CS-107 : Mini-Project
ICMon

H. REMMAL, F. NEMO & J. SAM

-

VERSION 1.0.3

Contents

This document uses colors and clickable links. It's advised to view it in digital format

1 Presentation

Over the past few weeks, you've become familiar with the fundamentals of a small adhoc game engine (see tutorial) that lets you create two-dimensional grid-based games. The simple outline obtained is similar to what you might find in an RPG-style game. The aim of this mini-project is to take advantage of this to create concrete variations of another type of game.

The basic game you'll be asked to create is strongly inspired by the famous Pokémon® games. Figure 1 shows an example of the basic outline[1], which you can then add to as your imagination takes you.

In addition to its fun aspect, this mini-project will give you a natural way of putting the fundamental concepts of object-oriented design into practice. It will enable you to experiment with the fact that design at an appropriate level of abstraction enables you to produce programs that are easily extensible and adaptable to different contexts.

Step by step, you'll get to grips with the complexity of the desired functionalities and the interactions between components.

The project comprises four mandatory stages:

- Stage 1 ("Basic ICMon"): at the end of this stage you will have created, using the tools of the game engine provided, a basic instance of the game with an actor moving around a room and able to collect a ball by applying the interaction management mechanism.
- Stage 2 ("Events and actions"): in this stage, the essential logic of the game will be put in place, making it possible to code game scenarios where events must be chained together and cause different actions to be carried out.
- Stage 3 ("Pokémon and battles"): this stage introduces the core of the subject, the notion of pokémon and battles with them.
- Stage 4 ("Event sequences"): this step refines the logic of the game, introducing the notion of event sequences. This will allow us to model *game scenarios* that can be made further complex.
- Stage 5 (Extensions, optional): During this stage, you'll be offered a number of more free-form extensions, enabling you to enrich the game created in the previous stage in your own way, or to create new ones.

Code a few extensions (your choice) to earn bonus points and/or enhance the value of your project and enter the competition.

[1] see the video demonstration.

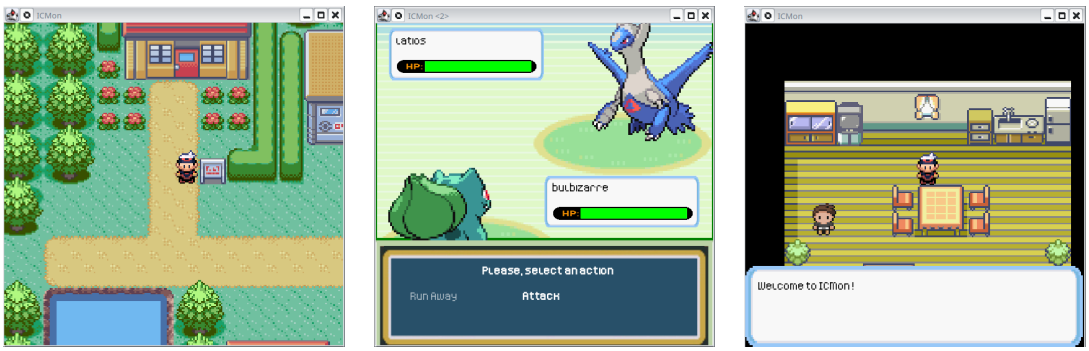

Figure 1: The character explores his environment according to different scenarios and collects 'Pokémon' with which he can fight those of his enemies.

Here are the main instructions for coding the project:

1. The archive supplied contains three folders: `game-engine` containing model tools, `tutos` which contains the tutorial solution, and `icmon` which you'll need to complete. **You will work exclusively in the `icmon` folder.**
2. The project will be coded using standard Java tools (imports starting with `java.` or `javax.`). If you have any doubts about the use of a particular library, just ask us, and pay particular attention to the alternatives that your IDE offers you to import on your machine. In particular, the project uses the `Color` class. You need to use the `java.awt.Color` version and not others.
3. Your methods will be documented according to javadoc standards (use the code provided as a guide). Your code should respect the usual naming conventions and be well **modularized and encapsulated**. In particular, avoid intrusive, publicly accessible getters on modifiable objects.
4. The indications can sometimes be very detailed. **This does not mean that they are exhaustive**. The methods and attributes required to carry out the desired processing are obviously not all described, and it's up to you to introduce them as you see fit, while respecting proper encapsulation.
5. Your project must not be stored on a public repository (such as GitHub). We recommend using GitLab, but any type of repository is acceptable as long as it's **private**.

2 Basic ICMon (step 1)

The aim of this step is to start creating your own little "ICMon" game in the modest vein of "Pokémon®". This basic version will contain a main character able to walk around a room in various ways and interact with an object in a basic way. This will be done according to the more general mechanism of *actor interactions*, as described in tutorial 3. At this stage, the game will involve:

- a main character;
- a ball for him to pick up.

You will be working in the provided `ch.epfl.cs107.icmon` package of the `icmon` folder.

2.1 Preparation of the `ICMon` game

> The tutorial solution is provided in the `tutos` folder, so you can use it as a starting point.

Prepare an `ICMon` game based on `Tuto2` (see the solution provided in the `tuto2` subpackage of the `tutos` folder). To begin with, this will be made up of:

- The `ICMonPlayer` class, which models a main character, to be placed in `icmon.actor`; leave this class empty for now, as we will come back to it a little later.
- The `ICMon` class, equivalent to `Tuto2`, to be placed in the `icmon` package; this class will have an `ICMonPlayer` as its character. Don't forget to adapt the `getTitle()` method, which will return a name of your choice (e.g. *`"ICMon"`*).
- The `ICMonArea` class, equivalent to `Tuto2Area`, to be placed in the `icmon.area` subpackage.
- The `Town` class, inheriting from `ICMonArea`, to be placed in a subpackage `icmon.area.maps` subpackage (equivalent to the `Ferme` or `Village` class of `tuto2.area.maps` in the `tutos` folder).
- The `ICMonBehavior` class, analogous to `Tuto2Behavior`, to be placed in `icmon.area` and which will nest a public `ICMonCell` class equivalent to `Tuto2Cell`.

However, a number of minor tweaks will be required to adapt to the spirit of the new game. We advise you to be careful when implementing these adaptations (to get things off on the right foot ;-)).

2.2 Adaptation of 'ICMonBehavior

In principle, `ICMonBehavior` and `ICMonCell` are equivalent to `Tuto2Behavior` and `Tuto2Cell`.

A first difference, however, lies in the fact that the cells will allow the main character to move in different ways, which can be modeled by an enumerated type:

```
public enum AllowedWalkingType {
        NONE,   // None
        SURF,   // Only with surf
        FEET,   // Only with feet
        ALL     // All previous
    }
```

The enumerated type describing the cell type and the associated movement mode can be described as follows:

Figure 2: First area in an ICMon game

```
        NULL(0, AllowedWalkingType.NONE),
        WALL(-16777216, AllowedWalkingType.NONE),
        BUILDING(-8750470, AllowedWalkingType.NONE),
        INTERACT(-256, AllowedWalkingType.NONE),
        DOOR(-195580, AllowedWalkingType.ALL),
        INDOOR_WALKABLE(-1, AllowedWalkingType.FEET),
        OUTDOOR_WALKABLE(-14112955, AllowedWalkingType.FEET),
        WATER(-16776961, AllowedWalkingType.SURF),
        GRASS(-16743680, AllowedWalkingType.FEET);
```

The nature of the "scenery" will no longer be the only element conditioning the character's movement. In the case of this new type of game, the presence of another actor who won't let himself be "walked over" will also hinder the character's movement. An object is walkable if its `takeCellSpace()` method returns `false`.

In concrete terms, two entities for which the `takeCellSpace()` method returns true cannot both live in the same cell. You will need to make sure that the `canEnter()` method of `ICMonCell` guarantees this.

Make the adaptations suggested above in the `ICMonBehavior` class.

2.2.1 Task

You are asked to code the concepts described above according to the given specifications and constraints. Start your `ICMon` game. Check that the empty area in figure 2 is displayed.

2.3 "ICMon"'s Actors

Now that the basic foundations have been laid, you will be asked to start modeling the actors in the `ICMon` games. They will be coded in the `icmon.actor` subpackage of the `icmon` folder.

You will consider that all the players involved in an `ICMon` game, the `ICMonActor`, are players who move on a grid (`MoveableAreaEntity`) and that, at this level of abstraction, they don't move in any specific way. The area to which they belong, their orientation and starting position, are specified at the time of construction. 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 `ICMonActor` is able to enter a given area, at a given position, and to leave the area it occupies (similarly to what `GhostPlayer`was able to do in `Tuto2`). As an `Interactable` it can be the object of contact interactions only (at this level of abstraction).

At this stage of the project, two more specific categories of `ICMonActor` need to be introduced: the *main character* and a simple *ball*. Let's start with the main character.

2.4 The main character

The `ICMonPlayer` class suggested above will be coded for the moment in the same way as `GhostPlayer`. However, you will need to review the inheritance links, as it's obviously also an `ICMonActor`.
One important difference is that the image used to draw it will depend on its *orientation* and will be *animated*. What's more, the animation will differ depending on whether the character is moving in water or on land.

To draw the character, you will use an `OrientedAnimation` object. More precisely, there will be two possible types of `OrientedAnimation`, one associated with movement in water and another for movement on land. The choice of current animation will depend on where the character is located, and the drawing method will be in charge of drawing the current animation.

An object of type `OrientedAnimation` is initialized as follows:

```
new OrientedAnimation(sprite_name, ANIMATION_DURATION/2, orientation, this)
```

where `sprite_name` is an image name and `ANIMATION_DURATION` is a constant to which you can give the value 8, for example and `orientation` is the chosen initial orientation. The two image names of interest here are *`"actors/player"`* for movement on land, and *`"actors/player_water"`* for movement in water. (See section 6.6 of the tutorial if you wish to understand the details of this code).

