ECS7014P - Lab Exercises

An Entity-Component Architecture

Lab Weeks 7-8 - Assignment 1-C (10%)

Lab Objectives

The objective of this lab is to improve the previous implementation of our Mini-game by programming an Entity-Component (EC) architecture. In this lab you will also use the **Component** pattern.

In particular, in this lab you will:

- Modify the fully functioning mini-game so its code is organized appropriately under the Entity-Component architecture.
- You'll do this by implementing the *Component* pattern and applying it to the previous codebase you implemented in the previous lab.
- You'll continue getting experience with elements of the C++ language, including the importance of forward declaration vs include directives.

The code you produce by completing this lab corresponds to **Assignment 1-C**. Assignments 1-D will also build on the outcome of this lab, so by completing these exercises you will also be indirectly working towards the final piece of assignment 1. Please read the instruction assignment to understand how this fits in the submission. You can also find some help for preparing this submission at the end of this document. Remember that you have the opportunity of submitting this assignment for a feedback round by **17**th **March**. No late submissions for the feedback round are allowed.

Independently from you submitting a draft submission for feedback, you must submit the final version of the assignment by **7**th **April**. The draft submission will **never** be evaluated as a final submission.

1 Before you start

In this case, you don't need to create any new Visual Studio 2022 solution. Keep using the same one you used in the previous lab, committing your changes to your repository. This lab doesn't have any extra materials provided beyond these instructions.

You can **only** start this lab once you've completed the previous one.

2 An overview: what are we doing?

The objective of this lab is to implement the Component pattern on the previous code of the mini game. As shown in Lecture 4 (The Entity Component System) during the explanation of the Component pattern, this exercise consists of **encapsulating** data and logic from the Entities of our game into their own component classes. The resultant codebase after completing this exercise will be more modular and flexible, easier to maintain and expand. One characteristic you'll appreciate of the final state of the code is that you'll have more classes in your project, but they will be smaller and more focused on specific pieces of functionality.

The instructions below describe in detail how to do this for the following components: *Input*, *Time to Live*, *Health* and *Position and Velocity*. By following these instructions, you'll complete the above components and opt to up to 60% of the mark for Assignment 1-C. You will also need to complete these components in order to carry out the next lab (for Assignment 1-D).

The instructions also provide information about three other components that you can implement in order to get a higher mark for this assignment: *Graphics*, *Collider* and *PlayerState*. These instructions are not as detailed, but you will have the experience at this point to attempt this on your own.

3 Building our first Component: Time To Live

To keep our code organized, create a new folder for our components. In fact, two: one for header files (include/**components/**) and another one for the sources (source/**components/**).

The Time To Live Component will be used by the only entity in the game that gets destroyed after several game ticks: Fire. Follow these steps:

- Create a new file, TTLComponent.h, in include/components/. This component is so simple that we won't need to add a source (.cpp) file.
- In this file, declare a class *TTLComponent* and give it the following members:
  - A private integer member, called *ttl*, that will determine the time-to-live value. When this gets to 0, the component will indicate that the entity that owns it should be destroyed.
  - A public TTLComponent constructor that receives an integer, which must be used to initailize (in the initializers list) the *ttl* member variable.
  - An *update()* function, which returns *void* and receives no parameters. This function simply subtracts 1 from the *ttl* member variable every time is called. For consistency, only subtract 1 if the value of *ttl* is greater than 0.
  - A member function that returns the value of *ttl* (call it, for instance, *getTTL()*).

Our next step is to add the component to the Fire object. Do the following:

- Add an include directive (for TTLComponent.h) to Fire.h, so Fire has access to the members of the TTLComponent class.
- Change the type of the *ttl* variable from an integer to a *unique pointer* to TTLComponent.
- The function *Fire::getTTL()* must return the *ttl* value from the TTLComponent. Return the value using the function *TTLComponent::getTTL()*.
- In Fire.cpp, remove the ttl initialization from the Fire constructor initializers list. Instead, we have to initialize the *ttl* unique pointer in the body of the constructor of the Fire class. Do this now. Note that the constructor of the TTLComponent must receive the constant *startTimeToLive*, which is already defined in the Fire class.
- In *Fire::update()*, rather than subtract one from the (now in-existent) *ttl* integer member, call the *update* method of the TTLComponent to reduce the time to live in the component.

