Sprite` is constructed by means of :

```
new Sprite("zelda/staff_water.icon", .5f, .5f, this));
```

A "cherry" is coded in the same spirit, but does not accept remote interactions. The associated `Sprite` is constructed with :

```
new Sprite("roguelike/cherry", 0.6f, 0.6f, this))
```

Finally complete the code of `Level0Room` so that it registers on creation :

- a staff in position `(4,3)` downward facing ;
- a cherry in position `(6,3)` oriented downwards.

[2]remember the `isCellInteractable` and `isViewInteractable`

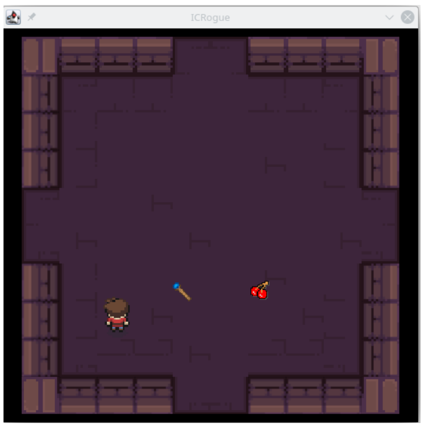


FIG. 6 : Initial position of the "player" and the actors `Cherry` and `Staff`.

1.3.1 Task

You are asked to code the concepts described above according to the given specifications and constraints. Launch your game `ICRogue` You will verify that the stick and the cherry appear (figure 6).

1.4 Interactions

You are now asked to apply the diagram suggested by the tutorial 3 [3] to set up the first interactions between the `ICRogueActor` coded so far.

1.4.1 Item Collection

The first step is to allow the `ICRoguePlayer` to collect items. To do this, start by implementing the fact that all `ICRoguePlayer` are `Interactor` (they can subject `Interactable` to interactions, for example to pick them up).
As an `Interactor`, `ICroguePlayer` must define the methods :

- `getCurrentCells` : its current cells (in practice it's a set containing only its main cell, as you have already had occasion to express) ;

- `getFieldOfViewCells()` : the cells of its fields of view consist of the single cell it faces

```
Collections.singletonList(
    getCurrentMainCellCoordinates().jump(
        getOrientation().toVector()));
```

As `Interactor`, it will always want all contact interactions (`wantsCellInteraction` always returns true). The fact that it is requesting range interactions (`wantsViewInteraction`) will be provided by the player : the key `W` will be used to indicate that the character wants ranged interaction. For example, if our character is in front of a staff, to indicate that we want him to pick it up, we press the `W` key. This key will obviously not deal only with the specific case of the staff. It will only be used to switch the character to "request remote

[3] A supplementary video is also available to explain implementing interactions : https://proginsc.epfl.ch/wwwhiver/mini-projet2/mp2-interactions.mp4

interaction" mode (and `wantsViewInteraction` will return `true` or `false` depending on the value of this mode).
Let's now look at the concrete management of interactions. In the `game.icrogue.handler` subpackage, create the `ICrogueInteractionHandler` interface inheriting from `AreaInteractionVisitor` and which provides a default definition of the interaction methods of any `Interactor` from `ICrogue` with :

- a game cell (`ICRogueCell`) ;
- the main game character (`ICRoguePlayer`) ;
- a cherry ;
- a staff ;
- and a fireball.

These definitions (by default) will have an empty body to express the fact that by default, the interaction consists of doing nothing. `ICRoguePlayer` as `Interactor` of the `ICRogue` game, should provide a more specific definition of these methods if needed.
All specific `Interactable` must now indicate that it agrees to have its interactions handled by an interaction handler of type `ICRogueInteractionHandler`. Their method `acceptInteraction` (empty so far) should be reworked as follows (in each of the relevant classes) :

```
void acceptInteraction(AreaInteractionVisitor v, boolean isCellInteraction) {
  ((ICRogueInteractionHandler) v).interactWith(this, isCellInteraction);
  }
```

This method may contain more code depending on the case (for example, it is reasonable to consider that consumed projectiles no longer accept interactions).
So that `ICroguePlayer` can manage more specifically the interactions that interest him, define in the class `ICRoguePlayer`, a private nested class
`ICRoguePlayerInteractionHandler` implementing `ICRogueInteractionHandler`. Add the necessary definitions to deal more specifically with the interaction with a cherry and with a staff.
These two items will need to be collected by interaction :

- the cherry by a contact interaction ;
- the staff by ranged interaction (remember that the character can express the wish for ranged interaction by means of 'W').

Note : The code of these methods should be no longer than two to three lines.
According to tutorial 3, for this to work, you need that :

- `ICRoguePlayer` has as its attribute its specific interaction handler (of type `ICRoguePlayerInteractionHandler`) ;
- its method `void interactWith(Interactable other, boolean isCellInteraction)` delegates the handling of this interaction to its handler (`handler` below) :

  ```
  other.acceptInteraction(handler, isCellInteraction);
  ```

  Remember that the method `void interactWith` is automatically invoked by the game core for any `void Interactor` for all `void Interactable` with which he is in contact or who is in his field of vision (revise the section 6.3.1 of the tutorial if necessary).

1.4.2 Projectiles as Interactor

You now understand the general mechanics of how one actor interacts with another. Actors other than the main character can of course be "interaction seekers". We now have to model the fact that projectiles are objects that can subject other actors to interactions, ie they behave like `Interactors`.
For this, `Projectile` overrides methods `getCurrentCells` and `getFieldOfViewCells()` like `ICRoguePlayer` does.
As `Interactor`, a projectile will always want all contact and ranged interactions (`wantsCellInteraction` and `wantsViewInteraction` always return true).
The fireball as a concrete projectile will have an interaction handler that will implement the following specific interaction : if the ball interacts (range interaction) with a cell of type `WALL` or `HOLE` it is consumed.
Finally, change the behavior of `ICroguePlayer` so that he can't throw fireballs using the `X` unless they have collected a staff beforehand.
At this stage you will model this situation in a very simple way : for example, a boolean attribute of the character indicating whether or not he has picked up a staff will do the trick. This can of course be sophisticated as desired later.

1.4.3 Task

You are asked to code the concepts described above in accordance with the given specifications and constraints.
Launch your `ICRogue` game. You will verify :

1. than the `RogueLikePlayer` behaves as for the previous step, except that he can pick up a cherry by walking on it and the staff if it is in his field of vision and the `W` button was pressed ;
2. that it cannot step on the staff ;
3. that picked up items disappear (visually) ;
4. that the character cannot launch fireballs with the `X key` unless it he has picked up the staff beforehand ;
5. Thrown fireballs disappear when they hit walls or holes.

Question 1

A priori, the logic put in place, as exposed in the tutorials and concretely used in this first part of the project, may seem unnecessarily complex. The advantage it offers is that it models in a very general and abstract way the needs inherent in many games where actors move on a grid and interact either with each other or with the content of the grid. How could you take advantage of this to implement a Pacman game for example ? What would it suffice to define ?

In the rest of the project, you will have to code many other interactions between actors or with the cells. All future interactions must imperatively be coded according to the scheme set up during this part and must not require type tests on the objects.