For the moment, when a character is spawned, you will choose "`actors/player`" as the current animation, and won't be interested in the mechanisms for switching to "`actors/player`" animation.

An `ICMonPlayer` behaves (updates) like an `ICMonActor`, but in a different way:

- it must be able to be moved using directional arrows, like the `GhostPlayer` coded in the tutorial;
- its current animation must also be updated according to the following algorithm: if a move is in progress, the current animation must undergo an `update`, otherwise, it must undergo a `reset`.
- the `moveIfPressed` method must orientate the current animation using its `orientate` method.

Finally, you will make sure that `ICMonPlayer` is not traversable.

Once `ICMonPlayer` has been coded, you can finalize the `begin` method of `ICMon` and start testing your game.

At launch, a player of type `ICMonPlayer` will be created at the position intended for it in the start-up area (take the value `(5,5)`). By default, it will be oriented downwards.

Figure 3: First ICBall

To facilitate testing, you will add the following control to `ICMon`: the `R` key should be used to reset the game, i.e. restart it in the same conditions as the very first time it was launched. Think of the modularizing your code here. In particular, note that the `begin` method of an abstract area game (from which `ICMon` inherits) empties the collections of areas and actors.

2.4.1 Task

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

Launch your `ICMon` game. You will see that it behaves as shown in the following short video: Step 1: character introduction (.mp4); concretely:

1. that the game starts with an ICMonPlayer looking forward (at the moment he's a bit biblical, as he's walking on water!);
2. that it can be moved freely around the area using the directional arrows, but that it must not be able to leave the area;
3. that its graphic representation is well adapted to its orientation;
4. and that its movements are animated.

2.5 Items to collect

We now need to model the objects that the main character will be able to collect, which will determine the course of the game. The objects to be collected will be implemented in a `actor.items` sub-package.

You are asked to introduce an abstract class `ICMonItem` modeling «collectable» objects and inheriting from the `CollectableAreaEntity` class supplied in the mock-up (package `areagame.actor`).

An `ICMonItem` is an object that cannot be traversed by default. It is drawn by means of a `Sprite` which characterizes it.

The area to which it belongs and its starting position within the area are given to the constructor, along with the name of the associated `Sprite`. Finally, an `ICMonItem` is oriented downwards by default. The `Sprite` of a "pickable" object can be constructed using the turn:

```
new RPGSprite(spriteName, 1f, 1f, this);
```

As an `Interactable`, by default it can be the object of contact interactions only (remember the `isCellInteractable` and `isViewInteractable` methods).

Only one concrete type of `ICMonItem` needs to be coded at this stage: a ball modeled by the `ICBall` class. A ball is an `ICMonItem` that accepts remote interaction (the character can invoke it if it's close enough to him [2]).
Its home range and starting position are given as parameters to the construction.

Finally, complete the code for `Town` so that when it is created, it registers a ball with starting coordinates `(6,6)`.

2.5.1 Task

You are asked to code the concepts described above according to the given specifications and constraints. Run your game and check that the ball appears as shown in figure 3.

2.6 First interactions

You are now asked to apply the interaction schema suggested in the third part of the tutorial (Clickable link)[3].

The first step is to enable the `ICMonPlayer` to interact with cells (i.e. objects of type `ICMonBehavior.ICMonCell`). Interaction will take the form of the character changing its representation according to the type of cell it comes into contact with.

To introduce the interaction mechanism, start by setting up the fact that all `ICMonPlayers` become `Interactors` (they can cause `Interactables` to undergo interactions).

As an `Interactor`, `ICMonPlayer` must define the methods:

- `getCurrentCells`: its current cells (reduced to the set containing only its main cell, as you've already said);
- `getFieldOfViewCells()`: the cells in his field of vision consist of the single cell he is facing:

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

As an `Interactor`, he will systematically want all contact interactions (`wantsCellInteraction` always returns true). The fact that he wants remote interactions (`wantsViewInteraction`) will be conditioned by the game user: the `L` key will be used to indicate that the character wants a remote interaction. For example, if our character is in front of a ball, to indicate that we want him to pick it up, we press the `L` key.

[2] it's not up to you to test proximity to an object, as this is done by the game engine.

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

Let's move on to the actual management of interactions.

In the `icmon.handler` subpackage, complete the `ICMonInteractionVisitor` interface, which inherits from `AreaInteractionVisitor`. This interface should provide a default definition of interaction methods for any `Interactor` in the `ICMon` set:

- a game cell (`ICMonCell`);
- a main character in the game (`ICMonPlayer`);
- a ball (`ICBall`)

These (default) definitions will have an empty body to express the fact that, by default, interaction consists of doing nothing. `ICMonPlayer`, as the `Interactor` of the `ICMon` game, must provide a more specific definition of these methods if necessary.

Any concrete `Interactable` must now indicate that it accepts to have its interactions managed by an interaction handler of type `ICMonInteractionVisitor`. Their `acceptInteraction` method (empty until now) must be reformulated in this sense (in each of the classes concerned):

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

This method may contain more code depending on the case, but for the moment it's sufficient for the three types of `Interactable` available: `ICMonPlayer`, `ICMonCell` and `ICBall`.

To enable the `ICMonPlayer` to handle more specifically the interactions it's interested in, define a private nested `ICMonPlayerInteractionHandler` class in the `ICMonPlayer` class, implementing the `ICMonInteractionVisitor` class. Add the necessary definitions to manage cell interaction more specifically, according to the following algorithm: if the interaction is a contact interaction (parameter `isCellInteraction` is `true`), adapt the current animation according to the nature of the cell. `FEET` must visually give rise to a terrestrial movement (current animation = `actors/player`) and `SURF`, an aquatic movement (current animation = `actors/player_water`). For all other cell types, the current animation will be retained.

Note : The coding for this method should be no more than five to seven lines long. In accordance with tutorial 3, for this to work, it is necessary that:

- the `ICMonPlayer` attribute must include its own interaction handler (of type `ICMonPlayerInteractionHandler`;
- its `void interactWith(Interactable other, boolean isCellInteraction)` method delegates management of this interaction to its handler (`handler` below):

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

remember that this `void interactWith` method is automatically invoked by the game kernel for any `Interactor` it comes into contact with or is in its field of view (if necessary, review section 6.3.1 of the tutorial).

2.6.1 Task

You are asked to code the concepts described above according to the given specifications and constraints. Run your `ICMon` game. Check that it behaves as shown in the following short video: Step 1: interactions

with cells (.mp4); in other words, `ICMonPlayer` behaves as in the previous step, but changes its animations according to the type of cells it interacts with.

> The logistics set up for interactions, as outlined in the tutorials and put to concrete use in this first part of the project, may seem *a priori* unnecessarily complex. The advantage it offers is that it models, in a very general and abstract way, the needs inherent in many games where players move around a grid and interact either with each other or with the grid's content.

Let us see how easy it is to manage a new type of interaction.

2.6.2 Ball collection

Complete the `ICMonPlayerInteractionHandler` class so that a ball can be collected during a remote interaction (remember that the character can express the wish for a remote interaction using the `L` key). **Note:** In principle, there's only one method to add, and the coding for it should be no more than two or three lines long.

2.6.3 Task

Start your `ICMon` game. Check that when the character is facing the ball and is in its field of perception, the `L` key causes the ball to collect and disappear visually.

> Later on in the project, you will have to code many other interactions between actors or with cells. All future interactions must be coded according to the schema established in this section, and must not require any type-testing of objects.

2.7 Validation of step 1

To validate this step, all checks in sections 2.2.1, 2.4.1, 2.5.1, 2.6.1 and 2.6.3 must have been completed.

> The `ICMon` game, whose behavior is described above, is to be handed in at the end of the project.

This part remains deliberately guided as it introduces the fundamental workings of the game's operation. However, not all details are given, of course. It is important to be meticulous in the implementations and adaptations suggested throughout the sections and to understand how they work.
Here we deliberately adopt the strategy of breaking the work down into many small testable sub-steps, even if it means making some refinements from one step to the next (refactoring). This "small steps" approach is useful in programming to facilitate debugging.
If you use git, remember to save a functional sub-step before moving on to the next!

3 Events and actions (step 2)

In Pokémon-type games, the unfolding of a level or a game is according to a scenario that involves *events* that may depend on each other. For example, a game scenario may require that the main character must first have had two interactions with the character Garry before being able to pick up a ball. Each interaction, or the need to pick up the ball, is in itself an *event* planned in the scenario.

Each event, when it starts or ends, can then be accompanied by *actions* to be taken. For example, the completion of an event may require the action of starting another one.

Another important aspect is that the game must be receptive to *information emanating from the main character*. For example, the latter could inform the game that he wishes to engage in a battle with a Pokémon.

The purpose of this step is to code these fundamental mechanisms of the game's logic, namely the concepts of *events*, *actions*, and *communication between the game and its main character*.

At the end of this step, these concepts will be illustrated using very basic events and actions, but fundamental aspects of the game's logic will have been put in place.

3.1 Simple events and actions

The following is about starting to model the concepts of events and actions as outlined in the introduction.

3.1.1 Action

Actions will be coded in a sub-package `gamelogic.actions`.

An abstract action can be modeled by an interface, `Action`, simply offering the method `void perform()`, allowing it to be executed.

At this stage, and primarily for testing purposes, a single simple class named `LogAction` will implement this interface. This class models an action that issues an informational message. It will be characterized by the message in question in the form of a string, and its `perform` method will simply display this message on the terminal. The constructor of the class `LogAction` will initialize the message using a parameter.

Feel free to complicate these types of actions later if you wish.

