Onjetivo:
Vas a programar una simulación de batalla entre civilizaciones usando programación orientada a objetos (POO).
El juego tendrá unidades militares y trabajadores, y cada civilización tendrá que gestionar sus recursos y enfrentarse en combate.

UNIVERSIDADE DA CORUÑA

Grado de Ciencia e Ingeniería de Datos
Fundamentos de la Programación II – Curso 2024/25

Práctica 1 (Programación orientada a objetos)

Una reconocida compañía de videojuegos nos ha encargado el desarrollo de una prueba de concepto para su nuevo título de *Civilizations*. En este juego de estrategia en tiempo real, cada civilización debe gestionar sus recursos, crear nuevas unidades y enfrentarse durante varios turnos. Nos piden una implementación de una pequeña demo, que se dividirá en 3 partes.

·Es un modelo base que no se puede instanciar directamente.

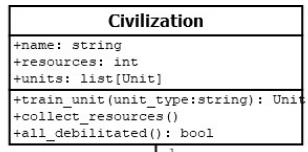
·Contiene métodos abstractos que las subclases deben implementar.→

PARTE 1: Implementación de la jerarquía de clases que nos permita instanciar distintos tipos de unidades, usando para ello el paradigma de la programación orientada a objetos. Se pide modelar una clase abstracta (Unit) y una serie de clases hijas concretas para instanciar distintos tipos de unidades militares (Archer, Cavalry, Infantry) así como unidades de recolección (Worker). La jerarquía, atributos y métodos a implementar se visualizan como parte de la Figura 1. Dichas clases se definirán dentro de un módulo denominado unit.py.

que es la clase abstracta unit ??? ↰

na clase abstracta es una clase que no puede ser instanciada directamente y sirve como modelo para que otras clases hereden de ella. Se usa para definir métodos que deben ser implementados por las subclases.

superclase →



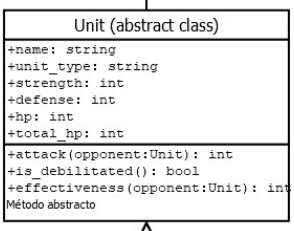
Civilización
Atributos:
nombre: string
recursos: int
unidades: list [Unit]
Métodos:
entrenar_unidad(unit_type: string): Unit
recolectar_recursos()
todo_debilitado(): bool (devuelve true si todas las unidades están muertas)

clases hijas → coge los atributos de la clase abstracta y se añaden los que hagan falta.

Unidad (clase abstracta)
Atributos:
nombre: sting ()
tipo_unidad: sting (arquero,trabajador,...)
fuerza: int
defensa: int
puntos_vida (hp): int
puntos_vida_totales: int
Métodos:
atacar(oponente: Unidad): int
esta_debilitada(): bool
efectividad(oponente: Unidad): int

1
0...n

from abc import ABC, abstractmethod
→ abc significa Abstract Base Classes



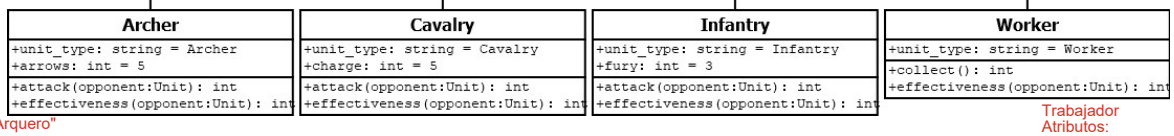
← superclase

Es la base para todas las unidades, pero no se puede instanciar directamente.

Caballería
Atributos:
tipo_unidad: string = "Caballería"
carga: int = 5
Métodos:
atacar(oponente: Unidad): int
efectividad(oponente: Unidad): int

Infantería
Atributos:
tipo_unidad: string = "Infantería"
furia: int = 3
Métodos:
atacar(oponente: Unidad): int
efectividad(oponente: Unidad): int

subclases hijas →



→ Clases Heredadas

Arquero
Atributos:
tipo_unidad: string = "Arquero"
flechas: int = 5
Métodos:
atacar(oponente: Unidad): int
efectividad(oponente: Unidad): int

Trabajador
Atributos:
tipo_unidad: string = "Trabajador"
Métodos:
recolectar(): int
efectividad(oponente: Unidad): int

Figura 1. Jerarquía de clases.

