Jocul are la baza colectarea de resurse, construire de cladiri. Initial, jocul are mai multi jucatori, fiecare jucator isi alege o civilizatie si va primi in Magazie resurse specifice pentru fiecare civilizatie, o cladire centrala si 3 persoanje satean. Cu resursele initiale, jucatorul poate folosi un Satean pentru a construi cladiri noi sau poate utiliza cladirile sa genereze alte personaje. Scopul jocului este de formarea a unei armate ce va duce un razboi cu un oponent.

Resursele vor avea o cantitate predefinita si vor fi de 3 tipuri: Hrana, Lemn si Piatra. Resursele pot fi consumate prin constructia cladirilor si generarea persoanjelor.

Sateanul colecteaza un anumit tip de resurse si aduce in magazie 50 de unitati din cele colectate. La utilizarea resurselor, acestea sunt mutate catre entitatea ce la va folosi.

Civilizatiile pe care le pune jocul la dispozitie sunt Vikingi si Huni, si in functie de civilizatie, se primeste o anumita cantitate din fiecare tip de resursa si un anumit personaj erou.

Cladirile sunt concepute astfel incat prezinta un nivel de fortificare si un echivalent in aur in functie de cate resurse s-au consumat pe constructia respectiva.

3 tipuri de cladiri: Centru, Cazarma, Staul. Rolul cladirilor e de a genera noi personaje. Centrul nu consuma resurse la constructie si este predefinit la pornirea jocului, rolul acestuia este de a genera personaje de tip Satean. Cazarma are rolul de a genera pesonaje tip Infanterist in schimbul unor resurse, iar Staului genereaja personaje tip Cavaler in schimbul resurselor.

Optional: posibilitatea de a avansa cavalerul/ infanteristul la un nivel superior prin unirea a 2 cavaleri de nivel 1 in unul de nivel 2 (asemanator pt infanterist – nivel maxim 3).

Personajele (satean, infanterist, cavaler) sunt de doua tipuri: personaj civil si personaj militar. Personajele militare (infanterist, cavaler) au armura si o putere de atac, personajele civile prezinta nnumai viata si sunt numai de tip Satean.

La selectarea tipului de civilizatii, fiecare jucator primeste un erou unic de clasa Erou. Eroul face parte si din categoria infanterie si din catergoria cavalerie

Resurse de pornire pentru civilizația Vikingi: 100xHrana, 150xLemn, 150xPiatra;

Resurse de pornire pentru civilizația Huni: 175xHrana, 125xLemn, 100xPiatra;

Cost constructie Cazarma: 50xLemn, 100xPiatra;

Cost constructie Staul: 100xLemn, 50xPiatră;

Satean: ID, capacitate recoltare: 50 unitati oricare material, cost creare: 50 hrana

Infanterist: ID, viata:55, armura: 1, atac: 5, cost creare: 60 hrana, 20 lemn

Cavaler: ID, viata:100, armura: 2, atac:10, cost creare: 60 hrana, 80 lemn

Erou:ID, viata 150, armura:5, atac:15

Proiectați și implementați obiectual, componentele necesare logicii funcționării jocului

Se vor implementa mecanismele necesare comparării, adunării și scăderii a două resurse (supraincarcare de operatori). Se va implementa un mecanism de afișare a cantității de resurse din magazie folosind operandul **cout**.

To implement the detailed Python classes for your game, we'll create several classes to represent the various components: `Resource`, `Building`, `Character`, `Civilization`, and `Game`. These will include methods for resource management, building construction, character creation, and game mechanics like unit advancement.

### 1. Resource Class

This class represents resources and includes operator overloading for addition, subtraction, and comparison.

```python

class Resource:

def \_\_init\_\_(self, food, wood, stone):

self.food = food

self.wood = wood

self.stone = stone

def \_\_add\_\_(self, other):

return Resource(self.food + other.food, self.wood + other.wood, self.stone + other.stone)

def \_\_sub\_\_(self, other):

return Resource(max(0, self.food - other.food), max(0, self.wood - other.wood), max(0, self.stone - other.stone))

def \_\_str\_\_(self):

return f"Food: {self.food}, Wood: {self.wood}, Stone: {self.stone}"

def is\_enough(self, other):

return self.food >= other.food and self.wood >= other.wood and self.stone >= other.stone

```

### 2. Building Class

This class represents different types of buildings in the game.

```python

class Building:

def \_\_init\_\_(self, type, resource\_cost):

self.type = type

self.resource\_cost = resource\_cost

# Additional attributes like fortification level can be added here

def \_\_str\_\_(self):

return f"Building Type: {self.type}, Cost: {self.resource\_cost}"

```

### 3. Character Classes

We'll have a base `Character` class and derived classes for different types of characters.

```python

class Character:

def \_\_init\_\_(self, character\_id, health, cost):

self.character\_id = character\_id

self.health = health

self.cost = cost

class Villager(Character):

def \_\_init\_\_(self, character\_id):

super().\_\_init\_\_(character\_id, health=50, cost=Resource(50, 0, 0))

self.gathering\_capacity = 50

class Infantry(Character):

def \_\_init\_\_(self, character\_id):

super().\_\_init\_\_(character\_id, health=55, cost=Resource(60, 20, 0))

self.armor = 1

self.attack\_power = 5

class Cavalry(Character):

def \_\_init\_\_(self, character\_id):

super().\_\_init\_\_(character\_id, health=100, cost=Resource(60, 80, 0))

self.armor = 2

self.attack\_power = 10

class Hero(Character):

def \_\_init\_\_(self, character\_id):

super().\_\_init\_\_(character\_id, health=150, cost=Resource(0, 0, 0)) # Cost for heroes can be defined as needed

self.armor = 5

self.attack\_power = 15

```

### 4. Civilization Class

This class represents a civilization in the game.

```python

class Civilization:

def \_\_init\_\_(self, name, starting\_resources, hero):

self.name = name

self.resources = starting\_resources

self.hero = hero

self.buildings = []

self.characters = []

def build(self, building):

if self.resources.is\_enough(building.resource\_cost):

self.resources -= building.resource\_cost

self.buildings.append(building)

return True

return False

def create\_character(self, character\_class):

character = character\_class(len(self.characters) + 1)

if self.resources.is\_enough(character.cost):

self.resources -= character.cost

self.characters.append(character)

return character

return None

def \_\_str\_\_(self):

return f"Civilization: {self.name}, Resources: {self.resources}, Hero: {self.hero.character\_id}"

```

### 5. Game Class

This class manages the overall game state.

```python

class Game:

def \_\_init\_\_(self):

self.civilizations = []

def add\_civilization(self, civilization):

self.civilizations.append(civilization)

# Additional methods for game logic can be added here

```

### Example Usage

```python

viking\_resources = Resource(100, 150, 150)

hun\_resources = Resource(175, 125, 100)

viking\_civ = Civilization("Vikings", viking\_resources, Hero("Viking Hero"))

hun\_civ = Civilization("Huns", hun\_resources, Hero("Hun Hero"))

barracks = Building("Barracks", Resource(0, 50, 100))

stable = Building("Stable", Resource(0, 100, 50

))

viking\_civ.build(barracks)

viking\_civ.create\_character(Infantry)

print(viking\_civ)

```

This code provides a foundational structure for your game. You can expand it with more features like unit advancement, combat mechanics, resource gathering by villagers, and more complex game rules.