3.1.2 Simple events

Events will be coded in a sub-package `gamelogic.events`.

An event in the game `ICMon`, an `ICMonEvent`, is an entity evolving over time (`Updatable`). It can be:

- started or not,
- completed or not,
- and suspended or not.

Simple booleans, for example, can model these possible states which are not exclusive. By default, they will all be initialized to `false`.

An event is characterized by four lists of actions. These contain respectively the set of actions to be executed when the event: starts, ends, is suspended, or is resumed after suspension (*"resumed"*).

As for the methods of the public use interface, you are suggested to code:

- `void start()`: which allows the event to begin. If it has already started, this method does nothing. Otherwise, it executes all the actions from the list of those to be executed when the event starts and marks the event as started.
- `void complete()`: which allows the event to be completed. If it is already completed or has not started, this method does nothing. Otherwise, it executes all the actions from the list of those to be executed when the event is completed and marks the event as such.
- `void suspend()`: which allows the event to be suspended. If it is already completed or suspended or has not started, this method does nothing. Otherwise, it executes all the actions from the list of those to be executed when the event is suspended and marks the event as such.
- `void resume()`: which allows the event to continue after suspension. If it is already completed or if it is not suspended or not started, this method does nothing. Otherwise, it executes all the actions from the list of those to be executed when the event is resumed after suspension and marks the event as such.
- `void onStart(Action action)`: which adds the action to the list of actions to be executed when the event starts.
- `void onComplete(Action action)`: which adds the action to the list of actions to be executed when the event is completed.
- `void onSuspension(Action action)`: which adds the action to the list of actions to be executed when the event is suspended.
- `void onResume(Action action)`: which adds the action to the list of actions to be executed when the event is resumed after suspension.

It must be possible to query an event to find out if it has started, completed, or suspended. Furthermore, none of the class methods should be redefinable.

Why is it useful to impose this last constraint?

The `update` method remains abstract at the `ICMonEvent` abstraction level and, at this stage, you are asked to introduce only one concrete class of event, the `CollectItemEvent` class.

This class models an event consisting of picking up an `ICMonItem`. The latter is an attribute and is initialized using a constructor parameter.

The `update` method of `CollectItemEvent` consists of completing the event when the `ICMonItem` is marked as collected.

3.1.3 (Temporary) refactoring of the `Town` Area

To test the operation of events, we will proceed very simply by assuming that the `Town` area has an event associated with it (later the events will be directly linked to the game).

You are therefore suggested to proceed as follows:

- add an `event` attribute of type `ICMonEvent` to the `Town` area;
- in the `createArea` method, following the creation of the ball, ensure:
  - that this event is an event related to the collection of the ball;
  - that at the start of the event a `LogAction` is executed (with an appropriate message for example the message *`"CollectItemEvent started!"`*);
  - and that when the event is completed another `LogAction` is executed (with an appropriate message for example *`"CollectItemEvent completed!"`*);
  - and that this event is launched.

Ensure that the `event` event evolves well over time in the most natural place to do so (`update` method of `Town`).

3.1.4 Task

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

- the game launches as before but in addition the message indicating that the collection event has started is displayed on the console;
- the main character behaves as before but as soon as the ball is picked up, the message indicating that the collection event is completed is displayed on the console.

3.2 Events Conditioning Interactions

So far, events act only according to time (the `update` method makes them evolve after the passage of a time step `dt`). The idea now is to ensure that an event can also condition the modalities of interactions between the main character and another `Interactable` of the game.

Let's take the example of a scenario where the main character must interact twice, distinctly, with a character *Gary*, before being able to achieve other goals.

Such a scenario would therefore include the events "first interaction with Gary" and "second interaction with Gary", and each of these events would potentially induce a specific behavior: for example, the opening of a dialogue between the character and Gary during the first interaction and the engagement of a battle between them during the second.

If we let only the character's interaction manager handle this, the task quickly becomes too complex (for example, here it would be necessary to be able to distinguish whether it is the first interaction or the second interaction). This would be manageable in a case as simple as this one, but quickly impracticable in the case of complex scenarios.

The idea is therefore now to extend the functionality of events so as to also make them capable of managing what must happen when an interaction occurs between the main character and a game entity (`Interactable`).

This allows triggering a *response linked to a specific event* rather than having to manage all the different possible interaction situations based on the game's progress in the character's interaction manager.

Moreover, for simplification, the only event tested in the previous part was directly created in an area. It is now a matter of getting closer to the true logic of operation of `ICMon` and ensuring that events are related to the game itself.

To begin and in order to illustrate events conditioning interactions, a new category of actors will be introduced, non-playable characters, with whom the main character will interact.

3.2.1 Non-playable characters

To begin with, you are asked to introduce the abstract concept of a non-playable character, `NPCActor` (in a package `icmon.actor.npc`). An `NPCActor` is an `ICMonActor` that can be drawn by means of a characteristic `Sprite`. Its area, position, orientation and the name of its `Sprite` are given as parameters for construction. The `Sprite` attribute can be constructed using the phrase:

```
new RPGSprite(sprite_name, 1, 1.3125f, this, new RegionOfInterest(0, 0, 16,
    21));
```

where `sprite_name` is the name of the `Sprite`.

As an `Interactable`, an `NPCActor` is defined as a non traversable object that accepts only remote interactions. The body of its `getCurrentCells` method is similar to that of `ICMonPlayer`.

The only concrete `NPCActor` type to be introduced at this stage is `ICShopAssistant`, whose `Sprite` name is *`"actors/assistant"`*.

To complete the integration of the `ICShopAssistant` concept into the game, don't forget to ensure that it accepts to have its interactions managed by an interaction manager of type `ICMonInteractionVisitor` and to make the necessary additions to the latter. At this stage, you'll find that `ICMonPlayerHandler` doesn't handle this interaction more specifically. We'll discuss the reasons for this in the next paragraph.

3.2.2 Events as interaction handlers

The idea now is to enable event-specific interactions between the character and other entities.

The model here is very simple to implement: just make an `ICMonEvent` behave like an `ICMonInteractionVisitor` interaction handler.

For testing purposes, you'll ensure that `CollectItemEvent`, as an interaction handler, specifically handles interaction with `ICShopAssistant` by introducing the method:

```
void interactWith(ICShopAssistant assistant, boolean isCellInteraction)
```

This method will, for the time being, simply display an appropriate message on the console; for example: *`"This is an interaction between the player and ICShopAssistant based on events!"`*

The modalities for invoking this method will be suggested in the following paragraphs.

In principle, events now need to know the main character because their `interactWith` method may potentially require it (think of the interaction example where the main character fights Gary).

Modify your code so that the main character is an attribute of `ICMonEvent` and that it is passed during construction.

3.2.3 Refactoring of `ICMon` and `Town`

As previously mentioned, it doesn't really make sense for a unique event to be associated with an area. It is now about revising the current design so that events are rather characteristics of the game itself.

You will begin by **removing the `event` attribute from `Town`**. Instead, you will ensure that the `ICMon` game is characterized not by a single event but by a *list* of events.

The instructions of `createArea` that consisted of creating the `CollectItemEvent`, associating it with actions and starting the event will have to be moved to the `begin` method of `ICMon`. You will also, of course, need to store this event in `ICMon`'s event list.

A problem that then arises is how to associate the `CollectItemEvent` with the ball that is, at present, registered as an actor in `createArea` of `Town`.

You are asked to remedy this by introducing a new action `RegisterinAreaAction` that allows for registering a given actor in a given area. The area and the actor will be parameters of the constructor. You will then need to:

- create the ball in the `begin` method of `ICMon` rather than in `createArea` of `Town`
- ensure that a `RegisterinAreaAction` action registering the ball in the current area is executed at the start of the `CollectItemEvent`.

Finally, it is no longer the `update` method of `Town` that should update the event, but the `update` method of `ICMon` that should update all the events in its list.

The fact that the game consists only of a single "hard-coded" event that is immediately started is obviously a point that will be improved in the later stages of the project. For the time being, it is only a matter of implementing and testing certain fundamental mechanisms of the game's logic.

3.2.4 Handling interactions related to events

You have previously ensured that events become interaction managers and that `CollectItemEvent` contains an `interactWith` method managing what should happen when the main character interacts with an entity as part of this event.

It is time to worry about when such methods will be invoked.

In fact, the content of the event list materializes a "state" of the game. Executing interactions related to this state amounts to executing the interaction methods specific to these events.

It is therefore suggested that you introduce a public nested class of `ICMon`, `ICMonGameState`, which will be in charge of this treatment. It will have as its sole content a default private constructor, doing nothing, and a method that codes according to this example:

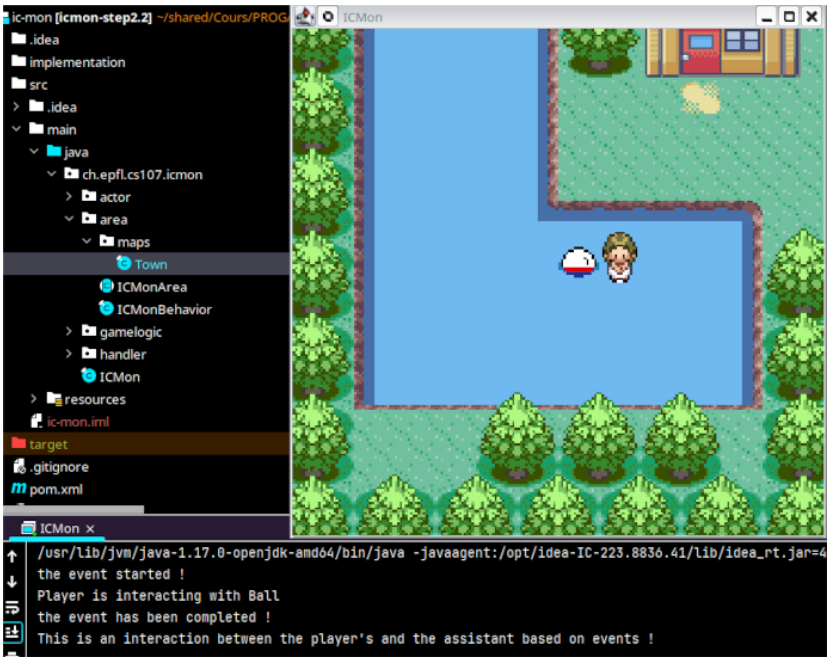

Figure 4: Example of execution of step 2.2

```
void acceptInteraction(Interactable interactable, boolean isCellInteraction){
  for(var event : ICMon.this.events)
                interactable.acceptInteraction(event, isCellInteraction);
}
```

where `events` is the list of events of `ICMon` (the method asks interactable to accept its interactions (with the character) to be managed by the events that constitute the state of the game.