- After this, check the value of the time to leave from the component, to verify if the entity should be deleted. Only if the TTL value received from the component is 0, the variable *deleted* must be set to `true` in this function.

This should be all needed in your code to extract the TTL functionality to a component. Build and play-test your code, to make sure it all works as before.

4 A second component: Health

Let's now encapsulate the functionality for the health system. Again, this will only be used by one entity in our example: Player. As in the case of the TTL, however, having this functionality extracted from the respective entity classes makes the code more modular and easy to reuse. Do the following:

- Create a new file, HealthComponent.h, in include/components/. As for TTL, we won't need a source (.cpp) file for this component, it's also pretty simple.
- The next step is to declare the class *HealthComponent* in this new file. Add the following members next:
  - Two protected variables of type *int*. One indicates the current health, and the other the maximum possible health.
  - A public constructor that receives two *int* arguments, one for the starting health and another one for the maximum health. These should be used to initialize the current and maximum health member variables, respectively, in the initializers list.
  - A member function (*getHealth()*) that returns the current value of the health of this component.
  - A member function *changeHealth()* that receives an integer parameter. In this function, the current health value must be modified by adding the variable received by parameter (i.e. it will add or subtract a value, if the argument is a negative value). The code needs to always make sure that the resultant value is between 0 and the maximum health value[1].

Now, we'll add the Health component to the Player class:

- Add a **shared** pointer to a HealthComponent in the Player class (declare it in Player.h and initialize in the constructor, as done in the previous task). You'll need to also add an include directive to this file to make the contents of HealthComponent.h accessible.
- Remove the health member variable from the Player class. This will make the function *int getHealth()* invalid. Change this function so it returns, instead, the shared pointer to the health component you've just added to the class. This function can no longer be *const* (as we'll return the component to then be modified) and should have a more descriptive name, as it retrieves a component now (i.e. getHealthComp()).
- Remove also the function *Player::addHealth()*, from both the header and the source Player files.
- In Player.cpp, create the pointer to the HealthComponent object in the constructor of Player. Use the Player const variables *startingHealth* and *maxHealth* to initialize this component.
- Remove the health initialization of the initializers list of the constructor.
- Finally, go to Game.cpp, to the collisions code. Concretely, where a collision with a Potion object is detected. In order to modify the health of the player, we must now retrieve the health component from the player and call the *HealthComponent::changeHealth()* method, instead of calling the player's addHealth method. Make sure you also update the console printout (if you are still printing it) so the current health value is still showing upon collision.

This should complete the encapsulation of the health component. Build and play-test to make sure all works as intended.

[1] This is the behaviour implemented at the moment in *Player::addHealth()*, which you can consult as well.

5 Adding the Input Component

We'll create now a component for the input functionality. This is still a simple component, just slightly more complex that the previous two. Note, for instance, that this will be a combination of two patterns: our *Command* pattern (which handles the input) and the *Component* pattern (which we are about to implement). We are implementing the Component pattern for the player only[2]. We'll encapsulate our data and logic from the entity class (Player) to this component.

Perform the following steps:

