Воин, Маг, Лучник

Создать иерархию классов для игровых персонажей. В иерархии должен быть абстрактный класс с абстрактными методами для атаки и защиты. Для конкретных типов персонажей (воин, маг, лучник) должны быть созданы соответствующие классы, наследующиеся от абстрактного класса и реализующие методы для атаки и защиты. Кроме того, для каждого типа персонажа должны быть созданы подтипы (например, для воина - рыцарь, берсерк, для мага - огненный маг, ледяной маг, для лучника - снайпер, стрелок из дротика), которые будут наследоваться от соответствующего типа и иметь свои уникальные способности.

Решение:

```
from abc import ABC, abstractmethod


class Character(ABC):
    def __init__(self, health, attack, defense):
        self.health = health
        self.attack = attack
        self.defense = defense

    @abstractmethod
    def attack_character(self, character):
        pass

    @abstractmethod
    def defend_character(self, damage):
        pass


class Warrior(Character, DefendMixin):
    def __init__(self, health, attack, defense, rage=0):
        super().__init__(health, attack, defense)
        self.rage = rage

    def attack_character(self, character):
        damage = self.attack * (1 + self.rage * 0.05)
        character.defend_character(damage)
        self.rage += 1
        return f'{self.__class__.__name__} атакует {character.__class__.__name__} нанося {damage}
урона.'

    def defend_character(self, damage):
        damage -= self.defense
        if damage > 0:
            self.health -= damage
            self.rage -= 1 if self.rage > 0 else 0
        return Def


class Knight(Warrior):
    def __init__(self, health, attack, defense):
        super().__init__(health, attack, defense)

    def defend_character(self, damage):
        damage -= self.defense * 1.2
        if damage > 0:
            self.health -= damage
        return f'{self.__class__.__name__} получает {damage} урона. Текущее здоровье: {self.health}'


class Berserk(Warrior):
    def __init__(self, health, attack, defense):
```

```
        super().__init__(health, attack, defense)

    def attack_character(self, character):
        damage = self.attack * (1 + self.rage * 0.1)
        character.defend_character(damage)
        self.rage += 1
        return f'{self.__class__.__name__} атакует {character.__class__.__name__} нанося {damage}
урона.'


class Mage(Character):
    def __init__(self, health, attack, defense, mana=0):
        super().__init__(health, attack, defense)
        self.mana = mana

    def attack_character(self, character):
        damage = self.attack * (1 + self.mana * 0.05)
        character.defend_character(damage)
        self.mana -= 1
        return f'{self.__class__.__name__} атакует {character.__class__.__name__} нанося {damage}
урона.'

    def defend_character(self, damage):
        damage -= self.defense
        if damage > 0:
            self.health -= damage
            self.mana += 1 if self.mana < 10 else 10
        return f'{self.__class__.__name__} получает {damage} урона. Текущее здоровье: {self.health}'


class FireMage(Mage):
    def __init__(self, health, attack, defense):
        super().__init__(health, attack, defense)

    def attack_character(self, character):
        damage = self.attack * (1 + self.mana * 0.1)
        character.defend_character(damage)
        self.mana -= 1
        return f'{self.__class__.__name__} атакует {character.__class__.__name__} нанося {damage}
урона.'


class Archer(Character):
    def __init__(self, health, attack, defense, arrows=0):
        super().__init__(health, attack, defense)
        self.arrows = arrows

    def attack_character(self, character):
        damage = self.attack * (1 + self.arrows * 0.05)
        character.defend_character(damage)
        self.arrows -= 1
        return f'{self.__class__.__name__} атакует {character.__class__.__name__} нанося {damage}
урона.'

    def defend_character(self, damage):
        damage -= self.defense
        if damage > 0:
            self.health -= damage
            self.arrows += 1 if self.arrows < 10 else 10
        return f'{self.__class__.__name__} получает {damage} урона. Текущее здоровье: {self.health}'


class Sniper(Archer):
    def __init__(self, health, attack, defense):
        super().__init__(health, attack, defense)
```

```
    def attack_character(self, character):
        damage = self.attack * (1 + self.arrows * 0.1)
        character.defend_character(damage)
        self.arrows -= 1
        return f'{self.__class__.__name__} атакует {character.__class__.__name__} нанося {damage}
урона.'


# Пример запуска
characters = [
    Knight(health=150, attack=20, defense=15),
    Berserk(health=120, attack=25, defense=10),
    FireMage(health=100, attack=30, defense=5),
    Sniper(health=110, attack=25, defense=15),
    Warrior(health=180, attack=20, defense=10)
]

for i in range(len(characters)):
    for j in range(i+1, len(characters)):
        print(characters[i].attack_character(characters[j]))
        print(characters[j].defend_character(characters[i].attack))
```

Здесь мы создаем абстрактный класс `Character` с абстрактными методами `attack_character()` и `defend_character()`. Этот класс будет базовым для всех типов игровых персонажей. Затем мы создаем конкретные классы для воина, мага и лучника, наследующиеся от `Character` и реализующие методы для атаки и защиты. Кроме того, мы создаем подтипы для каждого типа персонажа, наследующиеся от соответствующего типа и имеющие свои уникальные способности.

В примере запуска мы создаем экземпляры каждого типа персонажа и подтипа, и проводим бой между ними. Это демонстрирует полиморфизм, поскольку мы вызываем одни и те же методы для разных объектов, и каждый объект выполняет методы по-своему.

Абстракция в этом решении заключается в том, что мы создаем общий интерфейс (абстрактный класс `Character`) для различных типов игровых персонажей, который позволяет нам взаимодействовать с ними единообразно, несмотря на их различия. Наследование и полиморфизм позволяют нам создавать новые классы, основанные на существующих, и переопределять их методы, чтобы реализовать нужное нам поведение.