La clase abstracta Unit implementará los atributos comunes a todos los tipos de unidades: nombre (name, de tipo `string`), tipo de unidad (unit_type, de tipo `str`), fuerza (strength, de tipo `int`), defensa (defense, de tipo `int`), puntos de salud (hp, de tipo `int`), y puntos de salud totales (total_hp, de tipo `int`). El atributo unit_type es un atributo común, pero su valor específico se definirá directamente dentro del constructor de cada subclase sin pasarlo como parámetro en la función constructora de las clases concretas, y se le asignará el valor de 'Archer', 'Cavalry', 'Infantry' o 'Worker', según corresponda.

↳ para leer y escribir esos atributos. Primero se debe escribir el @prperty para que @getter y @setter se puedan utilizar.

Además, se implementarán los métodos de acceso (getters) y modificación (setters) para dichos atributos. Estos métodos se definirán obligatoriamente utilizando los decoradores @property y @setter, siguiendo un estilo Pythonic (ver ejemplo en property_example.py en la carpeta con los materiales de esta práctica).

el @getter se utiliza para que muestre el valor de la variable, el @getter se utiliza para modificar el valor de la variable.

La clase abstracta Unit implementará los métodos comunes a todos los tipos de unidad:

- attack (opponent: 'Unit') -> int
  Funcionamiento: Comportamiento de ataque por defecto para las unidades, que disminuye el valor del atributo hp del oponente en 1 unidad de daño.
  Devuelve: 1 unidad de daño.
  Postcondición. La vida del oponente debe ser mayor o igual que cero.

al hacer daño baja 1 de hp del openente => devuelve 1 daño ¿a quien hace daño despues de bajar el hp?

baja el hp del oponente al que ataca, pero el oponente debe tener por lo menos una vida.

- is_debilitated() -> bool
  Devuelve: `True` si el atributo hp tiene valor cero y `False` en caso contrario.

está debilitado si el oponente se queda sin vida.

Además, la clase abstracta incluirá el método abstracto effectiveness(opponent: 'Unit') -> int, implementado en las clases hijas (ANEXO I), y redefinirá el método mágico __str__() para que, al mostrarse por pantalla una instancia de un objeto Unit, devuelva una cadena de texto con la información:

el método mágico hace que cuando se llama al print aparezca una información legible para las personas, sin el método mágico aparecería algo como <__main__.Persona object at 0x...>

{name} ({unit_type}) Stats: ATT: {strength}, DEF: {defense}, HP: {hp/total_hp}.

donde los valores entre {} indican los valores de los atributos de la unidad[1]

crear instancias significa crear un objeto para una clase concreta (no se puede instanciar una clase abstracta), es decir, darle un valor a una variable definida en la clase concreta.

En la jerarquía Unit existirán las siguientes clases concretas: Archer, Cavalry, Infantry y Worker, que se usarán para crear instancias de distintos tipos de Unit. Ver ANEXO I con la especificación concreta para la implementación de los métodos de cada una de ellas.

para provar el codigo

Para comprobar automáticamente la implementación correcta de los aspectos básicos de la jerarquía de clases y sus métodos, ejecutad el archivo unit_tests.py desde la línea de comandos con: python unit_tests.py -v, o directamente desde Spyder para conocer qué casos de prueba causan problemas.

hay que hacer una clase para civilizacion con atributos:
· nombre
· recursos
· unidades (arqueros, caballería, infantería y trabajadores)

PARTE 2. Implementar un módulo civilization.py que defina una clase Civilization (Figura 1). La clase Civilization dispone de tres atributos: el nombre de la civilización (name, de tipo `string`), la cantidad de recursos (resources, de tipo `int`) y una lista con todas sus unidades (units, de tipo `list`), que incluye tanto unidades militares como trabajadores (respetando el orden en el que se van creando, es decir, insertando las nuevas unidades por el final). Deberán definirse sus métodos getter y setter utilizando los decoradores `@property` y `@setter` correspondientes, así como implementar el resto de sus métodos (Figura 1), cuya especificación se recoge en el ANEXO II.