- Create a new file, InputComponent.h, in the include/components/ directory.
- Declare the *InputComponent* as a **pure virtual** class in this header file. Add the following to the declaration of this class:
  - A virtual destructor for the class.
  - A pure virtual function named *update*. It must return *void* and receive a reference to a game object (*Game&*) by parameter.
  - In this case, we are **not** adding an include directive to this class, as it would create a circular dependency with other files. Instead, do a forward declaration to the class Game in this file. [3]
- Then, in the same file (InputComponent.h), declare another class (*PlayerInputComponent*) that inherits from *InputComponent*. Include the following declarations (only declare it here, we'll add the definitions in a source file) in the subclass:
  - A default constructor for PlayerInputComponent.
  - A (public) function *update* that overrides the function of the same name on the base class.
  - A (private) member **unique** pointer to a PlayerInputHandler. You'll see that you also need to forward declare this class. This pointer is going to substitute our old Player::input, in order to encapsulate it here. Here is an example of how are we *moving* data from the Entity class to the components.
- The PlayerInputComponent functions need to be defined in a source file. Create a file InputComponent.cpp in the source/components/ directory. Then, in this file:
  - Include the header file InputComponent.h.
  - Define the constructor of this class. This constructor must initialize the PlayerInputHandler unique pointer.
  - Define the component update function. What we need to do here is to bring the code we have in Player::handleInput() so it's this component, and not the Player, who deals with the input. This requires (see Player::handleInput() if you need remembering how to do this):
    1. Set the velocity of the player to $0, 0$ at the start of the function. Note that you can access the player by using the function Game::getPlayer().
    2. Retrieve the vector of commands (i.e., calling the function *PlayerInputHandler::handlePlayerInput()* from the *input* member variable.
    3. Execute all commands received.
  - Note that you'll also need to include a certain number of headers in this source file for this code to build properly.

At this point, the PlayerInputComponent class has all the functionality needed to capture the input for the player, but we need to set up Player so it uses this component. Let's go to our Player class and do the following:

- Remove the pointer to the PlayerInputHander in Player.h, and change it for a unique pointer to an InputComponent (you can still call it *input*). You'll also need to change the forward declaration at the top of the class to `class InputComponent;`.

[2] We'll leave the Game input handler as is. This could also be a component for the object Game, but we'll keep it simple for now.

[3] As a rule, When possible, try to *forward declare* classes in header files, and include headers in the source files.

- In Player.cpp, in the default constructor, initialize the input to the correct type now. You'll need to change the include directive as well so you include the InputComponent header file.
- The function Player::handleInput() must now only call the update method of the input component. The rest of the code of this method can be deleted, as it's the component who's taking care of all this functionality.

You should be able now to build the solution and play-test the game. If everything was implemented correctly, you should see the same behaviour as when the input handler was placed in Player.

6 Position and Velocity Components

The components we've taken care of so far are the easiest ones to extract, as the code for handling TTL, health and input is relatively independent from the rest of the codebase. The rest of the code is much more intertwined and hard to separate in components. Two notes about this:

1. This *characteristic* of the code we are modifying does not only make it (a bit) challenging to modify to turn it into an EC architecture. It actually makes it (a lot) harder to expand and maintain if we were to continue developing the game with more features without following the EC (or ECS) architecture. This doesn't mean that the code we wrote before today was *wrong*. It was a decent implementation for a simple game, but going beyond that would turn the code into a difficult, monolithic, codebase to work with.
2. Obviously, if you were to implement a new game, you'd start directly using an EC (or an Entity-Component-System, as we'll see in the next assignment) architecture. You wouldn't build a monolithic code to *then* turn it into an EC/ECS. The objective of these exercises is to show you the difference between the distinct game architectures, as well as to illustrate how different the code is under different paradigms.

Okay, after this quick philosophical detour, let's get back to it. It's time to encapsulate position and velocity from our entities to respective components.

6.1 Position Component

Let's start with the Position Component:

- Create a new header file, PositionComponent.h, in the appropriate components directory. Declare the PositionComponent class and the following members. Given the simplicity of this class, you'll be able to declare and define everything in this header file:
  - A private member variable, *position*, of type Vector2f.
  - A public member function that returns a constant reference to the *position* member variable.
  - A public member function that sets the position. It must receive two float variables, one for x and one for y, which should be set as the x and y of the position vector.

All entities in the game have a position (not all have a velocity, though). Let's now give those entities a Position Component and remove the respective members from the entity classes. In Entity.h:

- Add a forward declaration to the PositionComponent and a unique pointer to a PositionComponent object, as a *protected* member of the class. Naming it *position* is a good idea, still.
- This member is replacing the member *Vector2f position*, so remove the vector now.
- When doing this, you'll get a compilation error because the function:

```
1 const Vector2f& getPosition() const { return position; }
```

  is now returning the wrong type. Turn this in the header file into a declaration, and we'll implement the definition of this method in the source file Entity.cpp.