1.5 Step 1 validation

To validate this step, all checks of sections 1.1.4, 1.2.2, 1.2.4, 1.3.1 and 1.4.3 must have been completed.

The game `ICrogue` whose behavior is described above is to be returned at the end of the project.

This part is deliberately guided because it is the heart of how the game works. However, all the details are obviously not given.

2 Exploration of a level (step 2)

In roguelike games, the main character must explore multiple rooms. The rest of the project will involve modeling the fact that :

- the game is made up of *levels* and that each level is made up of a set of *rooms* ;
- passage from one room to another is done by means of *connectors* (waypoints) ;
- that each room has a challenge associated with it, like removing an enemy. When this challenge is solved, all connectors open, except those that require the possession of a key (those that are locked). The character must therefore have previously picked up such a key (possibly in another room) ;
- there is a particular room (“boss” room) for which you must have found the key to enter it. The goal is to defeat this enemy, having sufficiently equipped yourself beforehand in the other rooms to be able to do so. When the ”boss” is defeated, the level is passed and it is possible to transit to the next level.

During this step, you have to start modeling the notion of *connector*, *room with a connector* and *level* of the game. The challenges and enemies will not be modeled for the moment.

2.1 Waypoints between rooms

We will call the crossing points from one room to another “connectors”. They can be closed, open, locked or invisible. It is natural to model them as actors (of type `AreaEntity`), because they are not just scenery elements : as connectors, they can have *behaviors*, such as going from the open state to the closed state depending on complex conditions.
A connector is locked when its opening is conditioned by the possession of a key. It is therefore necessary to begin by slightly completing the hierarchy of `Item`.

2.1.1 Keys

Keys (class `Key`) are exactly like `Cherry`. The only difference is that a `Key` is characterized by an integer identifier (initialized by a value passed as a constructor parameter). The `ICRoguePlayer` picks them up by stepping on them. He must remember the identifiers of the keys collected because this is what will allow him in certain cases to open passage points.

2.1.2 Connectors

You are therefore asked to introduce the concept of `Connector` (in the `icrogue.actor` subpackage).
A `Connector` is characterized (at least) by :

- a state (an enumerated type with the values `OPEN`, `CLOSED`, `LOCKED` and `INVISIBLE` is a good option here) ;
- the name of the destination area (a `String` ; remember that each area has a name returned by its `getTitle()` method) ;
- the arrival coordinates in the destination area (these are the coordinates of the cell where we arrive, of type `DiscreteCoordinates`) ;
- and the identifier of the key that allows it to be opened (an integer). By default there is no need for a key (value of the identifier : a constant `NO_KEY_ID` of integer type for example).

The area to which a connector belongs, its position and its orientation will be given as parameters during construction and it is invisible by default when it is created.
The drawing of the connector depends on its state : we will draw a specific `Sprite` if it is invisible, locked or closed and nothing otherwise. The code for extracting sprites based on state and initializing them in the constructor is given to you for simplicity :

```
// for invisible:
new Sprite("icrogue/invisibleDoor_"+orientation.ordinal(),
   (orientation.ordinal()+1)
// for closed:
new Sprite("icrogue/door_"+orientation.ordinal(), (
   orientation.ordinal()+1)
// for locked
new Sprite("icrogue/lockedDoor_"+orientation.ordinal(),
   (orientation.ordinal()+1)
```

As `Interactable`, a `Connector` is defined as a traversable object if it is in the open state and it accepts remote interactions. The body of its method `getCurrentCells` is given to you for simplification :

```
DiscreteCoordinates coord = getCurrentMainCellCoordinates();
return List.of(coord, coord.jump(new
   Vector((getOrientation().ordinal()+1)%2,
   getOrientation().ordinal()%2)));
```

which implements the fact that the connector occupies its main cell and those immediately to its left and right.

2.2 Rooms with connectors

The `ICRogueRoom` class, sketched in the previous step, represents the abstract concept of a “room in an ICRogue game”. You are now asked to flesh out this concept a bit and model the fact that it is also characterized by a *set of connectors* (a dynamic array).
It is considered that the list of connector coordinates and the list of their respective orientations as well as the coordinates of the room in the map are also given as parameters to the constructor of an `ICRogueRoom`.

```
  ICRogueRoom(List<DiscreteCoordinates> connectorsCoordinates,
              List<Orientation> orientations,
              String behaviorName, DiscreteCoordinates roomCoordinates)
```

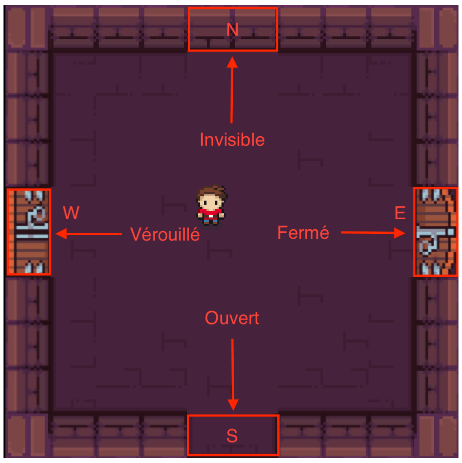


FIG. 7 : The `Level0Rooms` all have 4 connectors placed in the same places : East (E), West (W), North (N) and South (S)

This constructor will obviously have to use this data to initialize its set of connectors.
In order to facilitate further implementations, the constructor of `ICRogueRoom` will store the connectors in its connector set in the order indicated by its parameters. For example, constructing an ICRogueRoom! by passing it `{(4,5),(2,3)}` for the connector positions and `{LEFT, DOWN}` for their orientation will store the connector in position `(4,5)` is left-facing (`LEFT`) as the first connector in its set. **The method `createArea` of `ICRogueRoom` will of course have to make sure that the connectors are registered as actors so that their behavior can be simulated** .

2.2.1 Specific connectors

A priori there can be rooms with arbitrary sets of connectors located at arbitrary places. To simplify implementation and testing, we will start from the idea that the `Level0Room` are rooms that all have 4 connectors, placed in specific locations (see figure7).
Introduce and properly complete the code of an enumerated type `Level0Connectors` in `Level0Room` and whose definition start is :

```
public enum Level0Connectors implements ConnectorInRoom {
    // order of attributes: position, destination, orientation
    W(new DiscreteCoordinates(0, 4), new DiscreteCoordinates(8, 5) ,
       Orientation.LEFT),
    S(new DiscreteCoordinates(4, 0), new DiscreteCoordinates(5, 8),
       Orientation.DOWN),
    E(new DiscreteCoordinates(9, 4), new DiscreteCoordinates(1, 5),
       Orientation.RIGHT ),
    N(new DiscreteCoordinates(4, 9), new DiscreteCoordinates(5, 1),
       Orientation.UP);
    ...
}
```

This type models the characteristics of the `Connector` of a `Level0Room` namely : the fact that there are 4 connectors, that the connector at position zero in the set of connectors is the one identified by W etc. The idea is to use this enumerated type to create the connectors