PARTE 3. Implementar la lógica de la batalla entre dos civilizaciones (main.py). Implementa el archivo main.py para realizar la simulación completa. Para implementar este punto, se incluyen dos ficheros de pruebas denominados battle0.txt y battle1.txt, cuyo formato es:

Parámetro	Descripción
{Civilization_1}: {resources_1}	Nombre de la primera civilización y sus recursos iniciales antes de crear unidades.
{Civilization_2}: {resources_2}	Nombre de la segunda civilización y sus recursos iniciales antes de crear unidades.
Turns: {num_turns}	Número total de turnos que durará la simulación.
Workers: {num_workers}	Número inicial de unidades de tipo Worker en cada civilización.
Archers: {num_archers}	Número inicial de unidades de tipo Archer en cada civilización.
Cavalry: {num_cavalry}	Número inicial de unidades de tipo Cavalry en cada civilización.
Infantry: {num_infantry}	Número inicial de unidades de tipo Infantry en cada civilización.

Tabla 1. Formato de los ficheros de prueba.

[1] Ejemplo de salida: Archer_1 (Archer) Stats: ATT: 50, DEF: 20 HP: 45/55 (donde el 45/55 refleja que la unidad ha sufrido 10 puntos de daño respecto a sus puntos de vida totales).

El fichero main.py contiene la funcionalidad básica para abrir los ficheros y leer su contenido.[2] Antes de comenzar la simulación, cada civilización crea sus unidades iniciales según los valores especificados en el archivo de entrada. La creación de estas unidades consume parte de los recursos iniciales y se realizará en el orden en el que aparece en el fichero (trabajadores, arqueros, etc.). Si los recursos se agotan antes de crear todas las unidades, dichas unidades no serán creadas. Crear un Archer, Cavalry o Infantry cuesta 60 unidades de recursos, mientras que crear un Worker cuesta 30.

Para cada civilización, el nombre de la unidad será de la forma X_Y, donde X es Archer, Cavalry, Infantry o Worker; e Y es el número de unidades de ese tipo creada hasta ese momento para dicha civilización. Por ejemplo, una civilización con dos Archer y un Worker sería: Archer_1, Archer_2, Worker_1. Cualquier unidad creada respetará los parámetros especificados en la Tabla 2.

3 puntos de ataque con las flechas, sin ellas solo 1 punto

5 puntos de daño por el ataque

3 puntos de daño por el ataque

Unidad	Strength	Defense	Hp/Total_hp	Recursos que consume	Atributo especial
Archer	7	2	15/15	60	3
Cavalry	5	2	25/25	60	5
Infantry	3	2	25/25	60	3
Worker	1	0	5/5	30	-

Tabla 2. Valores de los atributos para las unidades creadas.

Una vez leído y procesado el fichero, la simulación se desarrollará en N turnos, y cada turno (que comienza en 1), comprenderá las siguientes fases:

a. Fase de Recolección: Cada civilización invocará el método collect_resources() (ver ANEXO II) para que los trabajadores operativos (hp>0) acumulen recursos. A continuación, se imprimirá un mensaje indicativo, mostrando la siguiente información:

solo los de clase worker pueden acumular recursos. Solo atacan o son atacados cuando no queden unidades militares.
10 recursos por cada worker y
cuantos tenemos antes de empezar (al principio para crear las unidades iniciales)

PHASE 1: REPORT

{civilization1.name} Resources: {civilization1.resources}

Worker : {worker_unit1.name} ({worker_unit1.hp}/{worker_unit1.total_hp}), ..., {worker_unitN.name} ({worker_unitN.hp}/{worker_unitN.total_hp})
Archer : {archer_unit1.name} ({archer_unit1.hp}/{archer_unit1.total_hp}), ..., {archer_unitM.name} ({archer_unitM.hp}/{archer_unitM.total_hp})
Cavalry : {cavalry_unit1.name} ({cavalry_unit1.hp}/{cavalry_unit1.total_hp}), ..., {cavalry_unitP.name} ({cavalry_unitP.hp}/{cavalry_unitP.total_hp})
Infantry : {infantry_unit1.name} ({infantry_unit1.hp}/{infantry_unit1.total_hp}), ..., {infantry_unitQ.name} ({infantry_unitQ.hp}/{infantry_unitQ.total_hp})