Now, in Entity.cpp:

- Initialize the unique pointer to the position component in both constructors. Note that you need to remove the position (vector) from the initializers list and add the line that creates the pointer in the body of the constructors.
- The Entity::update method is accessing and modifying the position of the entity as it if were a vector. Make the changes in this function so the position accessed and modified is the one from the component (using the PositionComponent functions you've implemented in the previous point of these instructions). Note that this not only affects the position component itself, but also affects graphics (sprite position) and collision (bounding box position). We'll get back to these further down the line, when building these respective components.
- The function Entity::setPosition also needs amending, in a very similar way to how it was done in Entity::update. Do this now.
- We need to define the function Entity::getPosition() (which we changed to a declaration-only in the header[4]). Implement this function, which must now just redirect the call to the PositionComponent::getPosition method.

The old Entity::position vector was used in all subclasses. Let's do a tour through our classes to make them use the PositionComponent instead:

- In StaticEntities.h, the *init()* method of **both** Log and Potion access position directly to set up the bounding box. Change it so they use the position from the component instead.
- Player::createFire() in Player.cpp, also uses the old position vector to determine the position where fire should be spawned when shooting. Change this as well so it uses the position from the component.

This should be all. Build[5] and play-test to see that everything still works as before.

6.2 Velocity Component

We'll work now on the Velocity Component. Note that **not** all entities have a velocity, so we'll only provide the Velocity Component to the Fire and Player classes, but not to Entity. Do the following:

- Create a new file, VelocityComponent.h (in include/components/), with the declaration of the class *VelocityComponent*. Declare the following members for this class:
  - The velocity component needs two member variables: the velocity direction (a *Vector2f*) and the speed (a *float*). Add them both to the declaration of *VelocityComponent*.
  - Add a public constructor for the *VelocityComponent* that receives a variable of type float, which must be used to initialize the speed member of this class. This argument must be optional, with a default value of $1.f$.
  - Like for the PositionComponent, add two member functions for i) setting the velocity (receiving two float variables for x and y) and ii) returning a constant reference of the velocity vector. This can be declared and defined in the header file.
  - Declare (without definition) an update function that returns *void* and receives two parameters: a reference to an *Entity* object and a float variable for the elapsed time.
- Create a new file, VelocityComponent.cpp (in source/components/), and define the function *VelocityComponent::update(Game &)* in it. Leave the body of this function empty for now.

Let's now bring the new component to the Fire and Player classes. First, for the Player:

- Add a shared pointer to a VelocityComponent object in Player.h (call it *velocity*).

[4] If you're wondering **why** have we done this: we are avoiding having to include another header in the Entity.h file, using a forward declaration instead. But the forward declaration only let us *name* the class, not actually *use* it. We *use* it in the .cpp file, where we can include the header of the PositionComponent with less overhead.

[5] If you have compilation errors, one of the reasons could be that you're missing some include directives in the classes that you've modified.

- Add a function to the Player class that returns this shared pointer from Player. Call this *getVelocityComp()*.
- The values of the velocity are now set via the component, so we can *delete* the *Player::setVelocityX()* and *Player::setVelocityY()* member functions.
- Initialize the shared pointer of VelocityComponent in the Player constructor. Note the playerSpeed is now passed to the VelocityComponent constructor, so we don't need to initialize the speed member variable in the Player's constructor. Remove the line *speed = playerSpeed;* from that function.
- Player::update (in Player.cpp) needs to change now so we access the x and y components of the velocity through the velocity component.
- The same applies to the function Player::positionSprite(), which also sets the velocity of the sprite in its last line.
- Our PlayerInputComponent::update() resets the velocity at the start of the method. We now need to update this, to reset the velocity of the player using the component of the player. From the game, retrieve the player (Game::getPlayer()) and then retrieve the velocity component from the player (using the getVelocityComp() function you've just added), to finally use the setter in the component to reset the velocity to (0, 0). You could implement this in a similar way to this:

```
auto v = game.getPlayer()->getVelocityComp();
v->setVel(0.f, 0.f);
```

