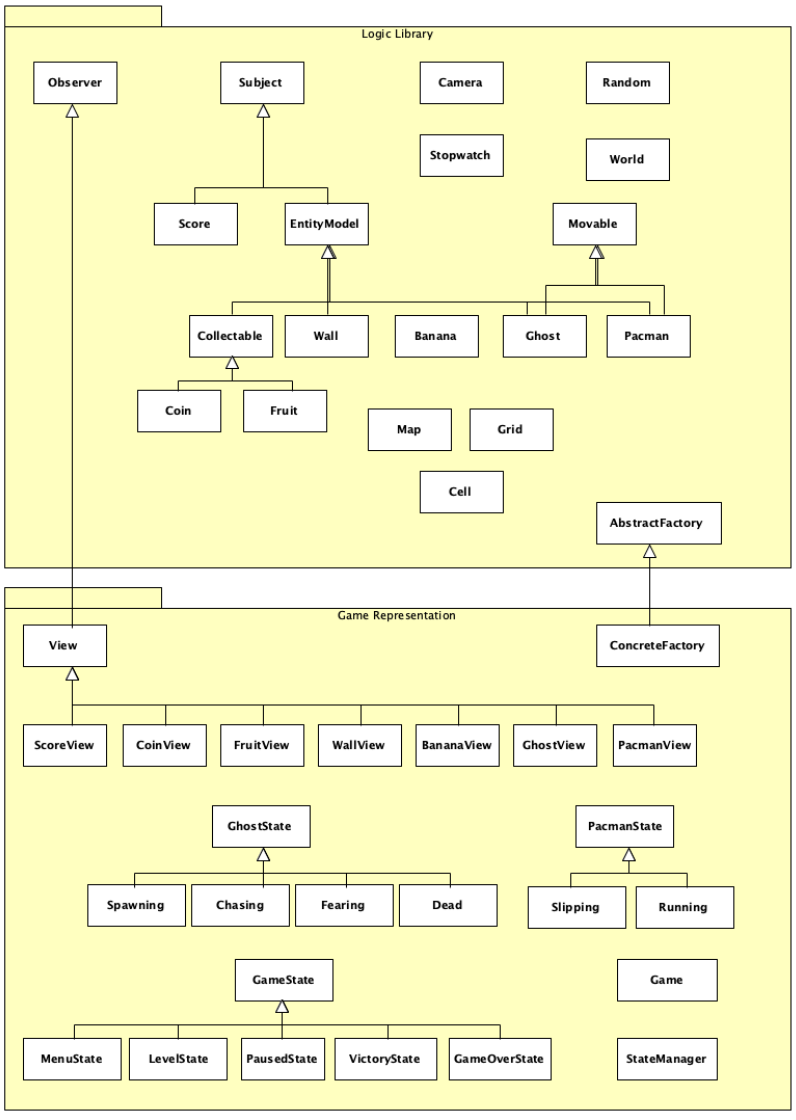
Erba Irene - Extension report

Since I've changed a lot of things from the last version of the project, I found necessary to start from the beginning, incorporating the extension details in the topics of the last version if needed.

The big picture

I implemented the project using the structured provided in the first assignment, with few variations and additions from the extension requirements. Here's how the structure looks like:



I list here the variations from the suggested structure:

- `GhostState` and `PacmanState` added, with their relative states.
- Interface `Movable` added, to handle specific behaviour and properties of objects that can move such as `Pacman` and `Ghost`. They inherit from it.

- Specific classes for maze logical representation and manipulation: `Map`, that encapsulates the `Grid`, which collects a set of `Cell` objects.
- `Banana` class added. It's just another kind of entity, not a collectable, with its own view.
- `Score` class is seen as a `Subject` and doesn't inherit from `Observer` class, as it's been previously suggested. Like the others `EntityModel` objects, it has its own View (`ScoreView`) which inherits from `Observer` class and it's updated when changes notifications arrive.

