Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2022-1

31 de Marzo de 2022
Actividad Sumativa

Actividad Sumativa 1

OOP2

Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** `Actividades/AS1/`
- **Hora del *push*:** 16:40

Importante: Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add*, *commit*, *push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (***push***), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

Introducción

El aumento de alumnos en el DCC ha provocado que los cursos de computación se queden sin salas para impartir sus clases. Es por esto que el DCC ha entrado en una fase expansionista y ha decidido declararle la guerra a los otros Departamentos de la Escuela de Ingeniería, con el objetivo de controlar las salas del campus para así tener más cupos en sus cursos. Se te ha encargado a ti estudiante de Programación Avanzada simular la guerra con los otros departamentos al estilo del juego Age of Empires 2.

Flujo del programa

Al ejecutar el archivo `main.py` se crean dos instancias de la clase `Reino`, correspondientes a los Departamentos que se enfrentarán. Luego, se inicializa una instancia de la clase `Simulacion`, la cual recibe a ambos reinos y la cantidad de semanas a simular. Una vez instanciada la simulación se da comienzo a la guerra mediante el método `run()`.

Archivos

En el directorio de la actividad encontrarás los siguientes archivos

- `main.py`: Este es el archivo principal del programa. No debes modificarlo
- `simulacion.py`: Contiene a la clase `Simulacion`. No debes modificarlo
- `reino.py`: Contiene a la clase `Reino`. Debes modificarlo
- `unidades.py`: Contiene a la clase `Persona` y a las clases que heredan de esta, las cuales son `Guerrero`, `Mago` y `MagoGuerrero`. Debes modificarlo
- `construcciones.py`: Contiene a la clase `Estructura` y a las clases que heredan de esta, las cuales son `Barracas`, `Muro`, `Catapulta` y `MuroCatapulta`. Debes modificarlo
- `parametros.py`: Contiene a todos los parámetros necesarios para el funcionamiento del programa. No debes modificarlo

Parte 1: Modelación de Unidades

En esta parte, deberás completar las clases `Persona`, `Guerrero`, `Mago` y `MagoGuerrero` las cuales se encuentran en el archivo `unidades.py`. Deberás definirlas correctamente en cuanto a sus atributos, métodos y relaciones. Al final del enunciado hay un diagrama de clases que te puede ayudar a visualizar las relaciones.

Clase Persona

- `class Persona`: Esta clase debes definirla como **abstracta** y representa a todas las personas que serán parte de la guerra. De esta clase heredan las subclases `Guerrero`, `Mago` y `MagoGuerrero`. Debes modificarlo
  - `def __init__(self, xp: int, stamina: int, **kwargs)`: Inicializador de la clase que recibe el `xp` y la `stamina` que tendrá cada `Persona`, ambos como `int`. Se asignan los atributos `self.xp` y `self.stamina` y también se encarga de llamar al método `asignar_parametros()`, el cual deberás completar más adelante. No debes modificarlo
  - `def stamina(self)`: Corresponde a la *property* de la `stamina` de cada `Persona`. Se encarga de verificar si una `Persona` puede revivir una vez que su `stamina` es menor o igual a cero. No debes modificarlo
  - `def revivir(self)`: Al llamar a este método se define mediante una probabilidad si la `Persona` puede revivir una vez que se le acaba su `stamina`. No debes modificarlo
  - `def asignar_parametros(self)`: Debes definir este método y declararlo como **abstracto**, ya que deberá ser implementado en todas las subclases. Será el encargado de asignar un valor a ciertos atributos de las subclases. Debes modificarlo

- `def accion(self)`: Debes definir este método y declararlo como **abstracto**, ya que deberá ser implementado en todas las subclases. Representa la acción de cada clase y retorna un valor dependiendo del tipo de `Persona`. Debes modificarlo
- `def __str__(self)`: Debes definir este método y declararlo como **abstracto**, ya que deberá ser implementado en todas las subclases. Se utiliza para mostrar información relevante de la clase. Debes modificarlo