- Similar changes are also required in our movement command classes (for moving up, down, left and right). They must change the velocity of the player through the component. Make those changes now.
- Player::update() must now call the Velocity::update() method, passing a reference to itself and the elapsed time. Add this call to the Player::update() function[6]. Something like this:

```
velocity->update(*this, elapsed);
```

- Finally, we need to carry out the update of the velocity. So far, entities that would move around updated their velocity in the Entity::update() call. Let's remove the code from there that changes the position of the entity (not the graphics) and place it to the VelocityComponent::update() function. You will have to use the new functions for setting the position (which use the PositionComponent) to update the position of the entity. Note that this is one component (Velocity) modifying the data of another component (Position).

You should now be able to compile, build and play-test the game. The game should work as before, with the player moving around fine, but you'll see that if you try to shoot the fire will not move. This makes sense: we've removed the code that makes it change position (from Entity::update()) but we still haven't added it to the Fire object. Let's do it now:

- Add a **shared** pointer to *VelocityComponent* to the Fire class (Fire.h) and a function that returns this velocity component pointer (i.e. Fire::getVelocityComp()).
- In Fire.cpp, initialize this shared pointer in the constructor.
- In Fire::update(), add a call to the update() method of the velocity component.

You can try to build and run the code now, but you'll see that the fire still doesn't move. This happens because Entity is **hiding** certain methods from Fire[7]. We need to do certain adjustments to our classes so we can completely remove the velocity functionality from Entity and fully delegate it to the Velocity component:

- Remove the *velocity* (Vector2f) and *speed* (float) member variables from the Entity class. This also cascades down to other members of the class, so you must also remove the functions that access these members: Entity::setVelocity(), Entity::getVelocity() and Entity::getSpeed().

[6] Anywhere. For instance, before the *Entity::update* call.
[7] Concretely, Player::createFire() is setting the velocity *vector* of the Entity, not Fire's velocity component.

- Similarly, we don't need to call Entity::setVelocity() in Game::buildEntityAt<T>() anymore, nor initialize the velocity and speed members in the Entity constructors' initializer lists.
- The function Player::createFire() must now retrieve the velocity component from the newly created fire entity to set its velocity.

Once this is done, you should be able to build run the game again to obtain the normal behaviour, where both player and fire objects move properly.

7 The rest of the components

For the rest of the lab, work through the codebase following the same principles we've seen so far. The suggested components to add to the codebase are listed below. You are recommended to follow a similar procedure for each component as before:

1. Create the class for the new components.
2. Add smart pointers to the classes that should own a component of the given type and decide where and how it should be initialized.
3. Move the data and the logic from the classes that have just received a component to the component themselves. This will require changes through the code so the appropriate functions are called from different objects.
4. Always verify that, after finalizing a component, the game still works as intended.

The following are a set of tips for you to follow when completing this part of the exercise:

- **GraphicsComponent:** the recommendation for this component is to have an interface (i.e. a class where **all** its functions are *pure virtual*) and two subclasses: one for sprite-sheet graphics and another one for simple sprite ones. The former one should take care of all functionality related to playing animations, while the latter must deal with static and simple sprites.

  The Graphics Component must be a member of the Entity (all entities need to be *drawn*) in the form of a shared pointer to the base class. Player objects instantiate this to the sprite sheet subclass, while the rest of the entities will have the other subclass for simple sprites. One possible way of setting this up is to pass the specific GraphicsComponent to the *init* method of the Entity[8], adding a new parameter to this function. The function in Entity could look like this:

```
1 class Entity{
2   // Some code stuff...
3
4 public:
5   virtual void init(const std::string& textureFile, std::shared_ptr<
        GraphicsComponent> gc);
6 }
```

  And the code that calls this function could be something like the following for a Player (in Game::init()):

[8] As we saw in the lectures, this is one possible way of configuring the *Component* pattern.

```
void Game::init(std::vector<std::string> lines) {

  // ...

  // For player:
  player->init("../img/DwarfSpriteSheet_data.txt", std::make_shared<
      SpriteSheetGraphicsComponent>());

  // ...
  // For other types of entity:
  ent->init(filename, std::make_shared<SpriteGraphicsComponent>(itemScale));

  // ...
}
```

Note that all entities will need to update the graphics component so the position of the sprite is updated with the position of the object, and the current animation progresses to the next sprite at the normal rate in the sprite sheet graphics component case. All entities must also delegate the drawing of the sprite / sprite sheet to the components, calling the appropriate function that draws to the window on each render call of the game loop.