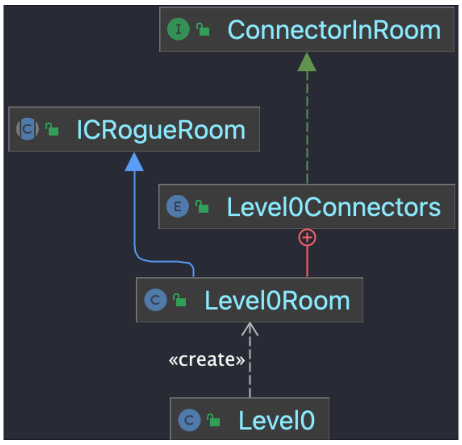


FIG. 8 : `Level0Room` inherits from `ICRogueRoom` It contains an enumerated type `Level0Connectors` which describes the characteristics of its connectors. It is possible to apply to each value of the enumerated type the methods `getIndex` and `getDestination` The existence of these methods is dictated by the interface `ConnectorInRoom` `getIndex` identifies a connector by its position in the set of connectors in the room. `Level0` is a level (coded in section refsec :level0). It is this level which will create the `Level0Rooms`.

in a given order and with given characteristics. The provided interface `ConnectorInRoom` allows to manipulate the values of the enumeration through a more abstract type which imposes them to provide :

- a method `DiscreteCoordinates getDestination()` which returns the destination coordinates of the connector corresponding to the enumerated type value ;
- and a method `int getIndex()` giving the index of this connector in the set of connectors of the room. This index simply corresponds to the position in the enumerated type (remember `ordinal()`).

Then define in the enumeration `Level0Connectors` the methods

```
static List<Orientation> getAllConnectorsOrientation()
static List<DiscreteCoordinates> getAllConnectorsPosition()
```

respectively returning the list of orientations and positions of connectors **in their order of definition in the enumerated type**.
These two methods should be used in the constructor of `Level0Room` so that it properly invokes its superclass's constructor.
Thus, for this specific type of room, if the method `getIndex()` imposed by interface `ConnectorInRoom` returns the position of the connector in the enumerated type, this will also match the position of the corresponding object in the list of connectors in the room.

Indication : the enumerated type of `Level0Connector` gives the orientation towards which the connector leads. At the time of the creation of the connector by the constructor of `ICRogueRoom`, it will be necessary to associate with the connector the opposite of this orientation (method `opposite()`. Thus, if the connector `W` has an associated orientation (`Orientation.Left`) it is because it leads the player to the left but its `Sprite` must be displayed as going to the right (because of the top view of the game).

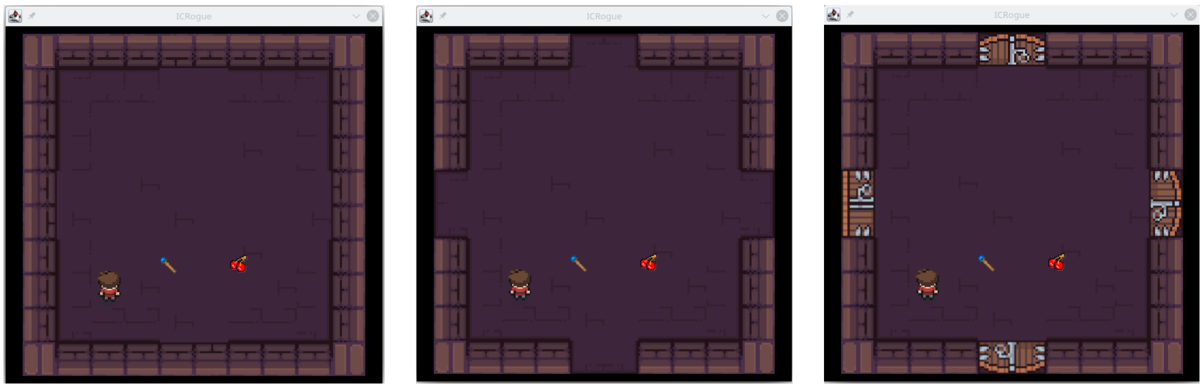


FIG. 9 : From left to right : invisible connectors, open, closed/locked

Adapting `ICRogueRoom` : We are going to test this part in a somewhat ad hoc way by first making some changes to the `ICRogueRoom` class. Indeed, as we have not yet modeled the notion of challenge and level, the logic that governs the opening and closing of connectors cannot yet be implemented. We will therefore "cheat" a little. Add the following controls to the `update` method of `ICRogueRoom` :

- if you press the `O` key, all the connectors go to the open state ;
- if the key `L` is pressed, the connector at the zero position locks with a key identified by 1 ;
- and if you press the `T` key, all the connectors switch states (from open to closed and vice versa, the locked connectors must remain locked).

2.2.2 Task

You are asked to code the concepts described above according to the given specifications and constraints. You will then verify (see figure reffig :connectors) :

1. that the room starts by being completely closed (connectors in the invisible state) ;
2. that the `O` key opens them and the `T` key closes them all ;
3. that the key `L` is used to lock the west connector (W), it then appears with a different visual of the closed connectors) ;
4. and that pressing the `T` button again will open/close everything except the locked connector.

2.3 Levels

A level, materialized by a class `Level` (to be encoded in the `icrogue.area` subpackage), is characterized by :

- a "map" of rooms (two-dimensional array `width x height` of `ICRogueRoom`) ;
- the arrival coordinates in the rooms (it is assumed for simplicity that it is the same for all the rooms of a level, `DiscreteCoordinates`) ;

- the map position of the "boss" room[4];
- and the name of the level's starting room.

Among the useful features associated with a level, you are asked to code the methods described below. In order to preserve the encapsulation you will make sure to define these methods as protected. You will add all the necessary methods, always taking care to preserve the encapsulation.

- `void setRoom(DiscreteCoordinates coords, ICRogueRoom room)` to assign the `room` at position `[coords.x][coords.y]` from the menu;
- `void setRoomConnectorDestination(DiscreteCoordinates coords, String destination, ConnectorInRoom connector)` : allowing to assign a destination to the connector at position `connector.getIndex()` in the room `[coords.x][coords.y]` of the map (namely, allowing for example to ensure that the 3rd connector of the room `[coords.x][coords.y]` leads to the room *`"icrogue/level000"`*); it is necessary to update both the name of the destination room and the coordinates of arrival in the latter;
- `void setRoomConnector(DiscreteCoordinates coords, String destination, ConnectorInRoom connector)` doing the same as the previous one but additionally marking the connector as closed;
- `void lockRoomConnector(DiscreteCoordinates coords, ConnectorInRoom connector, int keyId)` allowing to assign a key to the connector at position `connector.getIndex()` in room `[coords.x][coords.y]` of the map; this method will also mark this connector as locked;
- and a "setter" which allows to give a value to the name of the starting room from discrete coordinates. For example, if the setter parameter is `[0][0]`, it will assign the name of the starting room to the title of the room at position `[0][0]` on the map).

The constructor of a `Level` will take as a parameter the starting position in the rooms as well as the dimensions of the map. It will cause the position of the "boss" room on the map to default to `[0][0]`. Finally, it will call a method `generateFixedMap`, which we don't know how to define at this level of abstraction but which will have the role of filling the map with rooms.

