Concepte OOP - exercitii

Clasa 'Player Status'

Cerinta

Realizeaza design-ul si implementarea unei clase PlayerStatus care mentine **actualizata** starea unui Player in cadrul jocului.

Clasa trebuie sa respecte principiile OOP invatate pana la acest moment. De exemplu, tine cont de: vizibilitatea starii obiectelor, separarea corecta a conceptelor, modularitatea codului, refolosirea functionalitatii, ascunderea detaliilor etc. Cu alte cuvinte, punctarea exercitiului se va realiza tinand cont atat de functionalitate cat si de arhitectura clasei.

Starea interna

Starea interna a obiectelor de tip PlayerStatus va contine:

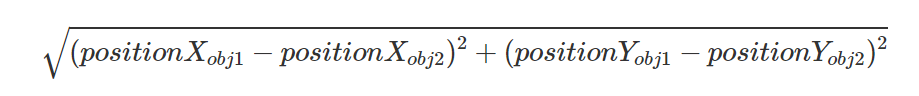
1. nickname: un sir de caractere, ce semnifica numele jucatorului
2. score: un numar intreg, ce reprezinta scorul jucatorului
3. lives: un numar intreg, ce simbolizeaza numarul de vieti al jucatorului
4. health: un numar intreg, intre 0 si 100, ce reprezinta procentul de viata ramas
5. weaponInHand: un sir de caractere ce simbolizeaza arma jucatorului
6. positionX: coordonata OX a pozitiei jucatorului, o valoare numerica reala
7. positionY: coordonata OY a pozitiei jucatorului, o valoare numerica reala
8. gameName: numele jocului.
   * toti jucatorii fac parte din acelasi joc. Prin urmare, valoarea campului va fi unica pentru toate obiectele.

Pe langa aceste campuri, poti adauga orice alte atribute pe care le consideri utile / necesare in implementarea functionalitatii cerute si in cadrul jocului.

Reguli de joc

Regulile generale de desfasurare a jocului sunt:

1. health:
   1. va lua valori intre 0 si 100
   2. daca health ajunge la 0 (sau o valoare negativa) in urma unei operatii, se va decrementa numarul de vieti (i.e. lives) cu 1 iar health va fi resetat la valoarea maxima, 100. Practic, jucatorul moare si se va mai consuma o viata din numarul de vieti ramase.
   3. daca health trece de valoarea 100 valoarea sa va fi trunchiata la 100. Numarul de vieti ramane nemodificat (i.e. nu se incrementeaza numarul de vieti).
2. daca lives ajunge la valoarea 0 atunci jocul s-a terminat, si se va afisa mesajul Game Over
3. weaponInHand arma curenta a jucatorului.
   1. poate avea una dintre urmatoarele valori:
      * knife - cost 1000 puncte,
      * sniper- cost 10000 puncte,
      * kalashnikov - cost 20000 puncte
   2. Obs: pentru a compara daca 2 siruri de caractere sunt identice, nu se va utiliza operatorul == ci metoda equals(String) apartinand clasei String. Consulta API-ul pentru detalii de utilizare.
   3. Obs2: pentru a achizitiona o arma, jucatorul trebuie sa aiba un scor minim egal cu valoarea armei. Dupa actualizarea armei, atributul score va fi actualizat in consecinta (i.e. score -= weaponPrice).
4. duelul intre 2 jucatori:
   1. daca cei doi jucatori au aceeasi arma in mana:
      * se va calcula pentru fiecare o probabilitate de castig astfel: (3 \* health + score / 1000) / 4.
      * jucatorul cu probabilitate mai mare va castiga lupta
   2. daca cei doi jucatori au arme diferite, puterea armelor depinde de distanta intre jucatori:
      * distanta > 1000 → puterea armelor descreste astfel: sniper > kalashnikov > knife
      * distanta ⇐ 1000 → puterea armelor descreste astfel: kalashnikov > sniper > knife
      * duelul va fi castigat de jucatorul cu o arma mai puternica, in functie de distanta dintre cei doi jucatori
5. distanta intre 2 jucatori:
   1. se calculeaza folosind formula distantei euclidiene, astfel:



Comportament

Comportamentul unui jucator va include:

1. 3 metode de initializare, cu diverse seturi de parametri:
   1. initPlayer(nickname): va initializa doar numele jucatorului (restul campurilor vor avea valoarea implicita a tipului.
   2. initPlayer(nickname, lives): va initializa campurile aferente: nickname respectiv lives(restul campurilor vor avea valoarea implicita a tipului.
   3. initPlayer(nickname, lives, score): comportamentul va fi unul asemanator, initializand campurile cu acelasi nume ca parametrii.
2. findArtifact(int artifactCode): metoda va actualiza starea jucatorului in functie de artifactul gasit, astfel:
   1. daca artifactCode este numar perfect:
      * score se incrementeaza cu 5000 puncte
      * lives creste cu 1 viata
      * health devine 100
   2. daca artifactCode este numar prim:
      * score se incrementeaza cu 1000 puncte
      * lives creste cu 2 vieti
      * health creste cu 25, dar nu poate depasi valoarea 100
   3. daca artifactCode este numar par si suma cifrelor sale este divizibila cu 3 atunci obiectul respectiv este o capcana. Prin urmare, situatia jucatorului se modifica astfel:
      * score se decrementeaza cu 3000 puncte
      * health se decrementeaza cu 25.
        + daca health ajunge la 0, atunci lives va fi decrementat cu 1 si health resetat la 100
   4. orice alte numere nu prezinta „puteri magice“, prin urmare, singura actualizare a starii este:
      * score va fi incrementat cu valoarea artifactCode
   5. Observatie: criteriile anterioare se vor testa pe rand (i.e. in ordinea data) si se va actualiza starea conform **primei conditii validate**.
      * De exemplu: daca un artifactCode este atat „numar perfect“ cat si „numar par avand suma cifrelor divizibila cu 3“ se va actualiza starea in conformitate cu „numar perfect“ deoarece acest criteriu precede pe celalalt.
3. setWeaponInHand(String weaponInHand): actiunea va schimba arma folosita de jucator in joc
   1. metoda va verifica initial daca arma primita ca argument este o arma valida, conform regulilor jocului
   2. se verifica apoi daca jucatorul are suficiente puncte acumulate in joc pentru a plati arma
   3. daca arma este valida si jucatorul are suficiente puncte pentru achizitie, arma va fi actualizata. Odata cu aceasta, se va actualiza si scorul, pentru a reflecta plata armei cumparate.
   4. metoda returneaza true daca schimbarea armei a avut loc cu succes si false in orice alt caz
4. getWeaponInHand(): returneaza arma curenta a jucatorului
5. transforma campurile positionX si positionY in proprietati ale obiectelor clasei PlayerStatus
6. implementeaza metodele necesare pentru ca atributul gameName sa devina o proprietate
   1. va fi acesta o proprietate a fiecarui obiect in parte, sau o proprietate a clasei
   2. implementeaza corespunzator metodele getter & setter
   3. oricine ar trebui sa aiba drept de accesare a proprietatii gameName, insa doar „codul“ apartinand aceluiasi pachet ar trebui sa aiba acces la actualizarea numelui jocului. Actualizeaza in consecinta modificatorii de acces a celor doua metode.
7. movePlayerTo(double positionX, double positionY)
   1. intuitiv, metoda actualizeaza pozitia jucatorului, prin actualizarea pozitiei acestuia
8. numele jucatorului nu ar trebui sa poata fi modificat din exterior, insa ar trebui sa poata fi citit. Cu alte cuvinte, transforma campul nickname intr-o proprietate **read-only** (i.e. care poate fi doar citita public)
   1. Hint: Care dintre cele doua metode speciale trebuie implementata in acest caz?
9. shouldAttackOpponent(PlayerStatus opponent): metoda simuleaza o batalie intre jucatorul curent si un adversar, pentru a sfatui jucatorul daca ar trebui sa atace (i.e. ar castiga o eventuala lupta)
   1. metoda returneaza true daca jucatorul curent ar castiga lupta si false altfel
   2. regulile dupa care se stabileste invingatorul sunt detaliate in sectiunea de Reguli Generale
   3. OOP-hint: utilizati principiul separarii logice a conceptelor implementate in toate metodele implementate
      * Exemplu: distanta dintre 2 jucatori ar trebui calculata intr-o metoda separata. Acea metoda trebuie apelata la nevoie, oriunde in cod. Deoarece metoda va fi folosita doar in cadrul implementarii clasei, deci intern, ce modificator de acces este cel mai adecvat?

Pe langa acestea, poti implementa orice alte metode (publice sau private) pe care le consideri necesare / utile in cadrul jocului sau in implementarea actiunilor detaliate anterior.