{civilization2.name} Resources: {civilization2.resources}

Worker : {worker_unit1.name} ({worker_unit1.hp}/{worker_unit1.total_hp}), ..., {worker_unitN.name} ({worker_unitN.hp}/{worker_unitN.total_hp})
Archer : {archer_unit1.name} ({archer_unit1.hp}/{archer_unit1.total_hp}), ..., {archer_unitM.name} ({archer_unitM.hp}/{archer_unitM.total_hp})
Cavalry : {cavalry_unit1.name} ({cavalry_unit1.hp}/{cavalry_unit1.total_hp}), ..., {cavalry_unitP.name} ({cavalry_unitP.hp}/{cavalry_unitP.total_hp})
Infantry : {infantry_unit1.name} ({infantry_unit1.hp}/{infantry_unit1.total_hp}), ..., {infantry_unitQ.name} ({infantry_unitQ.hp}/{infantry_unitQ.total_hp})

donde las unidades que aparecen son sólo aquellas con hp > 0.

[2] Consola: python main.py battle0.txt. o Spyder, con el fichero main.py seleccionado, Run -> Configuration per file, Run file with custom configuration, General settings y en command line options añadir battle0.txt .

b. Fase de Producción: Cada civilización evalúa si es posible crear una nueva unidad (militar o de recolección) en función de los recursos disponibles, de acuerdo con los valores de la Tabla 2.

- Si turno % 4 == 0, se intenta crear un Archer.
- Si turno % 4 == 1, se intenta crear un Cavalry.
- Si turno % 4 == 2, se intenta crear un Infantry.
- Si turno % 4 == 3, se intenta crear un Worker.

no entiendo → dependiendo del turno que toque se puede crear una de las unidades (archer, calvary...)

A continuación, se imprimirá el siguiente mensaje:

PHASE 2: PRODUCTION

---

esto indica para cada civilización que unidad se creó, creo que es el "método mágico" el que permite que esta información sea legible, si no serían un conjunto de bits.

{civilization1.name} creates {unit1.name} ({unit1.unit_type}) Stats: ATT: {unit1.strength} DEF: {unit1.defense}, HP: {unit1.hp/unit1.total_hp}.

{civilization2.name} creates {unit2.name} ({unit2.unit_type}) Stats: ATT: {unit2.strength} DEF: {unit2.defense}, HP: {unit2.hp/unit2.total_hp}.

Si una civilización no ha podido crear ninguna unidad, se indicará en su lugar el siguiente mensaje:

{civilization1.name} cannot create any unit right now.

c. Fase de Batalla: Simulará una batalla entre dos civilizaciones predefinidas siguiendo un orden de ataque alternado (en forma de "cremallera"). Esto significa que:

- La primera unidad militar operativa (hp>0) de la Civilización 1 ataca.
- A continuación, la primera unidad militar operativa de la Civilización 2 ataca.
- Luego, la segunda unidad militar operativa de la Civilización 1 ataca.
- Seguidamente, la segunda unidad militar operativa de la Civilización 2 ataca.
- Y así sucesivamente, hasta que todas las unidades han atacado, alternando entre ambas civilizaciones.
- Si una civilización tiene más unidades operativas que otra, las unidades restantes atacarán al final del turno.

Para seleccionar el objetivo se procederá de la siguiente manera:

- Cada vez que una unidad ataca, seleccionará como objetivo la primera unidad militar operativa de la civilización contraria que pertenezca a un tipo contra el cual su ataque sea más efectivo. Por ejemplo, si la unidad atacante tiene mayor efectividad contra unidades del tipo "Infantry", buscará como objetivo la primera unidad que sea de tipo Infantry y que tenga hp > 0 en la civilización contraria.
- Las unidades de tipo Worker no atacan, y tampoco son atacadas, mientras queden unidades militares operativas. Sólo son seleccionables, bien como atacantes u objetivos, cuando todas las unidades militares han sido eliminadas.

El formato completo de la impresión para esta fase es:

PHASE 3: BATTLE STATUS

Alternating attacks (Cremallera)

{civilization1.name} - {attacker_unit1.name} attacks {civilization2.name} - {target_unit1.name} with damage {damage} (hp={target_unit1.hp}/{target_unit1.total_hp}).