Question 2

What do you think is the benefit of using the `ConnectorInRoom`? What design alternatives could you consider to identify a connector in a room.

The class `Level` is meant to represent an abstract game level. In what follows, we will create a concrete type of level `Level0`. To make things a little more interesting, we will first diversify a bit the rooms that can constitute such a type of level.

[4]Remember that the ultimate goal of "RogueLike" type games is to reach such a room by having sufficiently equipped yourself beforehand!

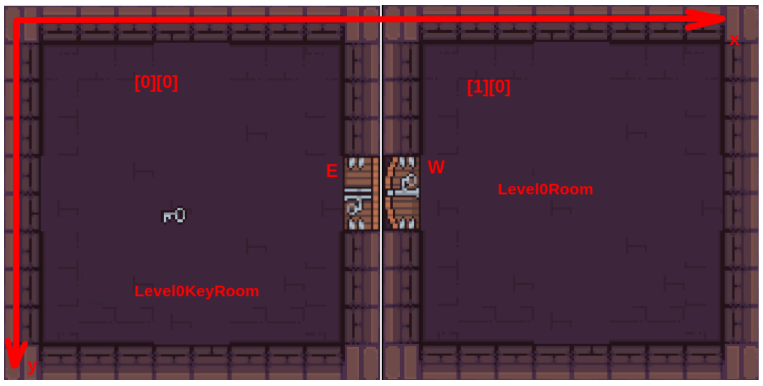


FIG. 10 : Both rooms are `Level0Room`. The room `[0][0]` is associated with a key. This key is used to unlock its "East" (E) connector.

2.4 Room Hierarchy

As you will have understood, the class `Level0Room` actually models a basic room of the future level `Level0`. You are now asked to complicate and specialize this concept a little by introducing the classes :

- `Level0ItemRoom` : which are an abstract `Level0Room` [5] containing a list of `Item`, it should be possible to add a `Item` given to the whole ;

- `Level0KeyRoom` and `Level0StaffRoom` : which are rooms with a unique `Item` : a key (`Key`) for the first and a staff (`Staff`) for the second. The position of this single `Item` will be the same for all created instances (e.g. `[5][5]`). The constructors will take as parameter the coordinates of the room on the map (and the key identifier for `Level0KeyRoom`). They will be in charge of creating their unique `Item`.

2.5 First concrete level : Level0

You are now asked to program a first concrete game level named `Level0` (to be placed in a `icrogue.area.level0` subpackage). This will of course also be a `Level`. To be instantiable, `Level0` must concretely define the method `generateMap`.
In order to facilitate your tests, you are asked to create two small maps using two utility methods `generateMap1` and `generateMap2`. The `generateFixedMap` of `Level0` can invoke one or the other of your choice (you can invoke both methods there and comment).
The `generateMap1` method will need to create the map in figure 10 and `generateMap2` that in figure 11. You will find using this link an example of how to code these methods (to be adapted to your code for the keys typically). It is assumed that the starting room always has the same coordinates for levels of type `Level0` (a common constant of `[1][0]` for example). The arrival position of the character in the rooms of the level will also have a default value (for example `[2][2]`). The class`Level0` will have a default constructor using this data and creating a 4x2[6] map.

[5]the point of making it an abstract class will become clearer when we link the rooms to the challenges
[6]these construction methods will be generalized in later stages

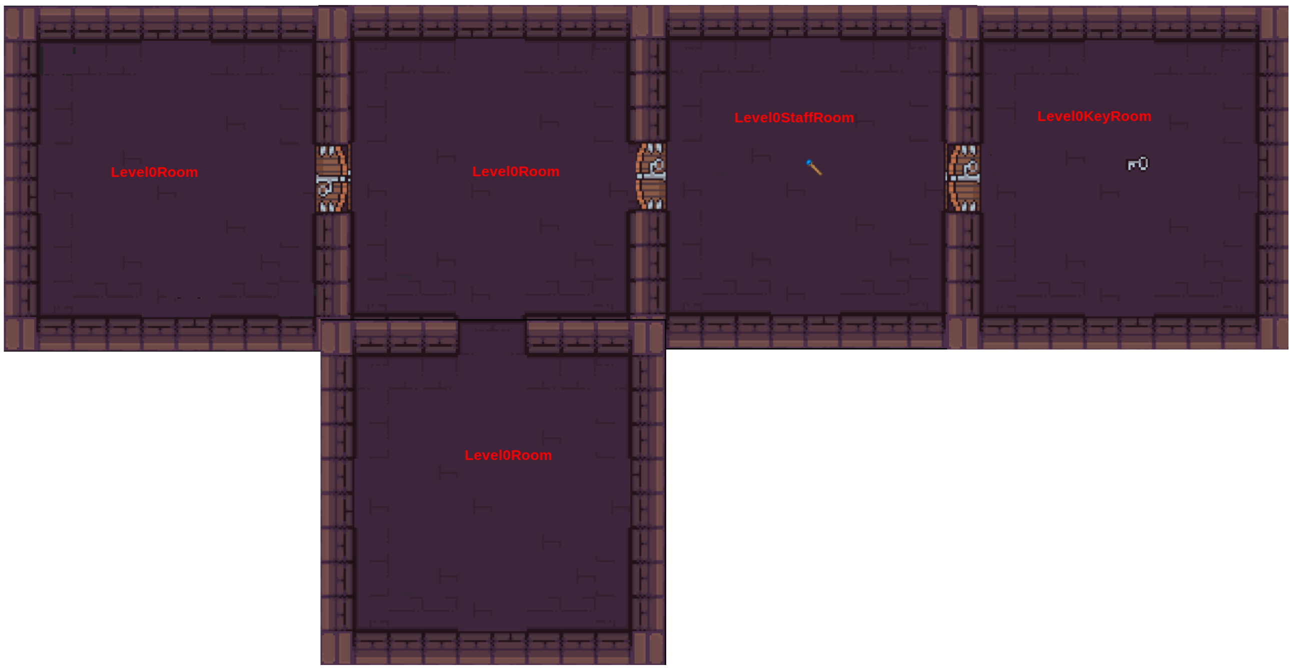


Fig. 11 : The room `[0][0]` is locked by the room key `[3][0]`.

2.6 Adaptation of existing code

Change to `ICrogue` : In the previous step, a game was characterized by a single room. It is of course now appropriate to replace this room with a *level* (`Level`). The method `initLevel()` will thus have to create a level (of type `Level0`), add all the rooms of the level as a game area (be careful not to introduce intrusive "getters"), designate the starting room of the level as the current game area and make the `ICRoguePlayer` enter this area. Moreover, a game of type `ICRogue` must be such that if you go to a room you have already visited, you must find it again in the state in which you left it (see the parameters of `setCurrentArea`). **You will make sure to adapt the code at this level by replacing the old content in an appropriate way.**

Change to `ICroguePlayer` : `ICRoguePlayer` must finally become able to pass through the connectors to pass from one room to another and this passage is potentially conditioned by the possession of keys.
It is therefore necessary to complete the interaction diagram between a `ICRoguePlayer` and `Connector` so that :

- if the player is interacting remotely, it tries to unlock the connector (the connector changes from locked to open if the player is in possession of the associated key) ;
- if it is in touch/contact interaction and not moving (`!isDisplacementOccurs()`), it can transit to the destination of the connectors.