- **ColliderComponent**: This collider component can be a single class that holds the bounding box of the objects. It should be in charge of checking for collisions with other ColliderComponent objects (which in turn check if their data - their bounding boxes intersect with each other). They must implement an *update* method (to update the position of the bounding box) and a *draw* method (to draw the collider on screen).

  The hardest bit about creating the collider component is how collision management is handled. A good implementation would not give a collider component to all entities, as some may not need it (for instance, our Fire object, which doesn't use it for anything). Only the classes that need to implement a collider (Log, Potion, Player) will hold a shared pointer to a collider component in their classes.

  However, in Game::update(), collisions are checked against entity objects (Entity elements in the Game's entity vector), so the Entity class needs a member virtual function that returns this collider component. Entity must return a null pointer, while the rest of the classes must return the actual member pointer. Given that in Game::update() we won't be able to retrieve the bounding box directly from the entities, we need a function in Player that checks the collision internally (using the collider). Something like this:

```
bool Player::collidesWith(Entity& other)
{
  return colliderComponent->interesects(other.getColliderComponent().get());
}
```

- **PlayerStateComponent**: This is a component that takes care of the logic of the Player. The data that this component handles includes the amount of wood, the variables for attacking/shouting, shooting cooldown, and the relevant constant members related to these variables. This component's update method takes care of (via the graphics component) setting the animation to play, updating the firing cooldown and shooting fire. In essence, this function would implement what *Player::update()* does.

  This component must inherit from a new abstract class LogicComponent, with a declaration like the following:

```
class LogicComponent
{
public:
  virtual void update(Entity* entity, Game* game, float elapsed) = 0;
};
```

The objective of this component is to serve as a base component for all entities in the game that require the implementation of specific logic. An example of this is the logic of the player, implemented in PlayerStateComponent.

With all components implemented, the *input*, *update* and *draw* functions of Player should simply redirect the calls to the components. It may look similar to the code below:

```
void Player::handleInput(Game& game)
{
  input->update(game);
}

void Player::update(Game* game, float elapsed)
{
  state->update(this, game, elapsed);
  velocity->update(*this, elapsed);
  collider->update(*this);
  graphics->update(*this, elapsed);
}

void Player::draw(Window* window)
{
  collider->draw(window);
  graphics->draw(window);
}
```

The functions in Entity should also end simplified to have a code similar to this:

```
void Entity::update(Game* game, float elapsed)
{
  graphics->update(this, elapsed);
}

void Entity::draw(Window* window)
{
  graphics->draw(window);
}
```

As you can see, the code for Player and Entity are quite simplified now. They are wrappers for components, to which they redirect calls within the Update pattern. These components now have the *data* of the entities, as well as their *logic*.

In the next lab, we'll move the logic out of the components to the systems, to implement a full Entity-Component-System architecture. This will not only make the code more modular (as logic will be separate from data), but also will make our entity classes even simpler.

Preparing your assignment 1-C submission

For the assignment submission, you are asked to fill and upload a form to QM+, as you did for the previous assignment. Only one submission per group is necessary. Check the document in QM+ to see what do you need to include in this form, which also includes the marking criteria. As before, the code is "submitted" as a github repository link (at `https://github.qmul.ac.uk/`). We are again asking you to publish a GitHub ***private release***. Note that, once a release has been created, further changes to the code will not be captured by that release. We should already have access to your repository from the previous submission, but double check that the accounts **eex516** and **acw634** do indeed have access to it.