{civilization2.name} - {attacker_unit2.name} attacks {civilization1.name} - {target_unit2.name} with damage {damage} (hp={target_unit2.hp}/{target_unit2.total_hp}).

{civilization1.name} - {attacker_unit3.name} attacks {civilization2.name} - {target_unit3.name} with damage {damage} (hp={target_unit3.hp}/{target_unit3.total_hp}).

{civilization2.name} - {attacker_unit4.name} attacks {civilization1.name} - {target_unit4.name} with damage {damage} (hp={target_unit4.hp}/{target_unit4.total_hp}).

...

#End of alternating sequence: One civilization has no more attackers left

#The remaining units of the stronger civilization (e.g. civilization1) now attack in sequence

{civilization1.name} - {attacker_unit5.name} attacks {civilization2.name} - {target_unit5.name} with damage {damage} (hp={target_unit5.hp}/{target_unit5.total_hp}).

{civilization1.name} - {attacker_unit6.name} attacks {civilization2.name} - {target_unit6.name} with damage {damage} (hp={target_unit6.hp}/{target_unit6.total_hp}).

{civilization1.name} - {attacker_unit7.name} attacks {civilization2.name} - {target_unit7.name} with damage {damage} (hp={target_unit7.hp}/{target_unit7.total_hp}).

...

Until all attacking units from the stronger civilization have attacked

Al finalizar la simulación del juego, **se mostrarán varias estadísticas, para lo cual se utilizará obligatoriamente la librería pandas**. Se muestra un ejemplo dentro del fichero pandas_example.py, incluido junto con los materiales de esta práctica. **No se aceptarán cálculos estadísticos realizados mediante implementaciones *ad hoc* que empleen diccionarios u otras estructuras de datos**. En concreto, deberá mostrarse la media y la desviación estándar para: (1) el daño promedio causado por cada unidad a nivel individual para cada civilización, (2) el daño promedio causado por cada tipo de unidad (Archer, Cavalry, Infantry, Worker) para cada civilización y (3) el daño promedio que cada tipo de unidad inflige a cada uno de los otros tipos, también para cada civilización.

Entrega

Se entregará un archivo zip con (exclusivamente) los archivos de código fuente—unit.py, civilization.py y main.py—y un manual de usuario (en formato pdf) que incluirá: (1) el mecanismo de ejecución del programa y (2) *una concisa y explicativa* descripción de las fases de desarrollo realizadas. **La memoria no debe exceder las 3 páginas** (fuente similar a Calibri, Arial o Times New Roman, con un tamaño de 11 puntos). La falta de cada apartado penalizará un 10% sobre la nota final. En cada archivo del código fuente y en la primera página del pdf se indicará el nombre y correo electrónico (UDC) de los miembros de la pareja de prácticas.

El código de cada clase se documentará con *docstrings* con el formato del ejemplo del ANEXO III. La falta de esta documentación se penalizará con un 10% sobre la nota final.

La práctica debe respetar los principios básicos de orientación a objetos vistos en clase (herencia, abstracción, encapsulación, etc.). Si no se cumplen, la práctica será suspendida automáticamente.

Fecha límite de entrega: viernes, 7 de marzo de 2025 a las 23:59.

Dónde se entrega: en el apartado de Prácticas de Moodle.

Quién entrega: Un sólo miembro de la pareja.

ANEXO I - Especificación de los métodos de las subclases Unit

Archer

attack(opponent: Unit) -> int

Si el valor del atributo arrows es mayor que cero, disminuye la vida del oponente en n unidades de daño, calculadas como máximo (1, (factor*$strength_{atacante}$) – $defense_{oponente}$), donde factor es igual a 1.5 contra unidades de tipo Calvary, 1 contra unidades de tipo Archer/Worker y 0.5 contra unidades de tipo Infantry. El resultado se redondeará siempre al número entero inferior. A continuación, el valor del atributo arrows se reduce en una unidad. Si el valor de arrows es cero, causa 1 unidad de daño.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

effectiveness(p: Unit) -> int

Devuelve 1 si p es tipo Cavalry, 0 si es tipo Archer/Worker y –1 si es tipo Infantry.