Note : For the passage to the destination room, introduce a boolean indicating if the character is passing a connector, which will become true at the moment of the interaction and whose value can be queried by the game. It is indeed up to the game itself to manage the transitions from one room to another (as is the case in `Tuto2` with the invocation of `switchArea`). It is therefore suggested that you code a method `switchRoom` (replacing `switchArea` but doing a similar job) which will be invoked when the character is in the state ”going through a door”.

2.6.1 Task

You are asked to code the concepts described above in accordance with the given specifications and constraints. Start by testing with the simplest map and then with the one with 5 rooms. You will then check :

1. that the movement constraints established in the previous step remain valid (the character cannot walk on the ”walls” or leave the map) ;
2. that characters can open/close all unlocked connectors with the `T` key ;
3. that he can walk to each of the rooms on the map through open portals ;
4. that he can collect the staff (if any) and the key ;
5. that once the key has been picked up, he can open the locked portal associated with that key using the `W key` ;
6. and once the staff is picked up, the character can throw fireballs with the `X` key.

Note : connector status is specific to each room. Thus, it is normal that in the second map for example, if you transit from `[0][1]` to `[0][2]` the west connector of `[0][2]` is closed even if the connector east of `[0][1]` is opened.

2.7 Step 2 validation

To pass this step, all checks in sections 2.2.2 and 2.6.1 must have been completed.

The game `ICrogue` whose behavior is described above is to be returned at the end of the project.

This part is deliberately less guided. It is important to think carefully about where to place the necessary attributes and methods to ensure a high degree of isolation.

3 Challenges and Enemies (Part 3)

This part aims to complete the general logic of the game by integrating the challenges whose resolution will allow building connectors as well as some enemies to spice things up a bit.

3.1 Signal Dependent Rooms

The tutorial introduced the notion of *signal* which can be exploited to set up the logic of the game. The goal is to make the closed connectors in a room open when the challenges associated with them are met. For example, for a `Level0StaffRoom`, the challenge will simply be to pick up the staff there. A room is said to be "solved" when it has been visited by the player and he has successfully completed the challenges specific to it.
The idea is therefore to ensure that each room *behaves like a* logic signal (`Logic`) which is enabled when resolved and disabled otherwise.
Make the necessary changes to your code so that :

- an `ICrogueRoom` behaves like a logical signal and that it must have been visited by the player in order to be able to be resolved ;
- a `Level0ItemRoom` is resolved when, in addition to having been visited, all of its items have been collected.

Note : introduce a boolean attribute indicating whether a room has been visited and make this value become true when the player enters the room (remember that the person has a method to enter an area).

3.2 Rooms with enemies

You are now asked to create rooms populated by enemies. The corresponding challenge will obviously be to beat them.

3.2.1 Enemy

An enemy, to be implemented in the sub-package `icrogue.actor.enemies`, is an `ICrogueActor` which can be dead or alive and whose status (dead or alive) can be interrogated. It has a method that causes it to die and is built on the basis of the following information : its area to which it belongs, its orientation, and its initial position in the area (`DiscreteCoordinates`).
You will consider at this stage only one concrete type of enemies (which will therefore play the role of "boss") : `Turret` which will be able to throw arrows at regular time intervals. It's up to you to implement extensions to enrich the range of opponents at will :-)
The `Turret` can be drawn using the sprite :

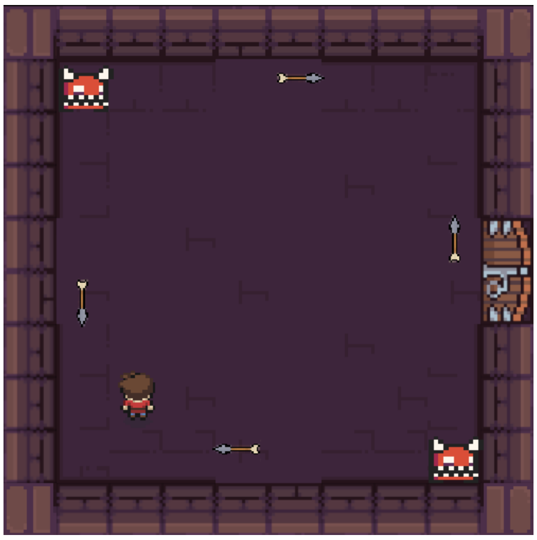


FIG. 12 : Turret enemies throwing arrows : In this example, the top left turret shoots right and down arrows and the bottom right shooter shoots up and left arrows.

```
new Sprite("icrogue/static_npc", 1.5f, 1.5f,
this, null, new Vector(-0.25f, 0));
```

The `Turret` can send arrows in multiple directions. It is therefore characterized by the set of directions (`Orientation`) towards which he shoots his arrows. This set is specified during construction (one can for example create a `Turret` which sends arrows up and down or only to the left).
The arrows (`Arrow`) are projectiles (much like the fireball and are built in a similar way) characterized by specific damage points (1 for example).
The associated sprite is calculated by :

```
new Sprite("zelda/arrow", 1f, 1f, this, new
    RegionOfInterest(32*orientation.ordinal(), 0, 32, 32), new
    Vector(0, 0));
```

where orientation is the orientation of the arrow.
An arrow interacts by contact with the player by inflicting its damage points, and disappears once it hits him.
So that the `Turret` can shoot his arrows at regular time intervals, you will give him a specific waiting time (for example `COOLDOWN` equal to `2F`) and a counter counting the time elapsed between two arrow throws. The counter will be initialized to zero and incremented by `dt` at each step of the simulation. When it reaches the value of `COOLDOWN`, the enemy goes on the attack (and the counter resets of course properly).
An attack consists of throwing an arrow in each of the possible directions (see figure 12). For his part, the player can defend himself by throwing fireballs or walking on the enemy. If a `Turret` has a contact interaction with a fireball, it dies (and the fireball disappears). He also dies if the character steps on him.

3.2.2 Rooms with Enemies

A `Level0EnemyRoom` is a `Level0Room` characterized by a *list of active enemies.*

It has a method allowing to give a value to this list (only the subclasses of `Level0EnemyRoom` and classes in the same package should be allowed to use this method).
A `Level0EnemyRoom` is resolved when the list of its active enemies is empty. It is responsible :

- to record all of its active enemies when initializing the area (method `createArea`) ;
- to unregister the dead ones between two simulation steps.

A `Level0TurretRoom` is a `Level0EnemyRoom` containing two `Turret`, both pointing upwards, one positioned at `(1, 8)` and throwing arrows down and right and the other positioned at `(8,1)` and throwing arrows up and left.

3.3 Game logic

To complete the set, we want to condition the general logic of the game to the success of a level. We want to model the fact that :

- any game level has an associated "boss" room[7] ; the coordinates of this room in the map are a characteristic of the level and they are by default `[0][0]` ;
- a level behaves like a logical signal : it is resolved when its "boss" room is.