Clase Guerrero

- `class Guerrero`: Esta clase hereda de la clase `Persona` y representa a los guerreros que serán los encargados de atacar al `Reino` enemigo. Deberás completar la clase para que herede correctamente de la clase `Persona`. Debes modificarlo
  - `def __init__(self, armado: bool, **kwargs)`: Inicializador de la clase que recibe un `bool` indicando si el `Guerrero` está armado, junto a los argumentos de la superclase que serán entregados como argumentos por palabra clave (*kwargs*). Deberás entregar los argumentos correspondientes a la superclase y definir el atributo `self.armado`. Es importante notar que para asignar los parámetros de un `Guerrero` se necesita el valor de `self.armado`. Debes modificarlo
  - `def asignar_parametros(self)`:Debes definir este método, en el cual deberás asignarle un valor al atributo `self.ataque` de cada `Guerrero`. Dicho valor se calcula mediante la siguiente fórmula

    $$ataque = 5 \cdot \pi \cdot xp \cdot \frac{1}{2}$$

    el cual debe ser redondeado a un `int`[1]. En el caso de que el `Guerrero` esté armado, deberás multiplicar por 2 el valor entregado por la fórmula anterior. Debes modificarlo
  - `def accion(self)`: Debes definir este método el cual al ser llamado, el `Guerrero` retornará el ataque que hace al reino enemigo. Debes chequear que la probabilidad[2] sea menor a `PROB_CRITICO_GUERRERO`, si esto sucede debes retornar el ataque del guerrero multiplicado por 1.5 (redondeado a entero). Debes modificarlo
  - `def __str__(self)`: Debes definir este método, el cual se encarga de mostrar información relevante de cada `Guerrero`, indicando si está armado, junto con su ataque. Debes modificarlo
    Por ejemplo:
    La línea 1 del código que se encuentra a continuación muestra el mensaje que deberías mostrar en caso de que este armado, y el mensaje de la segunda línea lo que se debe imprimir si no está armado.

```
1 f'Guerrero Armado con {self.ataque} pts de ataque'
2 f'Guerrero con {self.ataque} pts de ataque'
```

[1] Puedes redondear un número mediante la función `round()` y usar el valor de π mediante `math.pi`
[2] Usando la función `random()` del modulo `random`

Clase Mago

- `class Guerrero`: Esta clase hereda de la clase `Persona` y representa a los magos que serán los encargados de curar al `Reino`. Deberás completar la clase para que herede correctamente de la clase `Persona`. Debes modificarlo

  - `def __init__(self, bendito: bool, **kwargs)`: Inicializador de la clase que recibe un `bool` indicando si el `Mago` está bendito, junto a los argumentos de la superclase que serán entregados como argumentos por palabra clave. Deberás entregar los argumentos correspondientes a la superclase y definir el atributo `self.bendito`. Es importante notar que para asignar los parámetros de un `Mago` se necesita el valor de `self.bendito`. Debes modificarlo

  - `def asignar_parametros(self)`: Debes definir este método, en el cual deberás asignarle un valor al atributo `self.curacion` de cada `Mago`. Dicho valor se calcula mediante la siguiente fórmula

    $$curacion = 1{,}6180 \cdot \pi \cdot xp \cdot \frac{1}{2}$$

    el cual debe ser redondeado a un `int`. En el caso de que el `Mago` esté bendito, deberás multiplicar por 2 el valor entregado por la fórmula anterior. Debes modificarlo

  - `def accion(self)`: Debes definir este método el cual al ser llamado, el `Mago` retornara la curación que hace al reino. Debes chequear que la probabilidad sea menor a `PROB_CRITICO_MAGO`, si esto sucede debes retornar la curación del Mago multiplicado por 1.5 (redondeado a entero) Debes modificarlo

  - `def __str__(self)`: Se encarga de mostrar información relevante de cada `Mago`, indicando si está bendito, junto con su curación. Debes modificarlo
    Por ejemplo:
    La línea 1 del código que se encuentra a continuación muestra el mensaje que deberías mostrar en caso de que este bendito, y el mensaje de la segunda línea lo que se debe imprimir si no está bendito.

```
1 f'Mago BENDITO con {self.curacion} pts de curación'
2 f'Mago con {self.curacion} pts de curación'
```

Clase MagoGuerrero

- `class MagoGuerrero`: Esta clase hereda de las clases `Mago` y `Guerrero`, representa a aquellas personas capaces de atacar y curar a algún `Reino`. Deberás completar la clase para que herede correctamente de la clases `Mago` y `Guerrero`. Debes modificarlo

  - `def __init__(self, **kwargs)`: Inicializador de la clase que recibe los argumentos de las superclases, los que serán entregados como argumentos por palabra clave. Es importante que en esta parte realices un **único** llamado al `__init__()` de las clases padres. De no ser así podrías estar inicializando dos veces la clase `object` y `Persona`, lo cual no queremos que suceda. Debes modificarlo