Cavalry

attack(opponent: Unit) -> int

Disminuye la vida del oponente en n unidades de daño, calculadas como máximo (1, ($charge_{atacante}$+factor*$strength_{atacante}$) – $defense_{oponente}$), donde factor es igual a 1.5 contra unidades de tipo Infantry, 1 contra unidades de tipo Cavalry/Worker y 0.5 contra unidades de tipo Archer. El resultado se redondeará siempre al número entero inferior.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

effectiveness(p: Unit) -> int

Devuelve 1 si p es tipo Infantry, 0 si es tipo Cavalry/Worker y –1 si es tipo Archer.

Infantry

attack(opponent: Unit) -> int

Disminuye la vida del oponente en n unidades de daño, calculadas como máximo (1, ($fury_{atacante}$+factor*$strength_{atacante}$) – $defense_{oponente}$), donde factor es igual a 1.5 contra unidades de tipo Archer, 1 contra unidades de tipo Infantry/Worker y 0.5 contra unidades de tipo Cavalry El resultado se redondeará siempre al número entero inferior.

Devuelve: El número de unidades de daño causadas.

Postcondición: La vida del oponente p no debe ser nunca inferior a cero.

effectiveness(p: Unit) -> int

Devuelve 1 si p es tipo Archer, 0 si es tipo Infantry/Worker y –1 si es tipo Cavalry.

Worker

collect() -> int

Devuelve: 10 (indicando que se recolectan 10 nuevos recursos).

effectiveness(p: Unit) -> int

Devuelve -1

ANEXO II - Especificación de los métodos de la clase Civilization

train_unit(unit_type: str) -> Unit

Funcionamiento: Permite crear una nueva unidad del tipo especificado ("Archer", "Cavalry", "Infantry" o "Worker").
Verifica que existan recursos suficientes.
Agrega la nueva unidad, insertando por el final, a la lista units.
Devuelve: La instancia de la unidad creada.
Postcondición: Si no hay suficiente recursos, devuelve None.

collect_resources() -> None

Funcionamiento: Recorre la lista units y, para cada unidad de tipo Worker, invoca su método collect(). La cantidad recolectada se suma al atributo resources.

all_debilitated() -> bool

Devuelve: `True` si todas las unidades, incluidos los Workers, han sido eliminadas (hp = 0). `False` en otro caso.

Cada turno tiene 3 fases:

Fase de Recolección:
· Los Workers recolectan 10 recursos cada uno.
· Se muestra un reporte del estado actual.

Fase de Producción:
· Si hay recursos suficientes, la civilización crea una unidad siguiendo este orden:
Turno % 4 == 0 → Archer
Turno % 4 == 1 → Cavalry
Turno % 4 == 2 → Infantry
Turno % 4 == 3 → Worker

Fase de Batalla:
· Las unidades atacan en orden alternado (uno de cada civilización por turno).
· Buscan el oponente más débil según su tipo.
· Los Workers no atacan ni son atacados, hasta que no queden unidades militares.

Cuando una unidad ataca:
· Resta vida al oponente según su efectividad.
· Si el hp del oponente llega a 0, queda fuera de combate.

ANEXO III – Ejemplo de docstrings

Docstrings para clases Python

```
class Clase:
    """Una sola línea de resumen.

    Descripción en varias líneas

    Attributes
    ----------
    attr1 : tipo
        Descripción.
    attr2 : tipo
        Descripción.

    Methods
    -------
    metodo1(param1):
        Una línea de resumen.
    """

    def __init__(self, attr1, attr2):
        """Asigna atributos al objeto.

        Parameters
        ----------
        attr1 : tipo
             Descripción.
        attr2 : tipo
             Descripción.

        Returns
        -------
        None.
        """

    def metodo1(self):
        """Una sola línea de resumen.

        Parameters
        ----------
        param1 : tipo
             Descripción.

        Returns
        --------
        str
            Resultado de...
        """
```

Docstrings para funciones Python

```
def functionA(param1):
    """Una sola línea de resumen.

    Parameters
    ----------
    param1 : tipo
        Descripción.

    Returns
    -------
    tipo
       Descripción.
    """
```