Indication : to remain compatible with the previous phases of the development of the game, you will cover the case where there is no "boss" room : the level is in fact solved if the "boss" room is not equal to `null`. The controls `T` `L` and `O` no longer have any reason to exist, you can comment them out (possibly you can keep them as a "cheat" key for testing purposes).
When a game level is solved, the game can go to the next level if it is not already terminated. The game therefore ends either in this case or when the player dies. To simplify, a simple message *`"GameOver"`* or *`"Win"`* in the console is enough. It's up to you in the extensions to code it differently and introduce other levels if you wish.

3.3.1 Map with "boss" room

To be able to test the set, make sure that `Level0` creates the map of figure 13 using a utility method `generateFinalMap()`. This is the same map as the one generated by `generateMap2()` but the room `[0][0]` is now a `Level0TurretRoom`.

3.3.2 Task

You are asked to :

- code the elements suggested above according to the specifications and constraints described ;
- test them using the map generated by `generateFinalMap`.

You will check that :

1. level `Level0` corresponds to the map of figure 13 (if the latter was chosen as the card when the game was launched `ICrogue`) ;
2. the player can explore the level `Level0` and move from one room to the other through the open connectors ;

[7]the "boss" can actually be any challenge more complex than the others, this is the "ultimate level challenge"

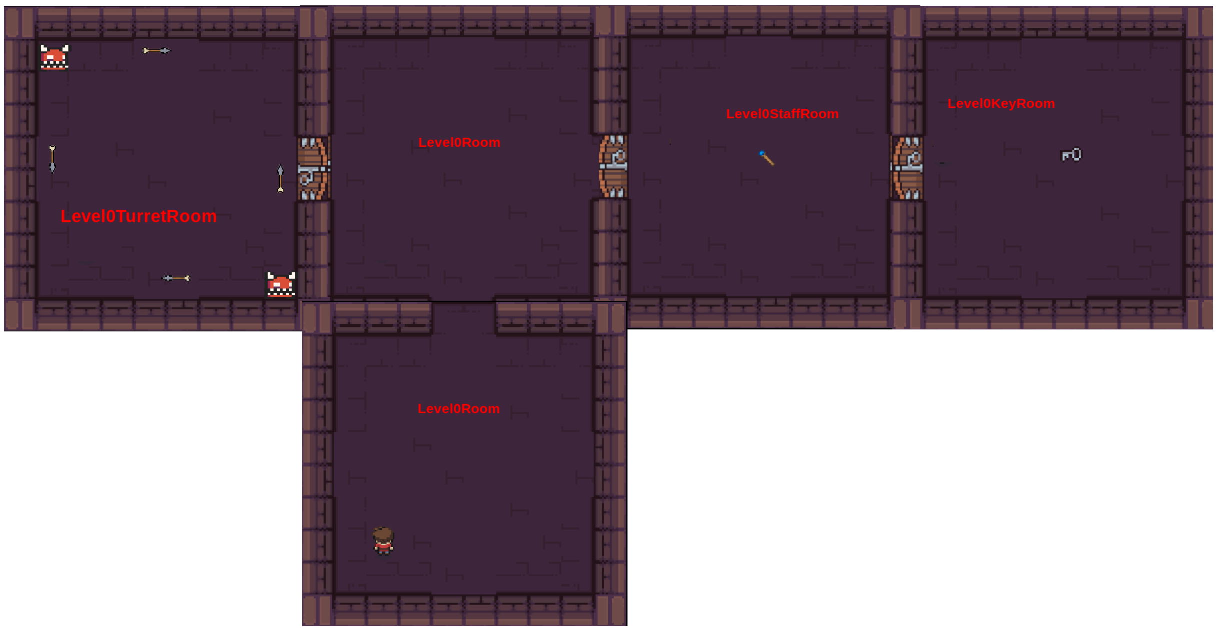


FIG. 13 : The room `[0][0]` is always locked by the room key `[3][0]`.

3. that the non-keyed connectors open according to the challenges of each room type ;
4. that the key to the boss room opens it ;
5. than the boss (`Turret`) throws arrows at regular intervals and that these arrows can kill the player ;
6. that the player can access the "Boss" room if he has collected the associated key beforehand ;
7. that the player can kill the enemies by stepping on them or by throwing fireballs at them (you can "cheat" the value of the `COOLDOWN` to achieve your goals faster ;-)). He have to pick up the stick first ;
8. that the level is resolved when the player kills the `Turret` or when he dies (resulting in a message *"Win"* or `Game Over!` accordingly).

3.4 Step 3 validation

To validate this step, all the checks of section 3.3.2 must be done.

The game `ICrogue` whose behavior is described above is to be returned at the end of the project.

To code this part you are relatively free regarding the implementation details. Your implementation should adhere to the specification and adhere to the principles of good object-oriented design. Do not code this part by reading the statement linearly. It is important to have a good understanding of what needs to be implemented and where it makes sense to place the suggested methods.

4 Random level generation (step 4)

This last step involves introducing one of the elements that make "RogueLike" type games specific : the*random generation of levels*. Essentially, it is a matter of providing game levels with a method `generateRandomMap` who will be in charge of creating the rooms randomly. The suggested algorithm has two phases :

1. the first decides the *placement of the rooms* ; it chooses which map coordinates will correspond to rooms ;
2. the second *generates the rooms* at the places decided by the first phase.

These two phases take place in principle according to the same logic whatever the level. To implement them, a simple idea is to consider that each level :

- is characterized by a set of *types of rooms* `{Type1, Type2... TypeN}` in a predetermined order (for example `Level0` would have the following typical room types : `TurretRoom, StaffRoom, Boss_key, Spawn, Normal` in this order) ;
- can be generated based on a desired distribution of its characteristic rooms. This distribution can be simply a set of integers describing the number of each level's type of rooms : for example the desired distribution 1, 4, 1, ... for `Level0` would mean that we want 1 room of type `TurretRoom`, 4 type rooms `StaffRoom`, etc.

Modify the constructor of `Level` so that it has the following header :

```
Level(boolean randomMap, DiscreteCoordinates startPosition,
    int[] roomsDistribution, int width, int height)
```

Its role will be to generate a fixed map of size `width*height` if `randomMap` is equal to `false` (using `generateFixedMap`). If the parameter is `true`, this constructor should generate a random map using `generateRandomMap` in terms of `roomsDistribution` : the random map will be of size `nbRooms * nbRooms` where `nbRooms` is the sum of the number of rooms specified by `roomsDistribution` (for example 7 for a distribution `{1,4,2}`).