It remains to allow the invocation of this method. As it aims to allow each event of the game to manage in its own way what happens when the main character interacts with an `Interactable`, it is natural that the `interactWith` method of `ICMonPlayer` be in charge of this invocation.

To make this possible, you are suggested to:

- equip `ICMon` with an attribute of type `ICMonGameState`
- and to transmit the value of this attribute to the character at its creation (the character is aware of the game state)[4].

3.2.5 Task

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

You will ensure that an `ICShopAssistant` facing downwards is registered in the `Town` area (using `createArea`). You could, for example, give it `(8,8)` as the starting position.

Launch your `ICMon` game. You will verify that:

[4]indeed, the character does not need access to the entire game, it is sufficient for them to know the "state of the game"!

- the message indicating that the collection event has started is still displayed on the console at the start of the game;
- the `ICShopAssistant` now appears
- the message indicating that the collection event is completed is still displayed on the console when the character picks up the ball;
- and that the message *`"This is an interaction between the player and ICShopAssistant based on events!"`* is displayed on the console when the character interacts with the `ICShopAssistant`.

In the example of figure 4, an interaction took place between the character and the ball (which made the latter disappear), then an interaction took place with the `ICShopAssistant` (key `L`).

Here you will note that it was the `ICMonCollectEvent` that took care of the interaction between the `ICShopAssistant` and the character and not the `ICMonPlayerHandler`. **That was precisely the goal to be achieved here**.

3.3 Multiple Events

So far, you have set up two fundamental elements of the project's architecture: the concept of *events* and *actions*.

These points were illustrated using a single event, `ICMonCollectEvent`, systematically started at the beginning of the game. This obviously does not suffice for the needs related to the development of a real game scenario.

The goal now is to allow for several events to take place according to a *chronology*.

The idea is to make `ICMon` behave according to the scenario of:

- launching a first event (ball collection);
- then, when this is completed, launching a second basic event ("end of the game").

3.3.1 "End of the Game" Event

Start by introducing an `EndOfTheGameEvent`. Its only role is to manage the interaction between the character and `ICShopAssistant` by mimicking a dialogue issued by `ICShopAssistant`. For now, this will be a simple message on the console (for example *`"I heard that you were able to implement this step successfully. Congrats!"`*).

3.3.2 Chronology of Events

It is now about allowing the fact that a game scenario may include a *sequence* of events.

More precisely, it is necessary to ensure that once an event has been completed it disappears from the game's event list and that the set of events to start or complete can change over time (between two consecutive calls to the game's `update` method).

A natural way to achieve this is to refine the modeling of events in `ICMon` by taking inspiration from what is done for the registration/deregistration of actors (if you are curious, you can look in the `Area` class of the mock-up), according to the following model:

- The list of events in `ICMon` is the list of active events

- During an `update`, events can start or be completed: those to start must be added to the list of active events and those completed must disappear from it.

Analogously to what is done in the game engine for the registration/deregistration of actors), it is therefore advisable to:

- provide two additional lists: one for the newly started events and one for the completed events (between two `updates`);
- at the beginning of the `update` of `ICMon`, use these lists to update the list of active events (remove completed events and add newly started ones);
- then empty these lists.

New Types of Actions For an event to register in the list of started events, it is sufficient to create an action `RegisterEventAction` that must be executed at the moment the event starts or is continued after suspension. Similarly, for an event to register in the list of completed events, it is sufficient to create an action `UnRegisterEventAction` to be executed when the event is completed or suspended. Remember methods, such as `onStart()`, `onComplete()`, which can be invoked in the constructor of an `ICMonEvent`. For example, as follows:

```
onStart(new RegisterEventAction(...))
```

This will have the effect of allowing the registration of the event in the game at the moment it starts.

The `perform` action of `RegisterEventAction/UnregisterEventAction` must therefore allow registering/deregistering an event in the `ICMon` game. To be able to execute properly, it must therefore be aware of the event to register/deregister and have access to the game. Consider introducing what is necessary in the construction parameters and as attributes.

> To avoid giving complete access to the game, it is possible to communicate only a restricted part of it; for example, an object `ICMonEventManager`, similar to `ICMonGameState`. This object could, for example, simply provide a means of registering or unregistering a given event in the game.

Furthermore, not all events are meant to be started systematically at the start of the game. The act of starting an action (invoking its `start()` method) can be done using an action `StartEventAction` that you are asked to introduce.

This allows introducing a chronology between events using phrases such as:

```
event1.onComplete(new StartEventAction(event2));
```

3.3.3 Refactoring of the `begin` Method of `ICMon`

In order to test your recent additions, modify the `begin` method of `ICMon` so that it:

- Creates the events “ball collection” and “end of the game”. The latter will be created similarly to “ball collection” with `LogAction` associated with the start and end of the event.
- Indicates that the “end of the game” event should begin when the ball collection has been completed.
- Starts the “ball collection” event.

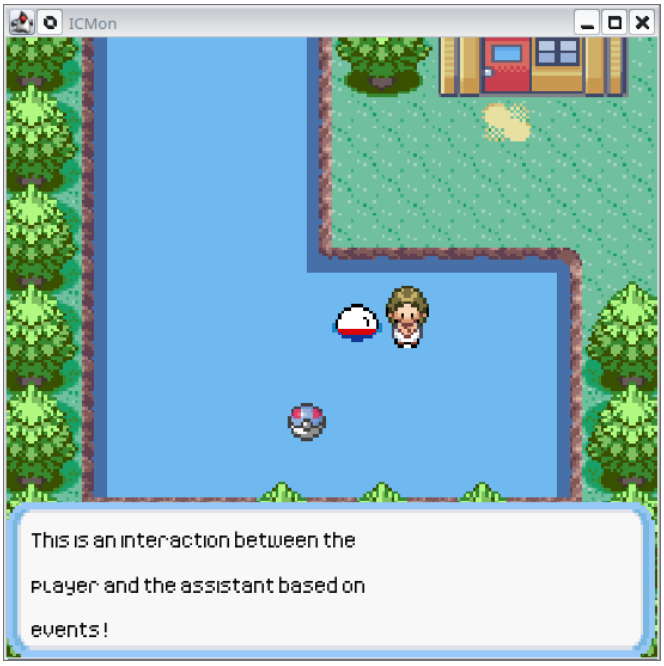

Figure 5: First dialogues

At this stage of the project, it is normal that there is nothing to complete the “end of game” event.

3.3.4 Task

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

Verify that the game behaves as before but that the interaction with `ICShopAssistant` that occurs after collecting the ball is well managed by an “end of game” event (different message). For example, a sequence “interaction with the assistant”, “collection of the ball” then “interaction with the assistant” should produce an output on the console resembling this:

```
ICMonItemCollect has started!
This is an interaction between the player and the assistant based on events!
Player is interacting with Ball
ICMonItemCollect has been completed!
the second event has started!
I heard that you were able to implement this step successfully. Congrats!
```

The last message should then be displayed each time you meet the assistant, as there is nothing to complement the ‘end of game’ event at the moment.

3.4 Dialogues

Interactions between the character and other `Interactable` objects are currently limited to displaying messages in the console. The goal now is to replace this with real "dialogues" as in the example of figure 5.

The mock-up already provides a `Dialog` class (in the `engine.actor` package) for this purpose. To create a dialogue, simply associate it with text present in the resources (subfolder `dialogs`), for example:

```
new Dialog("collect_item_event_interaction_with_icshopassistant")
```

The `update` method of `Dialog` allows the text to unfold. The `isCompleted` method returns `true` when all the text has been displayed.

Since it is the character who "talks" with other entities, the implementation of dialogue can be done by:

- equipping it with a `Dialog` type attribute, which models a speech it holds at a given moment, and a method `openDialog` to assign a value to this attribute;
- introducing two *states* for the character: one during which it "is in dialogue," where only the dialogue is updated (the `update` method of `Dialog`); the usual behavior of the character (moving, etc.), no longer takes place.

3.4.1 Refactoring the interactions

You are asked to refine the two interactions managed by the events so that they open a dialogue (you can keep the display on the console in addition for debugging purposes):

- `CollectItemEvent` will open the dialogue *"collect_item_event_interaction_with_icshopassistant"* when the character interacts with `ICShopAssistant`
- `EndOfTheGameEvent` will open the dialogue *"end_of_game_event_interaction_with_icshopassistant"*.

> Opening a dialogue obviously means associating the correct value with the `Dialog` type attribute of the character and switching it to the "in dialogue" state.