  - `def asignar_parametros(self)`: Debes definir este método en el cual se le asigna un valor a los atributos `self.curacion` y `self.ataque` del `MagoGuerrero`. Dichos atributos se calculan mediante las mismas fórmulas usadas en las clases `Mago` y `Guerrero`. Por lo que en esta parte simplemente deberás llamar al método `asignar_parametros()` de cada una de las superclases. Debes modificarlo

  - `def accion(self)`:Debes definir este método, el cual al ser llamado el `MagoGuerrero` retornara una tupla con la curación y el ataque que hace en la forma `(ataque, curación)`. Debes

chequear que la probabilidad sea menor a `PROB_CRITICO_MAGO_GUERRERO`, si esto sucede la curación y el ataque deben ser multiplicados por 2. Debes modificarlo

- `def __str__(self)`: Se encarga de mostrar información relevante de cada `MagoGuerrero`, indicando si está bendito y armado, junto con su curación y ataque. No debes modificarlo
  Por ejemplo:

```
f'MagoGuerrero BENDITO y ARMADO con {self.curacion} pts de ' \
f'curacion y {self.ataque} pts de ataque'
```

Parte 2: Modelación de Construcciones

En esta parte, deberás completar las clases `Estructura`, `Barracas`, `Muro`, `Catapulta` y `MuroCatapulta` las cuales se encuentran en el archivo `construcciones.py`. Deberás definirlas correctamente en cuanto a sus herencias, atributos y métodos. Recuerda que al final del enunciado hay un diagrama de clases que te puede ayudar a visualizar las clases junto a sus atributos y métodos.

Clase Estructura

- `class Estructura`: Esta clase debes definirla como **abstracta** y representa a todas las estructuras que serán parte de cada reino. De esta clase heredan las subclases `Barracas`, `Muro`, `Catapulta` y `MuroCatapulta`. Debes modificarlo
  - `def __init__(self, edad: str, *args, **kwargs)`: Inicializador de la clase que recibe la `edad` actual de cada `Estructura` en formato de `str`. Se asigna el atributo `self.edad` y también se encarga de llamar al método `asignar_atributos_segun_edad()`, el cual deberás completar más adelante. No debes modificarlo
  - `def asignar_atributos_segun_edad(self)`: Debes definir este método y declararlo como **abstracto**, ya que deberá ser implementado en todas las subclases. Será el encargado de definir ciertos atributos según la edad. Debes modificarlo
  - `def accion(self)`: Debes definir este método y declararlo como **abstracto**, ya que deberá ser implementado en todas las subclases. Representa la acción de cada clase y retorna un valor dependiendo del tipo de `Construccion`. Debes modificarlo
  - `def avanzar_edad(self)`: Debes definir este método y declararlo como **abstracto**, ya que deberá ser implementado en todas las subclases. Se utiliza para realizar los cambios necesarios cuando se avance de edad. Debes modificarlo

Clase Barracas

- Esta clase hereda de la clase `Estructura` y es el encargado de crear las unidades `Guerrero`, `Mago` y `MagoGuerrero` de cada `Reino`. Deberás completar la clase para que herede correctamente de la clase `Estructura`. Debes modificarlo
  - `def __init__(self, *args)`: Debes llamar al inicializador de la superclase y entregarle los argumentos posicionales (args). También debes asignar el atributo `self.hp` y darle un valor de `HP_BARRACAS`. Debes modificarlo
  - `def asignar_atributos_segun_edad(self)`: Debes definir este método, en el cual se establecen las unidades (en una lista) que puede crear la barraca. Si la edad es `"Media"` el atributo `self.unidades` será `["Guerrero", "Mago"]`, en cambio si la edad es `"Moderna"` su valor será `["Guerrero", "Mago", "MagoGuerrero"]` Debes modificarlo

- `def avanzar_edad(self)`:Debes definir este método, en el cual debes verificar si la edad actual es igual a `'Media'`. En dicho caso debes cambiar la edad a `'Moderna'` y agregar `'MagoGuerrero'` a la lista de `self.unidades`. Debes modificarlo
- `def accion(self)`: Es el método encargado de simular la creación de las unidades, retornando una instancia de alguna unidad en particular. No debes modificarlo