Design patterns usage

Model-View-Controller (MVC)

As required, MVC pattern is adopted. As already seen above, the project has a Logic and a Representation directory, responsible for the graphic rendering (I didn't succeeded in importing Logic as library though).

Entities have both a *model* and *view*. At the creation of each Entity, both the correct model and view are created and Model is attached to the View (it holds a pointer to it). In case of the entities it's responsibility of the `EntityFactory`; in case of the score there's a specific function in `State_Menu` , since it's not an entity.

Every change to the Model state is handled in the EntityModel first and than it's notified thanks to the observer pattern, which updates the view.

The changes to the objects are reflected also in the `Grid` object, which keeps track of the entities present in the maze and their positions (and it's encapsulated by the `Map` class).

Everything is handled by the World class, which behaves as controller of entities.

Observer

As mentioned, Observer pattern is been used to reflects changes of models to their views.

View inherit from Observer, which contains the methods to notify the changes to the objects (such as UpdateView), and EntityModel inherit from Subject, which contains the list of all observed objects.

`EntityModel` base class implements the `notify(message)` function, to trigger the correct set of observed object to change (e.g. if the message is `"coin taken"`, just the observer of the score will be notified).

Abstract Factory

As required, the Abstract Factory is been used to handle correctly the entity creation, since it involves few steps specific for each `Subject` (not just entities, since also the score is handled this way).

It also ensures that just an instance of a specific entity is created each time. It creates the model to attach to its correct view. Everything is handled with shared pointers and their lifetime is limited to the Word lifetime, which collects them in a vector and it's responsible for their creation and behaviour handling.

The Abstract Factory interface is part of the Logic library.

Singleton

There are few singletons in the project. They've been added when it wasn't necessary to create classes multiple times (or even desired, in case of the problem of preserving score data through multiple levels) and it was necessary to have a global point access to them. In fact, most of them are utilities classes, whose lifetime is considered more acceptable to be the whole program duration. The classes are:

- `Camera` - Converter from pixel to logic dimensions and viceversa.
- `Random` - Contains functions to randomly generate directions or position on the Map.
- `StopWatch` - Keeps track of the time that passes by from the beginning of the Game.
- `ResourcesManager` - This class is responsible for loading the textures from file (there's one for the entities and one for the walls) and cropping the correct sprite for each element. it's called in the view class of each entity in the Setup function. It consists of a parameter (a vector) containing the positions of the different kinds of sprite on the texture and the methods `CropSpriteTexture(SpriteType spriteType, Direction spriteDirection)` (that crops the sprites from the first texture and considering also the direction that is pointed to, in case of movables like

ghosts and Pacman) and `CropWallTexture(std::shared_ptr<Grid> i_map, int a, int b)` (that crops the wall sprites considering also their position on the grid)

- `SharedContext` - I introduced this class in order to have a mediator between the classes that needed an information that couldn't retrieve from their parameters, such as the Entities that needs to update their state on the `Grid`. It hence contains pointers to `World`, `Grid` and `Score` class. Since they don't always consists in the same instance for all the duration of the program, they don't need to be singleton themselves. In Score case, the status needs to be preserved through multiple levels, but at a certain point needs to be destroyed (when the user does Game Over, for example). For this reason a new one is set up in the SharedContext when needed.

State

GameState

The state is handled through a logical implementation of a finite state machine with a stack. The `StateManager` is not responsible for deciding when applying the switch: it's each state's responsibility. However, the changes (popping from and pushing new states in the stack) are handled by this class.

`State_Menu` **is** responsible for displaying the main menu graphic, allowing the user to choose among the available mazes, and also creating the score object, since it needs to displaying the top scores. It also creates the Map when the user select the maze and set the parameter of the SharedContext accordingly.

`State_Level` contains the pointer to the world and contains the update function that states at which frame the world needs to be updated. It also checks if pacman is still alive or it has eaten all the collectables, triggering the switch to GameOver state or Victory state.