3.4.2 Refactoring of the `ICMonPlayer` class

Now refine the `ICMonPlayer` so that:

- it only wants remote interaction via the `L` key when it is not in dialogue;
- in the "in dialogue" state, the `Dialog` type attribute is updated when the space bar (`keyboard.get(Keyboard.SPACE).isPressed()`) is pressed, and the character switches back to its "not in dialogue" state when the dialogue is finished (`isCompleted()` method of `Dialog`). `keyboard` is the `Keyboard` of the player's owner area.

When the character is not dialoguing, it must have the behavior it had until now.

> Finally, consider that drawing an actor also involves drawing the associated dialogue when they are dialoguing.

Figure 6: Door from `Town` to `Lab`

3.4.3 Task

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

Launch your `ICMon` game. You will verify that it behaves as shown in the following short video: Step 2: events triggering dialogues (.mp4); if one interacts with the assistant before collecting the ball. The use of the space bar should allow the dialogue boxes to disappear.

3.5 Communication between the game and the character

One last element is now to be introduced to build the game logic: allowing the main character to express to the game requests that it is more able to execute.

For example, the main character might need to ask the game to stop (execute its `end()` method in some tragic situations where it is cruelly defeated, for example). This is a situation where the game has easier access to the useful data to perform the required processing than the one requesting their execution.

You are therefore suggested to implement this in the form of a *"mailbox"* in `ICMon` into which the main character can post a message representing a request to be executed in due time by the game itself.

To illustrate this concept, you are asked in what follows to add a new actor, the doors, allowing to transit from one area to another. The main character will ask the game, via messages in the mailbox, to make it cross these doors to change area.

3.5.1 "Door" actor and new area

A door, `Door`, is an `AreaEntity` type actor that allows for transit to a destination area. This actor is characterized by:

- the name of the area to which it allows transit (a string);
- and the arrival coordinates in the destination area (type `DiscreteCoordinates`).

It occupies at construction *a set of cells*. The constructor of a `Door` takes as parameters: the name of the destination area, the arrival coordinates in the destination area, the area to which it belongs, the position of its main cell, and possibly the list of coordinates of the cells it occupies in addition to its main cell (expressed using an ellipse in the second constructor). The `getCurrentCells()` method must therefore be redefined in `Door` to return the set of these coordinates.

A `Door` is a traversable actor that accepts contact interactions, and whose drawing method does nothing by default. Indeed, doors will coincide with elements of the decor and will not draw themselves as such. Therefore, their orientation is not very important. You could, for example, orient them by default upwards at construction.

You are asked to introduce this new actor and to register an instance with the following characteristics:

Destination area name	Arrival coordinates	Door's coordinates
"lab"	(6,2)	(15,24)

in the `Town` area. This amounts to positioning a door at the location indicated in figure 6.

Finally, create a new `Lab` area titled `"lab"` and register therein the door allowing transit in the other direction to `Town`.

Destination area name	Arrival coordinates	Door'scoordinates
"town"	(15,23)	(6,1)(7,1)

The locations of the doors can be found by counting the coordinates of the red squares in the "behavior" images (for example, see `resources/images/behaviors/lab.png`)

3.5.2 "Mailbox" (exchange of messages)

To simplify, the mailbox of `ICMon` can only receive one message whose type is now to be defined.

To test this concept of mailbox, the idea is to allow the character, at the moment of its interaction with a door, to send a message to the game. This message will have the role of asking the game to make the character transit to the door's destination area.

This can be done simply by a specialized definition of `interactWith` for doors and whose code would look like this:

```
@Override
public void interactWith(Door door, boolean isCellInteraction) {
    if (isCellInteraction)
        ... message = ...;
        game.send(message);
```

where `message` is a message that can be stored in the mailbox and `game` is the state of the game to which the character has access if you have correctly coded what is suggested in section 3.2.4.

The specific message treatment for door passing will obviously consist in making sure that the character leaves the area it occupies and enters the door's destination area, which will become the current area of the game.

Here are suggestions on how this can be set up:

- A hierarchy of `GamePlayMessage` models the different types of messages that the `ICMon` game can receive. This hierarchy is divided into several subclasses, such as `PassDoorMessage`.
- A `GamePlayMessage` has a `process` method performing the required processing: thus, `ICMon.update` could make a call to `process` on the single message in its mailbox and this would translate polymorphically into the appropriate processing according to the type of message. Typically, the `process` method of `PassDoorMessage` would undertake the actions allowing the character to pass a door.

Then the question arises of how useful information is communicated from one class to another, for example, how the `process` method of the message `PassDoorMessage` knows the door to pass and the character who must pass this door.

How do you propose to manage this while avoiding public getters on modifiable objects (typically the character)?

3.5.3 Task

You are requested to code the concepts previously described in accordance with the given specifications and constraints.

Ensure that the game behaves as before but in addition, the character can move back and forth between the `Town` area and the `Lab` area as shown in the video Step 2: doorway passage via message sending (.mp4)

The `ICMon` game, whose behavior is described above, is to be submitted at the end of the project.

4 "Pokémon" and Battles (Step 3)

This step finally takes us to the heart of the matter. It is about introducing battles between the main character's Pokémon and other Pokémons. At the moment, we are not concerned with how the main character collects Pokémon; that will be the focus of the final step of the project.

4.1 Battle Arena

To test this step, a specific area will be dedicated to engaging in battles. Therefore, you are asked to introduce a new area, `Arena`, into the game with the following characteristics:

- Its title is *`"arena"`*.
- A door allows passage from this area to `Town` (arrival coordinates: `(20,15)`, door coordinates: `(4,1)`, `(5,1)`).
- Another door allows passage from `Town` to `Arena` (arrival coordinates: `(4,2)`, door coordinates: `(20,16)`).

4.1.1 Task

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

Ensure that the game behaves as before, but now the character can move between the `Town` area and the `Arena` area, as shown in the video Step 3: Combat Area (.mp4).

4.2 Sketch of a Battle with Pause

In this section, the goal is to introduce:

- first types of `Pokemon` for testing purposes.
- the concept of a battle that suspends the normal course of the game and shifts it into a different execution mode.

Pokémons are meant to battle each other. The main character will have a certain number of them that can be selected as weapons to engage in battles against other Pokémons encountered during their journey.

A draft `Pokemon` class is provided in the `pokemon` subpackage of `actor`.

It needs to be completed. For now, you can ignore the nested `PokemonProperties` class.

4.2.1 First Pokemon

A `Pokemon` is an abstract `ICMonActor` characterized by:

- a specific name (a string).
- a number of hit points ("hp"), modeled as an integer.
- a maximum number of hit points.
- the number of damage points they can inflict (also an integer).

It is drawn using an `RPGSprite` whose associated image is the concatenation of *`"pokemon/"`* with its specific name:

Figure 7: Bulbasaur in the battle arena

```
new RPGSprite("pokemon/" + name, 1, 1, this)
```

Its `getCurrentCells` method is coded similarly to that of the main character (but feel free to reconsider this choice). By default, it is traversable, accepts contact interactions but not remote interactions.

It can receive a given number of damage points, decreasing its hit points accordingly (but the hit points cannot become negative). A `Pokemon` is considered dead if its hit points reach zero.

The coordinates of a `Pokemon`, its name, the number of damage points it can inflict, and its maximum hit points are given as parameters during construction. The hit points start with the maximum hit points.

The first concrete type of `Pokemon` is `Bulbizarre`, characterized by its specific name (no surprise: *"bulbasaur"*). All `Bulbizarre` inflict 1 damage point and have a maximum of 10 hit points.

Registering a `Bulbizarre` at `(6,6)` in the `Arena` area should make it appear as shown in Figure 7.

In the same vein, code the Pokémons `Latios` and `Nidoqueen` (for the moment, you can give them the same characteristics as `Bulbizarre`). The sprite names are "latios" and "nidoqueen".

4.2.2 Link between the character and Pokemon

The main character must now have a collection of Pokémon. Ensure that they can receive a Pokemon (i.e., store it in their collection), and for now, they will receive a `Bulbizarre` at construction.

4.2.3 Switching to "Combat" Mode

Now, the goal is to ensure that the contact interaction between a `Pokemon` and the main character switches the game to a combat menu, as represented in Figure 8. To anticipate the fact that the player will potentially have to fight not only `Pokemon`, but also entities which possess them, start by creating an `ICMonFightableActor` interface. The content of this interface is up to you and it will be implemented by the `Pokemon` class.

At this stage, only the logic of the *switching* will be implemented: a completely black pause menu will be displayed for a few seconds, and then the game will return to "normal" mode.

To achieve this, it is suggested to introduce the class `ICMonFight` (in a sub-package `fights` of `gamelogic`), which is a kind of pause menu (`PauseMenu` in the `engine` package of the mockup). An `ICMonFight` is

Figure 8: Switching to combat mode

characterized by two `Pokemon` engaged in battle, passed to it at construction.

For testing purposes, this class will redefine its `void update(float deltaTime)` method so that, after a call to the `update` method of its superclass, it simply decrements a counter attribute initialized, for example, to `5` seconds (`5.f`).

It should be possible to test if the menu is currently displayed using a method `isRunning()` that returns `true` if the counter is greater than zero. The `drawMenu` method inherited from higher up does not need to do anything for now.

The idea now is to enrich the game logic so that when the main character interacts by contact with a Pokémon, a `fight` method, parameterised by an `ICMonFightableActor`, is invoked. It will ensure that :

- A "combat" event (`PokemonFightEvent`) is registered in the game. This event is characterized by a pause menu `ICMonFight`.
- A message "suspension of ongoing events" (`SuspendWithEvent`), parametrized by the "combat" event, is sent to the game (sending the message: "suspension of ongoing events due to a combat event").
- The completion of the `PokemonFightEvent` should cause the action of making the battled actor disappear (leaving its area). In other words, for this stage, when the battle is over, the Pokémon involved disappears. A new type of `LeaveAreaAction` parameterised by an `ICMonActor` is suggested to carry out this treatment.