Clase Muro

- Esta clase hereda de la clase `Estructura` y es el encargado de reparar, o aumentar el `hp`, de cada `Reino`. Deberás completar la clase para que herede correctamente de la clase `Estructura`. Debes modificarlo
  - `def __init__(self, *args, **kwagrs)`: Debes llamar al inicializador de la superclase y entregarle los argumentos posicionales. También debes asignar el atributo `self.hp`, el cual debe ser igual a `HP_MURO` Debes modificarlo
  - `def asignar_atributos_segun_edad(self)`: Debes definir este método el cual fija los rangos de `self.reparacion` (que es una lista de dos valores), según la edad correspondiente. Si la edad es `"Media"` el rango es `[20, 80]`, en cambio si la edad es `"Moderna"` el rango es `[40, 100]` Debes modificarlo
  - `def accion(self)`: Debes definir este método, el cual retorna la reconstrucción que se le efectúa al `Muro`. Esta reconstrucción es un valor aleatorio realizado con un `randint()` el cual recibe como argumentos los rangos de `self.reparacion`. Debes comprobar que la probabilidad sea menor a `PROB_CRITICO_MURO`, sí esto sucede la reconstrucción debe ser multiplicada por 2 Debes modificarlo
  - `def avanzar_edad(self)`: Debes definir este método el cual al ser llamado verifica si la edad actual es igual a `'Media'`. En dicho caso debes cambiar la edad a `'Moderna'` y cambiar el valor de `self.reparacion` por la lista `[40, 100]`. Debes modificarlo

Clase Catapulta

- Esta clase hereda de la clase `Estructura` y es el encargado de atacar al `Reino` enemigo. Deberás completar la clase para que herede correctamente de la clase `Estructura`. Debes modificarlo
  - `def __init__(self, *args)`: Debes llamar al inicializador de la superclase y entregarle los argumentos posicionales. También debes asignar el atributo `self.hp`, el cual debe ser igual a `HP_CATAPULTA`. Debes modificarlo
  - `def asignar_atributos_segun_edad(self)`:Debes definir este método el cual fija los rangos de `self.ataque` (que es una lista de dos valores), según la edad correspondiente. Si la edad es `"Media"` el rango es `[40, 100]`, en cambio si la edad es `"Moderna"` el rango es `[80, 140]` Debes modificarlo
  - `def accion(self)`:Debes definir este método, el cual retorna el ataque que se le efectúa al `Reino` enemigo. Este ataque es un valor aleatorio realizado con un `randint()` el cual recibe como argumentos los rangos de `self.ataque`. Debes comprobar que la probabilidad sea menor a `PROB_CRITICO_CATAPULTA`, sí esto sucede el ataque debe ser multiplicado por 2 Debes modificarlo
  - `def avanzar_edad(self)`: Debes definir este método el cual al ser llamado verifica si la edad actual es igual a `'Media'`. En dicho caso debes cambiar la edad a `'Moderna'` y cambiar el valor de `self.ataque` por la lista `[80, 140]`. Debes modificarlo

Clase MuroCatapulta

- Esta clase hereda de las clases `Muro` y `Catapulta` y es el encargado de reparar el `Reino` junto con atacar al `Reino` enemigo. Deberás completar la clase para que herede correctamente de las clases `Muro` y `Catapulta`. Debes modificarlo
  - `def __init__(self, *args)`: Debes llamar una **única** vez al inicializador de la superclase y entregarle los argumentos posicionales. También debes asignarle el atributo `self.hp` y darle un valor de `HP_MUROCATAPULTA`. Debes modificarlo
  - `def asignar_atributos_segun_edad(self)`: Debes definir este método el cual fija los rangos de `self.ataque` y `self.reparacion`, según la edad correspondiente. Dichos atributos se calculan mediante las mismas fórmulas usadas en las clases `Muro` y `Catapulta`. Por lo que en esta parte simplemente deberás llamar al método `asignar_atributos_segun_edad()` de cada una de las superclases. Debes modificarlo
  - `def accion(self)`: Debes definir este método el cual retorna la reparación y el ataque (en ese orden) que efectúa el `MuroCatapulta`. Al igual que en las superclases esta reparación y el ataque son `randint( )` el cual recibe los argumentos de `self.reparacion` y `self.ataque` respectivamente. Debes comprobar que la probabilidad sea menor a `PROB_CRITICO_MURO_CATAPULTA`, sí esto sucede la reparación y el ataque deben ser multiplicados por 2.5 y posteriormente redondeados. Debes modificarlo
  - `def avanzar_edad(self)`: Debes definir este método, el cual al ser llamado verifica si la edad actual es igual a `'Media'`. En dicho caso debes cambiar la edad a `'Moderna'` y cambiar el valor de `self.reparacion` por la lista `[40, 100]` y el valor de `self.ataque` por la lista `[80, 140]`. Debes modificarlo

Parte 3: Modelación de Reino

En esta parte **solamente deberás completar el método** `avanzar_edad()` de la clase `Reino`, la cual se encuentra definida en el archivo `reino.py`. Recuerda que al final del enunciado hay un diagrama de clases que te puede ayudar a visualizar las relaciones.