4.1 Placement of rooms

You are asked to code a protected method :

```
MapState[][] generateRandomRoomPlacement()
```

which allows you to decide the placement of rooms on the map for any level.
The type `MapState` allows the placement algorithm to work by marking the status of a given position while the algorithm is running. It can be defined as follows :

```
protected enum MapState {
  NULL,               // Empty space
  PLACED,             // The room has been placed but not yet
     explored by the room placement algorithm
  EXPLORED,           // The room has been placed and
     explored by the algorithm
  BOSS_ROOM,          // The room is a boss room
  CREATED;            // The room has been instantiated in
     the room map

  @Override
  public String toString() {
    return Integer.toString(ordinal());
  }
}
```

the array returned by `generateRandomRoomPlacement`, which we will call *investment map*, is of the same dimensions as the map to be created.
The suggested algorithm for room placement is then as follows :

1. initialize all entries in the placement map to (`MapState.NULL`) ;
2. start by marking the room in the center of this map as placed (`PLACED`) ;
3. then, as long as there is a number `roomsToPlace` (>0), for each room placed but not explored :
   (a) calculate the number `freeSlots` of possible locations around this room (the locations at the top, bottom left and right which are within the boundaries of the map and are worth `MapState.NULL`) ;
   (b) randomly draw a maximum number of rooms to be placed (worth at most the minimum between `roomsToPlace` and `freeSlots`) ;
   (c) place these rooms randomly in the possible locations available (there are different ways to do this and you are free to implement it as you wish) ;
   (d) mark the room as explored.
4. finally place the ”boss” room : choose a free space next to a placed room and mark it with the value `MapState.BOSS_ROOM`.

The class `RandomHelper` provided in the subpackage `game.icrogue` puts at your disposal a minimal tool for the management of randomness. For all random draws related to map generation, you can use :

- the turn :

  ```
  RandomHelper.roomGenerator.nextInt(boundMin, boundMax)
  ```

  to randomly draw an integer between `borneMin` (included) and `limitMax` (not included) ;
- the method :

  ```
  List<Integer> RandomHelper.chooseKInList(int k,
      List<Integer> list)
  ```

  who returns `k` integers chosen randomly from the list `list`. This allows for example, if we have a set of `N` potential map positions to choose `k` among `N` (the method will give the indices to be exploited in the set in question). Example of use :

  ```
  RandomHelper.chooseKInList(3, List.of(1,4,4,5,6));
  ```

  returns 3 elements randomly chosen in the list, for example : (1,4,4) ou (4,5,1) ou (1,5,6) etc.

You will notice that the generation is actually only pseudo-random : from the same "seed" (`ROOM_SEED` from `RandomHelper.java`) it will always be the same sequence of numbers that will be drawn (this can make debugging easier).
Finally, to make it easier to check for random placements, you can incorporate the utility method into your code :

```
private void printMap(MapState[][] map) {
  System.out.println("Generated map:");

  System.out.print("  | ");
  for (int j = 0; j < map[0].length; j++) {
    System.out.print(j + " ");
  }
  System.out.println();
  System.out.print("--|-");
  for (int j = 0; j < map[0].length; j++) {
    System.out.print("--");
  }
  System.out.println();

  for (int i = 0; i < map.length; i++) {
    System.out.print(i + " | ");
    for (int j = 0; j < map[i].length; j++) {
      System.out.print(map[j][i] + " ");
    }
    System.out.println();
  }
  System.out.println();
```

```
}
```

which displays a map on the terminal.

4.2 Room generation

The creation of rooms can be done on the basis of a placement scheme `roomsPlacement` of type `MapState[][]` and a desired distribution `roomDistribution` of type `int[]` (it's up to you to see if it makes sense to make them attributes or method parameters). The suggested simple algorithm is :

- For any index entry `i` in `roomDistribution` :
  - there is `k = roomDistribution[i]` type rooms `i` to place : choose for it randomly `k` usable locations (all those worth `PLACED` or `EXPLORED` in `roomsPlacement`) ;
  - create at these locations a room of type `i` (be careful to create new rooms each time !) ;
  - mark the corresponding positions as `CREATED` in `roomsPlacement`.
- set up the connectors between the rooms created ;
- create the "boss" room in the place reserved for it and set up the connectors that connect it to its neighbors.

Note : it is considered that this algorithm is the same for all types of levels. It is understood however that the actual creation of a room of the type `i` can only be done in concrete levels. The same goes for the "boss" room and for placing the connectors. You will ensure that your implementation respects this.
For the installation of the connectors, it is reasonable to introduce a protected method :

```
setUpConnector(MapState[][] roomsPlacement, ICRogueRoom room)
```

invoked in the general algorithm and responsible for creating connectors.
Its concrete definition can obviously only be made in the concrete subclasses of levels. For example, in `Level0` this method can iterate over each of the possible connectors for this type of level. For each of the connectors allowing entry into a room (destination coordinates corresponding to a room different from `MapState.NULL`),it must update its destination coordinates and state appropriately. All connectors will be created closed. The destination coordinates can be calculated as follows :

```
DiscreteCoordinates destination =
    roomCoordinates.jump(connector.getOrientation().toVector());
```

where `roomCoordinates` represents the coordinates of the room to which these connectors belong. This method should also be responsible for ensuring that all connectors in the "boss" room are locked using the level-specific key.

4.3 Adaptation of the existing code

To complete this material, it is necessary to finalize class `Level0`. You must first describe the distribution of rooms desired for this type of level. For this, you can use the type listed as outlined below (to be completed by you) :

```java
public enum RoomType {
  TURRET_ROOM(3), // type and number of roon
  STAFF_ROOM(1),
  BOSS_KEY(1),
  SPAWN(1),
  NORMAL(1);
  //..
}
```

You can formulate this enumerated type as you see fit. In the one suggested, the idea is to have three `TurretRoom` rooms, one `Level0StaffRoom` room, one `Level0KeyRoom` room which would contain the key of the "boss" room. All the others would be `Level0Room`. The one corresponding to `SPAWN` would be moreover the starting room of the level.
Note : endow the enumerated type with a method that returns a distribution list as needed by the superclass constructor.
The class `Level0` will now have 2 constructor overloads :

- `Level0(boolean randomMap)` which appropriately invokes its superclass's constructor passing to it its specific distribution list. Dimensions for fixed map generation are always default`[4][2]` ;
- `Level0()` which defaults to random generation.

You can change the call to this constructor in `ICrogue` (by passing to it nothing or `false` so that a fixed level or a random level is generated).
The "Boss" room and the starting area are therefore no longer fixed. Make sure that the "hard" values initialized so far when building the levels are only valid for fixed (non-random) maps.

4.3.1 Task

You are asked to :

- code the elements suggested above according to the specifications and constraints described ;
- test them by viewing the result using the suggested print function (which you can invoke in `generateRandomMap`).

You will check :

1. that it is now possible to generate on demand (according to the parameter passed to the constructor of `Level0` in `ICrogue`) levels randomly ;
2. that the levels generated obey the constraints of the desired distribution (adequate number of rooms of each type, no rooms disconnected from the others, presence of a "Boss" room) ;
3. that the connectors are correctly positioned and functional ;

4. that the general logic of the game remains respected for the two types of levels (random and fixed).

4.4 Step 4 validation

To validate this step, all the checks of section 4.3.1 must be done.

The game `ICrogue` whose behavior is described above is to be returned at the end of the project.