`SuspendWithEvent` message handling Upon receiving a message `SuspendWithEvent`parametrized by the event `event`, the game must analyze this event. If it is an event involving a "pause menu" (which is the case for combat), it must ensure that:

- its start causes:

  - the action of associating the relevant pause menu with the game (use the `setPauseMenu` method for this). The game `ICMon`, as a grid-based game, is capable displaying a pause menu when "paused". Such a pause menu can be assigned to it using its `setPauseMenu` method.
  - the action of invoking a pause in the game (method `requestPause`)
- its completion causes:
  - the action of ending the pause in the game (method `requestResume`)

The game's handling of a message `SuspendWithEvent` must also cause:

- the suspension of each ongoing event at the start of `event` (a `SuspendEventAction` is suggested to implement this treatment)
- the restart of each of these events upon completion of `event` (a `ResumeEventAction` is suggested to implement this treatment)
- the start of `event`.

Care must be taken to ensure that events are suspended and restarted as they were listed when the message was received. For this purpose, it is recommended to work on a copy of all the events.

4.2.4 Task

You are required to code the concepts described above in accordance with the specified specifications and constraints.

Verify that the game behaves as before, but when in contact with the Pokémon, a black window appears for 5 seconds. Afterward, the game resumes its normal course in the `Arena` area where `Bulbizarre` has disappeared.

4.3 Combat Graphics

Engaging in a battle results in a somewhat dark screen. It is time to fix that. The goal is to code the `drawMenu` method of `ICMonFight` so that the screen resembles the one shown in Figure 8.

Implementing these graphics involves code that is a bit tedious and time-consuming to write within the given timeframe. Therefore, we provide several pre-coded utility classes for these processes.

These classes are:

- `ICMonFightArenaGraphics`, which uses
  - `ICMonFightInfoGraphics`
  - `ICMonFightInteractionGraphics`.
- As well as `ICMonFightTextGraphics`.

These classes model the different graphical information zones of the combat window, as shown in Figure 9.

You are asked to make the necessary modifications to your code so that:

- A battle (`ICMonFight`) is characterized by the character's `Pokemon` and that of their opponent.

Figure 9: Graphic zones of the combat screen

- The constructor of `ICMonFight` now takes as parameters the values of these two `Pokemon` (`player` and `opponent`).
- The `draw` method of `ICMonFight` draws an attribute `arena` of type `ICMonFightArenaGraphics`. This object will have been constructed using a statement such as:

```
new ICMonFightArenaGraphics(CAMERA_SCALE_FACTOR, player.properties(),
    opponent.properties());
```

Also complete the nested class `Pokemon.PokemonProperties` so that the Pokemon's properties can be passed to `ICMonFightArenaGraphics`.

The interaction zone is what will later make it possible to have an interaction menu to influence the course of battles.

For the moment, it should only display a simple text, which can be obtained as follows:

```
arena.setInteractionGraphics(new ICMonFightTextGraphics(CAMERA_SCALE_FACTOR,
    "hello world"));
```

The main character should now have a collection of `Pokemon`. You'll make sure that he can receive a Pokemon (i.e. store it in his collection) and for the time being, it will immediately receive a `Bulbizarre` when it is built, as well as a `Latios` and a `Nidoqueen`.

Remember also to tweak the method of interaction between the character and a `Pokemon`, so that the `ICMonFight` created at this point is built with the right protagonists.

At this stage, the character's `Pokemon` will be the first one in his list.

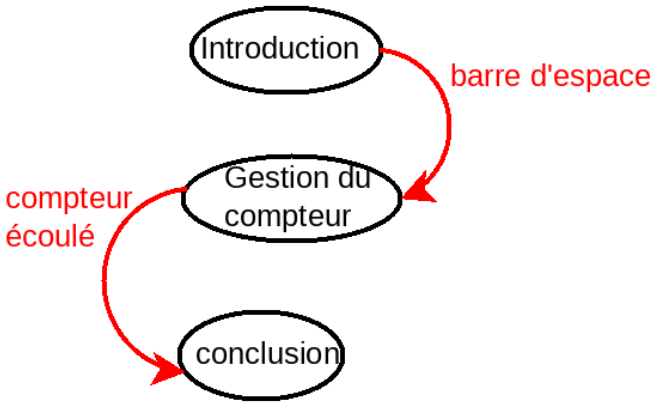

Figure 10: Finite State Machine describing the evolution of a battle (first simple version): black states, red actions causing transitions to another state

4.3.1 Task

You are required to code the concepts described above in accordance with the specified specifications and constraints.

Verify that the game behaves as before, but when in contact with `Bulbizarre`, the game behaves as shown in the video Step 3: graphical combat screen (.mp4) (the program freezes for 5 seconds on the combat screen, then returns to the game where `Bulbizarre` has disappeared, and the character can be controlled as usual).

4.4 Logic of battles: Finite State Machine

We are now interested in modeling the logic of battles. This logic can be implemented using what is called in computer science a finite state machine. Figure 10 gives a graphical representation of such a concept for the simple battle logic we will start implementing.

You are now asked to add to your code the elements allowing to model the fact that:

- a battle (`ICMonFight`) can go through three states: "introduction," "conclusion," and "counter handling" (this is expected to evolve later, the goal here is only to introduce the finite state machine concept);
- a battle (`ICMonFight`) has a *current state* which is the "introduction" state at its creation.

You will then modify the body of the `update` method of `ICMonFight` so that it implements the algorithm given by the finite state machine in Figure 10, typically:

If the current state is:

- *introduction*: ensure that the text `"Welcome to the fight"` is displayed in the interaction menu and that if the space bar is pressed, the battle transitions to the "counter management" state;
- *counter handling*: decrement the counter and display its value in the terminal (for debugging purposes), when the counter reaches zero, transition to the *conclusion* state;

- *conclusion*: ensure that the text *"Good fight!"* is displayed in the interaction menu and that if the space bar is pressed, the battle ends (which can be simply implemented by calling the `end()` method of `ICMonFight`).

Hints: use a `switch` statement and model the states with an *enumerated type* (be careful with the correct usage of `break`!).

4.4.1 Task

You are required to code the concepts described above in accordance with the specified specifications and constraints.

Verify that when in contact with the Pokemon, the game behaves as shown in the video Step 3: FSM draft for battles (.mp4): by pressing the space bar, you switch to a black interaction screen while the counter decrements, the message *"Good fight!* is displayed when the countdown is over.

4.5 Actions during Battles

The goal now is to make the combatants more active by giving them the ability to undertake *combat actions*. These actions should:

- Be queryable for their name (a string).
- Execute on a given `Pokemon`. The execution of the combat can proceed or be interrupted. A simple boolean as the return value of the action execution method can model this fact.

An interface `ICMonFightAction`, placed in the `fight` sub-package, is suggested to model this.

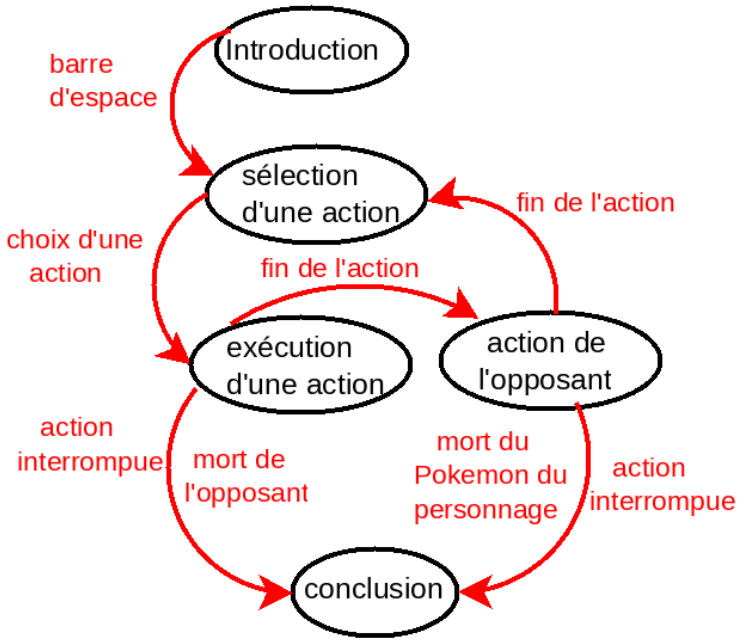

Figure 11: Automaton describing the evolution of a battle: in black are the states, in red are the actions causing transitions to another state

The concepts to model are as follows:

- Each Pokemon can undertake a list of `ICMonFightAction`.
- This list exists for each Pokemon, but its content is specific to each type of Pokemon.
- The Pokemon `Bulbizarre` can undertake two types of combat actions: *attack* and *escape*. You are free to adopt the names of your choice for these actions (for example, `"Attack"` and `"Run away"` that you can code in a sub-package `actor.pokemon.actions`). For now, these two attacks do nothing when executed. The execution method returns `false`, indicating that the combat action could not be completed.

4.5.1 Logics of Battle with Actions

To begin integrating the combat actions of Pokemon, the finite state automaton governing the battle mechanics must be revised to conform to that of Figure 11. In other words:

- The state "counter handling" no longer has a reason to exist; it is replaced by two new states: "action selection" to be undertaken and "execution of the selected action."
- The algorithm implemented by the automaton becomes the following:
  - If the current state is:
    - *"introduction"*: ensure that the text "Welcome to the fight" is displayed in the interaction menu, and if the space bar is pressed, the battle transitions to the "action selection" state.