Clase Reino

- `class Reino`: Esta clase representa a los reinos que serán parte de la Guerra. Cada `Reino` contiene instancias de las clases que heredan de `Persona` y de `Estructura`, permitiendo así la interacción entre las distintas entidades.
  - `def __init__(self, nombre: str, dinero_inicial: int, construcciones: dict, ganancias: dict)`: Inicializador de la clase que recibe el `nombre` y `dinero_inicial` de cada `Reino`, junto a los diccionarios `construcciones` y `ganancias` que indican los costos de las construcciones y las producciones de las ganancias. Se asignan los distintos atributos necesarios para modelar un `Reino`. No debes modificarlo
  - `def hp(self)`: Es una *property* implementada para que el atributo `self.__hp` no pueda tomar valores negativos. No debes modificarlo
  - `def ataque(self)`: Es una *property* que permite calcular el ataque de todas las `Personas` y `Estructuras` que son capaces de atacar. No debes modificarlo
  - `def reconstruir(self)`: Mediante este método se calcula la reconstrucción total del `Reino`, según las `Personas` y `Estructuras` que son capaces de reconstruir o curar. No debes modificarlo

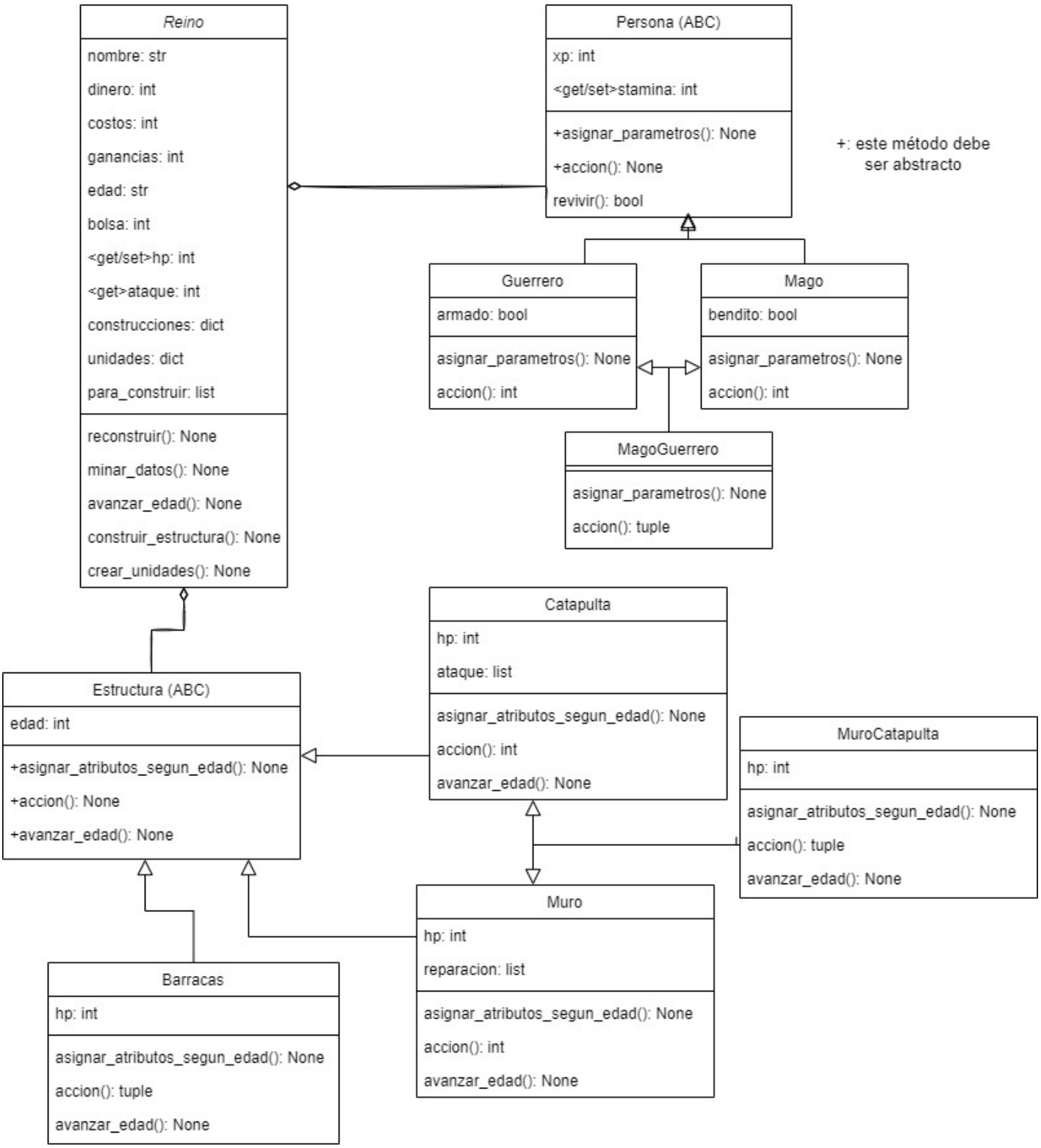
- `def minar_datos(self)`: Este método representa la forma en que el `Reino` gana dinero durante la simulación. No debes modificarlo
- `def construir_estructura(self)`: Escoge una `Estructura` para construir aleatoriamente, suma el `hp` al reino y después lo guarda en `self.construcciones`. No debes modificarlo
- `def crear_unidades(self)`: Se encarga de crear una `Unidad` al azar por cada `Barraca` siempre y cuando tenga dinero, luego se guarda en `self.unidades`. No debes modificarlo
- `def avanzar_edad(self)`: Al llamar a este método debes verificar si la edad actual es igual a `'Media'`. En dicho caso debes cambiar la edad a `'Moderna'` y agregar `'MuroCatapulta'` a la lista `self.para_construir`. Además deberás recorrer los valores del diccionario guardado en `self.construcciones` los cuales serán listas de `Estructuras`, luego para cada `Edifcio` dentro de cada lista deberás llamar a su método `avanzar_edad()`. Por último deberás imprimir lo siguiente **Debes modificarlo**

```
1 f'\n {self.nombre} HA AVANZADO HACIA LA EDAD MODERNA \n'
```