`State_Paused` doesn't do anything, except for suspending the game. Pressing the Continue button makes you reentering in the game, and the Menu button makes you go back to the State Menu, destroying the current Level.

`State_GameOver, State_Victory` are triggered in `Level_State`, as mentioned.

PacmanState, GhostState

The implementation of Pacman and Ghost states is similar to the Game one, with the only difference that `Pacman` and `Ghost` classes act as their own state managers, holding the stack for the states and the other push and pop functions.

> In GhostState I thought it was necessary to add a `GhostState_Dead` class to handle the coming back to the spawn point and the related change of skin of the ghost. Such behaviour is not visible though, since I faced problems in implementing the ghost movement towards such a target. However, the ghost passes through this state before entering in Spawning mode.

The `update()` function present in both - and that is called by the world in `updateWorld()` through `move_pacman()` and `move_ghosts()` - consists in an updating function of the current state (the one at the top of the stack). States are hence responsible for the calling the correct functions in `Ghost` and `Pacman` and guarantee the specific movement for that state and deciding when switching state (except in 2 cases: from Running to Slipping mode for Pacman and from Chasing to Fearing for Ghost, since they happens as result of a collision and hence handled in the World class)

Strategy

I used the strategy pattern to assign the correct target to which the ghost has to point to in its movement. It consists in a `TargetingStrategy` interface and a set of concrete strategies classes, one specific for each state (`SpawningStrategy, ChasingStrategy, FearingStrategy, DeadStrategy`). All the `calculateStrategy` functions of the strategies take the same inputs, but give back a different output for each one.

Visitor

This pattern is been used to reconstruct the shortest path from the ghost to his target and it's been built specific for the Grid class. It consists in an abstract interface Visitor implemented by the concrete `ShortestPathVisitor` class.

Since the grid is represented as a vector of cells (which contain a method accept to be visited by the Visitor object), Visitor pattern can explore them and collect data about the shortest path on its way, using the A* algorithm.

Extension topics

Dynamic maps

Loading the maze from an extern file was a goal already reached in the past version of the project.

I added in the `State_Menu` the possibility to choose among 2 different mazes and modified the `Map` class in order to accept an index at the creation, which is be used to select the correct file; `Global.h` contains an array of filenames (`const std::array<std::string, 2> MAP_MAZE_PATHS = {"Logic/Map/Maze1.txt", "Logic/Map/Maze2.txt"};`) , so that a given index corresponds to an element in the vector.

It's possible to add new maps by modifying such file adding a new element in the array of filenames and adding the new map in the Maps folder. The changes to Menu graphic are automatic.

However, the size of the new map must not be different from the predefined size (21×21).

The format I used for the extern file is my own.

The maze creation is responsibility of the `Map` class, which is responsible of loading the maze from the file and convert the format from the file in a grid of enum types that will be used by the world to create the entities.

It encapsulates also the `Grid` class, which is used to keep track of the position changes of the models in the grid and it's used to calculate the shortest path among them by the `ShortestPathVisitor`.

Ghost Setup mode

I succeeded in the implementation of the Spawning mode (`GhostState_Spawning`) in order to make the ghosts escape from their starting point, but just with the discrete movement. As already mentioned before, the spawning point is handled like the other targets (using the `TargetingStrategy` class) and the path to it is calculated through the `ShortestPathVisitor` class.

Banana slips

Its creation and spawning timing is handled by the world, which contains in the loop a timer for it. Just if no other bananas are spawned at that moment and if a certain time is passed by (10 sec), a new Banana object is created and spawned. When the collision function detect that Pacman slipped, it removes the entity from the list of entities and reset the timer. At the same time, the `slip()` function of `Pacman` is called and Pacman enters in the `State_Slipping`, that freezes it for few seconds. As soon as the time is passed by, `State_Slipping` calls the switch to the `State_Running`.

GameoverState

Also the GameOver state was already an achievement of the past version: however, I've adjusted the logical transitions of the finite state machine and added a better implementation of it.