* *"action selection"*: choose as the action to be undertaken the first one in the character's Pokemon list (this will be refined a bit later) and transition to the "execution of the action" state.
* *"execution of the action"*: execute the chosen action on the opponent's Pokemon. If the opponent dies as a result, transition to the "conclusion" state where an appropriate message should be displayed (for example, *"The player has won the fight"*). Otherwise, if the execution could not be completed (which is systematically the case for the escape action at the moment), also transition to the "conclusion" state where a different message should be displayed, for example, *"The player decided not to continue the fight"*), otherwise, transition to the "opponent's action" state.
* *"opponent's action"*: find if among the opponent's actions, there is an attack. If so, execute that attack on the character. If the character's Pokemon dies as a result, transition to the "conclusion" state where an appropriate message should be displayed (for example, *"The opponent has won the fight"*). If the execution could not be completed (which does not happen at the moment), also transition to the "conclusion" state where a different message should be displayed, for example, *"The opponent decided not to continue the fight"*), otherwise, return to the "action selection" state.
* *"conclusion"*: a message depending on the situation is displayed (as suggested in the description of the previous state), and if the space bar is pressed, the battle ends.

In the mandatory part of the project, for simplification, the enemy Pokemon can only use the attack action.

4.5.2 Task

You are required to code the concepts described earlier in accordance with the given specifications and constraints. Verify that when in contact with the Pokemon, the game behaves as shown in the video Step 3: Battle Mechanics (1) (.mp4): transitioning from one message to another is done using the space bar.

4.6 Selection of Actions to Undertake

The goal now is to be able to use an action selector so that, in the "action selection" state, the action to be executed by the character's Pokemon on the opponent can be chosen (instead of always using the first action in the list).

To facilitate your task, a class `ICMonFightActionSelectionGraphics` is provided, which you need to integrate into your code. You are asked, to begin with, to read the code of this class and understand how it works. You will, in fact, be required to produce very similar code in the next sub-step.

The aim is to ensure that in the "action selection" state, the menu from Figure 12 is displayed and allows the selection of the action to be executed. The right and left arrows will then allow selecting the desired action, which is displayed in bold. The `Enter` key validates the selected choice. The `choice()` method of `ICMonFightActionSelectionGraphics` returns the selected action.

You are asked to revise the finite state automaton of battles so that the transition to the "action selection" state:

- Triggers the display of the selection menu (remember the `setInteractionGraphics` method, which allows modifying the graphic object associated with the interaction zone of the combat screen).

Figure 12: Combat Action Selection Menu

- Calls the `update` method on this graphic object so that any user interaction for selecting an attack is taken into account.
- If an action has been selected (the `choice()` method returns something other than `null`), transition to the "execution of an action" state to execute the selected action.

It is up to you to decide where the `ICMonFightActionSelectionGraphics` object should be created. Note that its constructor takes as a parameter the list of actions of the character's Pokemon.

4.6.1 Task

You are required to code the concepts described earlier in accordance with the given specifications and constraints. Verify that your program behaves as shown in the video Step 3: Battle Mechanics (2) (.mp4):

- Initially, the space bar allows transitioning to the "action selection" state, which displays the selection menu.
- In this example, we select the `Attack` action each time with the right arrow and then `Enter`.
- The transition from the "execution of the action" state to the "opponent's action" state is almost instantaneous; both fighters suffer damage.
- The character has a head start and kills the opponent, then there is a transition to the conclusion state.
- The space bar allows returning to normal gameplay mode.

Figure 13: Menu for selecting the Pokémon to engage in battle (here the character would have 3 `Bulbizarre`)

4.7 Pokémon Selection (Optional)

To conclude this step, the idea is to model the fact that `ICMonPlayer` has a collection of `Pokemon`, and before engaging in a battle, the player can choose which one to use as a weapon.

You are asked to modify your code so that when encountering a `Pokemon`, instead of immediately engaging in battle with it, an intermediate step of selecting the Pokemon to use is introduced, as shown in Figure 13.

Here are some indications to implement this, in particular by modifying the `fight` method:

- At the time of interaction between the character and a Pokemon, a new type of event (`PokemonSelectionEvent`) is created instead of the `PokemonFightEvent`.
- `PokemonSelectionEvent` is associated with a new type of pause menu that it should display at its start (`PokemonSelectionMenu`, which plays a similar role to the `ICMonFight` menu for the `PokemonFightEvent` event).
- Upon completion, `PokemonSelectionEvent` causes the execution of a new type of action (`AfterPokemonSelectionFightAction`), which is responsible for creating the `PokemonFightEvent` event and sending the game suspension message to initiate the battle. In other words, the processing related to the `PokemonFightEvent` event is done in the `perform` method of `AfterPokemonSelectionFightAction` and no longer directly in the method of interaction between the character and the Pokemon.
- `PokemonSelectionMenu` can be coded very similarly to `ICMonFightActionSelectionGraphics`: instead of displaying text as selection items, it will display images associated with selectable Pokemon.

For example, the instruction on line 73 in `ICMonFightActionSelectionGraphics` will look something like this in `PokemonSelectionMenu`:

```
var spriteName = "pokemon/"+ pokemons[currentChoice + 1].properties().name();
var scale  = CAMERA_SCALE_FACTOR;
var image  = new ImageGraphics(ResourcePath.getSprite(spriteName), scale/2,
   scale/2);
image.setAlpha(.6f);
selectors[2] = new GraphicsEntity(new Vector(scale / 3 + 3f, scale / 2 - 4f),
                                  image);
```

Be careful about how the different classes involved interact to avoid breaking encapsulation.

4.7.1 Task

You are required to code the concepts described above in accordance with the given specifications and constraints.

Ensure that your program behaves as before, but now allows you to graphically select the character's Pokémon to engage in a battle.

The `ICMon` game to be submitted at the end of the project should allow the character to:

- Engage in a battle with an encountered Pokémon
- Select, through a graphical menu, the action to be performed by the chosen Pokémon
- Be able to interrupt the battle (here, systematically with the escape action).

The `ICMon` game, whose behavior is described above, is to be submitted at the end of the project.

5 Complete Scenario (Event Sequences, Step 4)

The goal of this final step is to finalize a complete game scenario involving a sequence of events. Two new characters will make their entrance: Gary, the main character's iconic enemy, and his grandfather, Professor Oak. The main character must now acquire its Pokemon s (instead of receiving them when it is created). This will enable him to battle Gary.

Start by commenting out the instructions giving Pokemons to the character in his constructor.

The scenario that must have been implemented at the end of this stage is as follows:

- The character appears in his house, where Gary is also; he can't fight him without Pokemon.
- He must interact with Professor Oak, who will give him his first Pokemon.
- Once the interaction with Professor Oak has taken place, the ball appears in the water and the character must also pick it up if he wants to battle Gary.
- When you return home, you can then battle Gary with the Pokemon you've received.
- Once this battle is over, an end-of-game event can begin.

A shop assistant can also guide the character to Professor Oak.

To make it easier to model this sequence of events, the concept of event sequences should be introduced, simplifying the writing of the `update` method in `ICMon`.

5.1 Interactions with Professor Oak

5.1.1 The character's house and Professor Oak

Start by introducing a new area named `House` with the label *`"house"`*.

A door allows passage from this area to `Town` (arrival coordinates: `(7,26)`, door coordinates: `(3,1)`, `(4,1)`);

A door allows passage from `Town` to `House` (arrival coordinates: `(2,2)`, door coordinates: `(7,27)`);

Ensure that the main character now appears in their house.

Next, introduce Professor Oak (`ProfOak`), who is a kind of drawable `NPCActor` using the sprite with the name *`"actors/oak"`* [5]. The latter will be created in the laboratory area (`Lab`) at position `(11,17)`, for example.

5.1.2 Chained Events

To simplify the implementation of a game scenario, introduce the new concept of `ICMonChainedEvent`, which is a kind of `ICMonEvent` that allows *chaining events*.

This can be implemented using a constructor that takes as parameters (among others):

[5] Here, we assume that there can be several Professor Oaks (and several Garys) :-) which opens up interesting perspectives for "cloning" in the extensions. Otherwise, they would have to be made what are called "singletons" (to be discovered in the second semester).

Figure 14: Interaction with Professor Oak

- the event with which to start the sequence (the initial event, an `ICMonEvent`):
- an indeterminate-sized set, let's call it `chain`, of events that should be chained in sequence after this initial event (an ellipsis is a good idea here).

The role of the constructor will be to ensure that:

- starting the `ICMonChainedEvent` implies starting the initial event (consider using `StartEventAction`);
- its completion implies starting the first event in the `chain` set;
- then the completion of each of the events in `chain` implies starting the next one (in sequence);
- and the completion of the last event in `chain` implies the completion of `this`. You can create a `CompleteEventAction`, for example, whose `perform` method completes a given event for this purpose.

The `update` method of an `ICMonChainedEvent` does not need any particular redefinition.

To test this new concept, introduce two new events characterized by a `Dialog`, for example:

- `FirstInteractionWithProfOakEvent`, which ensures that the character opens this dialogue when interacting at a distance with `ProfOak` (and this dialogue is not completed). Figure 14 shows what is expected. In the provided resource files, this dialogue is named *`"first_interaction_with_prof_oak"`*. It is now at this moment that the character will receive its first `Bulbizarre`. The `FirstInteractionWithProfOakEvent` event ends when the dialogue is completed (remember that it has already been ensured that the space bar allows ending a dialogue).

- `IntroductionEvent`, which ensures that the character opens this dialogue at the start of the event (for example, the constructor of the event indicates with `onStart` that the action of opening the

Figure 15: Introduction Dialogue

dialogue with the character must be performed). In the provided resource files, this dialogue is named `"welcome_to_icmon"`. Figure 15 shows what is expected. The `IntroductionEvent` event ends when the dialogue is completed.