Requerimientos

- (2.2 pts) Modelación de Personas
  - (0.55 pts) Modelar correctamente la clase Persona
  - (0.55 pts) Modelar correctamente la clase Guerrero
  - (0.55 pts) Modelar correctamente la clase Mago
  - (0.55 pts) Modelar correctamente la clase MagoGuerrero
- (3.3 pts) Modelación de Estructuras
  - (0.66 pts) Modelar correctamente la clase Estructura
  - (0.66 pts) Modelar correctamente la clase Barracas
  - (0.66 pts) Modelar correctamente la clase Muro
  - (0.66 pts) Modelar correctamente la clase Catapulta
  - (0.66 pts) Modelar correctamente la clase MuroCatapulta
- (0.5 pts) Modelación de Reino
  - (0.5 pts) definir correctamente el metodo `avanzar_edad()`

Anexo


Reino
nombre: str
dinero: int
costos: int
ganancias: int
edad: str
bolsa: int
<get/set>hp: int
<get>ataque: int
construcciones: dict
unidades: dict
para_construir: list
reconstruir(): None
minar_datos(): None
avanzar_edad(): None
construir_estructura(): None
crear_unidades(): None
Persona (ABC)
xp: int
<get/set>stamina: int
+asignar_parametros(): None
+accion(): None
revivir(): bool
+: este método debe ser abstracto
Guerrero
armado: bool
asignar_parametros(): None
accion(): int
Mago
bendito: bool
asignar_parametros(): None
accion(): int
MagoGuerrero
asignar_parametros(): None
accion(): tuple
Catapulta
hp: int
ataque: list
asignar_atributos_segun_edad(): None
accion(): int
avanzar_edad(): None
MuroCatapulta
hp: int
asignar_atributos_segun_edad(): None
accion(): tuple
avanzar_edad(): None
Estructura (ABC)
edad: int
+asignar_atributos_segun_edad(): None
+accion(): None
+avanzar_edad(): None
Muro
hp: int
reparacion: list
asignar_atributos_segun_edad(): None
accion(): int
avanzar_edad(): None
Barracas
hp: int
asignar_atributos_segun_edad(): None
accion(): tuple
avanzar_edad(): None