5 Extensions (step 5)

To obtain bonus points (which can compensate for possible penalties on the compulsory part) or to participate in the contest, you can code some freely chosen extensions.
You can gain a maximum of 20 additional points : coding a lot of extensions to compensate for the weaknesses of the previous parts is therefore not an option.
The implementation is free and very little guided. Only a few suggestions are given below. Do not hesitate to contact us for an evaluation of the scale of your extensions if you decide to go in a specific direction. A small bonus may also be awarded if you are inventive in the design of the game.
You can code your extensions in-game `ICRogue` through additional levels or in a new game using the logistics you put in place in the previous steps. The basic game (with the level `Level0`) must be able to remain playable as requested in the compulsory part.
You should **comment carefully** the implementation of your extension in your `README`. In particular, we need to know which controls to use and with which effects without reading your code. Here is an example of a `README` (partial) that explains how to play a game :

- sample file `README.md` from an old project : README.md

You are expected to choose a few extensions and code them to completion (or almost). The idea is not to start coding lots of little pieces of disparate and unfinished extensions to collect the necessary points ;-). An estimate of the number of points associated with extensions is given. It may be worth more if more than expected effort is dedicated.

5.1 Extension Tracks

In reality, the game that you have coded can be enriched at will. In particular, the “signal” component can be leveraged to create increasingly complex challenges. You will find “in bulk” some suggestions for extensions in the following sections.

5.1.1 New Room Types

(number of points depending on the effort made)
Perhaps the most natural extension is to add new types of rooms characterized by new content and challenges. For example one can imagine adding a room characterized by a store where the character could buy useful objects. For new content and challenges you can take inspiration from the following section. It is also possible to add new types of levels (with potentially other connectors). It is important in your additions to respect the spirit of ”RogueLike” type games : the rooms must always be able to be visited in any order before arriving at the final room of a level. Clearly, there should not be explicit conditions in the levels imposing having to visit a room before another.

5.1.2 New actors or player extensions

All kinds of actors can be considered and their visual rendering can be improved.

- animations : instead of representing the actors by means of a `Sprite` unique, it is possible to associate them with an animation which would be a sequence of `Sprite` displayed in turn to give an illusion of movement ; the model offers a class `Animation` that you can use for this ; (~2 to 3 points)
- new objects to collect, even modeling of a system of resources : gold, money, wood, food, doses of care, etc. ; (~2 to 5pts)
- various actors that can serve as signals : orbs, torches, pressure plates, levers ; (~2 to 5 points)
- advanced signals for challenges (oscillators, signals with delay) : an oscillator is a signal whose intensity varies over time ; (~4pts/signal)
- all kinds of enemies with specific movement and behavior modalities ; (~2 to 5 pts/-character, up to 10 points if an AI is coded)
- introduction of friendly characters who can help the character (notably through dialogue, see below) ; (~2 to 5 pts/character)
- create a follower character like Red's Pikachu in pokemon yellow ; (~3 points)
- create one or more scenario events that are triggered with signals. For example, a type of room can be linked to a signal that when "on" would cause a character to appear. The latter in turn gives an object or an instruction to solve the challenge of the room. We can also imagine rooms with a variable number of enemies and that an event (like staying too long in the room) doubles the number of enemies etc. (number of points depending on the complexity of the addition)
- make behaviors more complex (for example, give characters periods of immunity) ; (~2 points)
- add new cell types with appropriate behaviors (water, ice, fire, etc.) ; (~2pts/cells)
- add a shadow or reflection to the player and some actors ; (~3 points)
- add new advanced controls (interactions, actions, movements, etc.) ; (~2pts/control)
- add random events (sets, signs, etc.) ; (~4pts)
- etc

5.1.3 Increase the amount of randomness

(number of points depending on the effort made)
"RogueLike" type games are often characterized by a large part of random aspects : room with a random number of enemies or objects, aspects of actors changing randomly, etc. Any idea in this direction is welcome.

5.1.4 Dialogues, pause and end of game

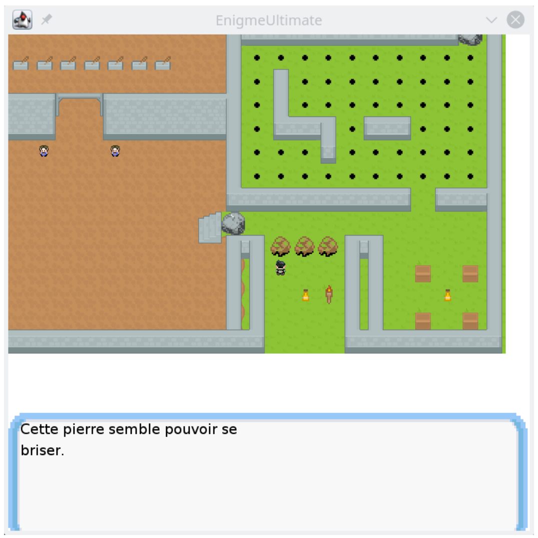
If the challenges are complex, the game can also quickly become unplayable without a few indications provided wisely. Introducing the possibility of using some dialogues can therefore be a very pleasant addition. Furthermore, setting up a system for pausing and resuming the game or managing endings with a "game over" screen (typically when the character has no more life points) would naturally complete your game.
Some indications are provided below to go in this direction.

Dialogs

(number of points depending on the effort made to set up the dialogues).
It is possible to attach texts to the actors (much like we did with `SimpleGhost`). You can initiate dialogs in certain situations. For example, when the character requests an interaction with an actor, an indication can then be displayed to put the player on the track of what must be done for the interaction to become possible.
Predefined texts for the dialog can be stored in files with format `.xml` like the one provided in `res/strings/enigma_fr.xml`. For example :

```
import ch.epfl.cs107.play.io.XMLTexts;
...
XMLTexts.getText("use_key");
```

will return the string `"use a CastleKey!"`. The file `xml` associated with the game is defined in `Play`.
Alternatively, and in a way a little closer to what is done in games of the type *gameboy*, you can leverage the provided class `Dialog` (of `rpg.actor`) and get dialog visualizations that look like this :



Cette pierre semble pouvoir se
briser.

Pausing the game and ending the game (~2 to 4pts)
The concept of an area can be exploited to introduce the pause of games. At the player's request, the game can switch to pause mode and then switch back to game mode. You can also introduce the management of the end of the game (if the character has reached an objective or has been beaten for example).
You can, in general, let your imagination run, and try your own ideas. If you come up with an original idea that seems to differ in spirit from what is suggested and that you wish to implement for the rendering or the competition (see below), you must have it validated before continuing (by sending an email to CS107@epfl.ch).

> However, be careful not to spend too much time on the project to the detriment of other branches !

5.2 Step 5 validation

As the end result of the project, create a documented game scenario in the `README` involving all coded extension components. A (small) part of the grade will be linked to the inventiveness and originality you demonstrate in the design of the game.

6 Contest

Those of you who have completed the project with special effort (interesting gameplay, visual effects, interesting/original expansions etc.) can compete for the "Best CS107 Game" award.[8]
If you wish to compete, you must send us by **22.12 at 18 :00** a small "application file" by email to the address **cs107@epfl.ch**. It will be a description of your game and the extensions you have incorporated into it (on 2 to 3 pages in .pdf format with some screenshots highlighting your additions).

> The winning projects will be presented during the back-to-school week (February).

[8]We have planned a small "Wall of Fame" on the course web page and a small symbolic reward :-)