The `events` method Finally, in `ICMon`, create an `events` method which will be responsible for creating all the events in the game as a single chained event corresponding to the sequence: `IntroductionEvent`, FirstInteractionWithProfOakEvent, `"ball collection"`, andEndOfTheGameEvent. `The`events' method will also be responsible for starting this chained event. As has been the case up to now, the "ball collection" event involves registering the ball in the current area when it starts.

You are free to maintain the presence of `LogAction`s and adapt their content or introduce new ones to follow the chaining of your events.

In the `begin` method of `ICMon`, the events created up to now have no real reason to exist. **Comment on them for testing purposes**.
The `begin` method should therefore look like this:

```
if(super.begin(window, fileSystem)) {
    // création des aires et du personnage
    //..
    // Création des événements des étapes antérieures du projet
    // en commentaire:
    /* ....
    //....
    */
   events(..);
   return true;
   }
 return false;
```

What should you pass as a parameter to the `events` method, if it is now responsible for creating the ball and the associated collection event?

If we consider the single chained event introduced so far as the main quest of the game, how would you code the `events` method to introduce parallel scenarios to this quest?

5.1.3 Task

You are asked to code the concepts described earlier in accordance with the given specifications and constraints. Check that the sequence of events can only be in the correct order, i.e. that:

- the game begins with the introductory dialogue
- once this dialogue has been completed, the character can interact with Professor Oak
- the ball only appears and can only be collected once the interaction with Prof. Oak has been completed. Oak.
- Once the ball has been collected, the game starts the `EndOfTheGameEvent`. It is possible to ensure this by interacting with an `ICShopAssistant` gives rise to the end of game dialogue as allowed by the `EndOfTheGameEvent`:

```
I heard that you were able to implement this step successfully. Congrats!
```

5.2 A little bit of guidance

To guide the character and help them follow the planned scenario, the task now is to find an assistant in a less aquatic position than in previous steps.

Introduce the area `Shop`, named *`"shop"`*.

A door allows passage from this area to `Town` (arrival coordinates: `(25,19)`, door coordinates: `(3,1)`, `(4,1)`).

A door allows passage from `Town` to `House` (arrival coordinates: `(3,2)`, door coordinates: `(25,20)`).

Register an `ICShopAssistant` in this area at coordinates `(8,8)`.

During the `FirstInteractionWithProfOakEvent` event, a remote interaction of the character with the `ICShopAssistant` will cause the opening of the dialogue *`"first_interaction_with_oak_event_icshopassistant"`*, where the assistant will advise the character to go find Professor Oak in his laboratory.

5.2.1 Task

You are asked to code the concepts described earlier in accordance with the given specifications and constraints. Check that when the character interacts with the shop assistant (or the one in the water):

- if the interaction with the teacher has not yet taken place, a dialogue takes place in which the assistant advises the character to go and see Professor Oak in his Laboratory;
- otherwise, if the ball has not yet been collected, the dialogue will be the one coded in the previous steps (feel free to modify the text to advise the character to collect the ball):

```
This is an interaction between the player and the assistant based on events !
```

- otherwise, it's the end-of-game dialogue that takes place:

```
I heard that you were able to implement this step successfully.
   Congratulations!
```

The dialogues thus adapt to the events in progress.

5.3 Finalization

To finish this long journey, introduce `Garry`, an `NPCActor`, which implements the interface `ICMonFightableActor` and is drawn using the sprite *`"actors/garry"`*. `Garry` has his own collection of `Pokemon`.

To simplify, this collection will only contain a `Nidoqueen` transmitted "hard-coded" when constructing `Garry`, and he will always fight with it. `Garry` will be registered in `House` (at position `(1,3)` for example).

Then, modify your code so that the scenario includes a `FirstInteractionWithGarryEvent` event preceding the end-of-game event. During this event, when the character comes into contact with `Garry`, they enter into a battle with Garry's Pokemon.

> Make sure that the opponent's Pokemon, which was artificially constructed until now, is now Garry's Pokemon.

5.3.1 Adapting `PokemonFightEvent`

Until now, the `PokemonFightEvent` was created in such a way that when it was completed, the opponent encountered disappeared. You are asked to change this so that this disappearance is no longer necessarily unconditional, but depends on a condition specific to the opponent: `Gary` must disappear if his `Pokemon` runs out of hit points, and the `Pokemon` must disappear unconditionally (as it did until now).

5.3.2 Task

You are asked to code the concepts described earlier in accordance with the given specifications and constraints. Check that the game behaves as before, but that:

- The end-of-game event now only launches after a battle has taken place between `Gary` and the character (the nature of the dialogues with the assistants will allow you to check this).
- Gary disappears if his Pokemon is defeated.
- There is no more interaction between the character and the assistants between the time when the character has collected the ball but not yet fought `Gary`.

The `ICMon` game, whose behavior is described above, is to be submitted at the end of the project.

You have now completed the compulsory part of the project. Well done! If you still have some time and energy left, you can now add some free extensions. Some suggestions are given in the next section.

6 Extensions (step 5)

To earn bonus points (which can compensate for any penalty on the compulsory part) or to take part in the competition, you can code a few freely chosen extensions. At most 20 points will be counted, so coding a lot of extensions to compensate for the weaknesses of earlier games is not an option.

Implementation is free, with very little guidance. Only a few suggestions are given below. Don't hesitate to ask us to evaluate the scale of your extensions if you decide to go in a specific direction. A small bonus may also be awarded if you demonstrate inventiveness in game design.

You can code your extensions in the existing game, but it's **imperative to preserve the mandatory functionality and testability of the requested scenario**. For example, the instructions in the `events` method in `ICMon` can be grouped together in a `story1` method, the call to which can be commented out. You can then add your own scenarios using other `story..` methods.

Alternatively, you can create a new game, `ICMonExtension`, using the logistics you have set up in the previous steps.

You must **carefully document** how to play your extension in your `README`. In particular, we need to know which controls to use and with which effects, without having to read your code. Here i s an example of a corresponding (partial) 'README' that explains how to play a game:

- example of a `README.md` file from an old project:README.md

You are expected to choose a few extensions and code them all the way through (or almost all the way through). The idea is not to start coding lots of small, inconsistent and unfinished extensions in order to collect the necessary points ;-). An estimate of the number of points associated with extensions is given. It may be worth more if a greater effort than expected is dedicated.

6.1 Extension tracks

In fact, the game you have coded can be enriched as you wish.

6.2 Complexing the opponent

(number of points depending on effort made)

`Gary` can take advantage of having several `Pokemon` and develop strategies to select which one to use in battle.

6.3 New events and scenarios

(number of points depending on effort made)

Use your imagination here. Scenarios may involve collecting `Pokemon` in the areas

6.4 New actors

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

- of course all kinds of new 'Pokemon' can be envisaged with specific characteristics and fighting methods;(~2+ depending on effort)
- animations : instead of representing actors with a single `Sprite`, it is possible instead to associate them with an animation that would be a sequence of `Sprite` displayed in turn to give an illusion of movement; the model offers a `Animation` class that you can exploit for this;((~2 to 3pts)

- new items to collect, or even a resource system: gold, silver, wood, food, treatment doses, etc.(~2 to 5pts)
- various actors that can be used as signals: orbs, torches, pressure plates, levers ;(~2 to 5 pts)
- advanced signals for challenges (oscillators, signals with delay): an oscillator is a signal whose intensity varies over time; (~4pts/signal)
- all sorts of actors with specific methods of displacements and behaviors (friendly and hostile); (~2 to 5 pts/character, up to 10 points if a complex behavior is implemented)
- all kinds of Pokemon, with specific actions options
- create a follower character like Red's Pikachu in Pokémon yellow ;(~3pts)
- create one or more scenario events triggered by signals. For example, an area type could be linked to a signal which, once "switched on", would trigger the appearance of a character. This character in turn would give an object or instruction to solve a quest of a challenge. We can also imagine areas with a variable number of enemies, where an event (such as staying too long in the area) doubles the number of enemies, etc. (the number of points depends on the complexity of the addition).
- make behaviors more complex (e.g. give characters periods of immunity);(~2pts and more depending of the complexity)
- new cell types with appropriate behaviors (water, ice, fire, etc.) ;(~2pts/cell)
- add a shadow or reflection to the player and some actors ;(~3pts)
- add new advanced controls (interactions, actions, moves, etc.) ;(~2pts/control)
- random events (scenery, sigals, etc.) ;(~4pts)
- etc.

6.4.1 Break and end of game (~2 to 4pts)

The notion of area can be used to introduce game pausing. At the player's request, the game can switch to pause mode and then back to game mode. You can also introduce end-of-game management (if the character has reached an objective or has been beaten, for example).

Generally speaking, you can let your imagination run wild and try out your own ideas. If you come up with an original idea that you feel differs in spirit from what is suggested, and you wish to implement it for the rendering or the contest (see below), you must have it validated before continuing (by sending an e-mail to CS107@epfl.ch).

Be careful, however, not to spend too much time on the project to the detriment of other branches!

6.5 Validation of step 5

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

6.6 Competition

Those of you who have completed the project with a particular effort on the final result (interesting gameplay, visual effects, interesting/original extensions etc.) can compete for the «Best CS107 Game» award.[6].

If you'd like to enter, you'll need to send us by **21.12 at 18:00** a small "application file" by e-mail to **cs107@epfl.ch**. This should include a description of your game and the extensions you've incorporated (2-3 pages in .pdf format, with a few screenshots highlighting your additions).

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

7 About the graphical ressources

The source of the images used in this project is https://bulbapedia.bulbagarden.net/wiki/. All these images are under "fair use" copyrighting, meaning that we can use them for education with no commercial aim : https://fr.wikipedia.org/wiki/Fair_use.

[6]We've planned a little «Wall of Fame» on the course web page and a small symbolic award :-)