Sebastian Molin

Systemutveckling - Java

Laboration 2
Textbaserat spel

Implementation

I min implementering av detta textbaserade spel har jag skapat flera klasser som representerar olika aspekter av spelet, såsom Player, Maze, och CombatHandler. Jag kommer välja ut ett axplock med metoder och hur jag resonerat kring dem. I slutet gör jag även en analys om vilka typer av förbättringar och förändringar jag gjort under arbetets gång. Jag hade inte i åtanke att jag skulle skriva en rapport medans jag gjorde arbetet så jag kommer använda mina commits som stöd, vilket innebär att du även finner mycket av förbättringarna i mina commit messages.

Game

`Game`-klassen fungerar som huvudingång och hanterar spelets övergripande flöde, inklusive meny, spelstart, vilka Obstacles, Monsters och Items som ska “spawnas in” och vart. Här hanteras även användaren commands, move up, show stats etc. Några av de viktigaste metoderna är:

- `start()`: Denna metod hanterar spelets huvudsakliga loop och styr interaktionen mellan spelaren och spelet, som till exempel rörelser och kommandon genom två while loops. en start menu loop och en game loop som körs när menyn stängs och således menu = false;. Dessa två loopar består nästan helt av två switch statements
- `main()`: Startar spelet genom att skapa en instans av `Game` vilket också initialiserar nya instanser av Maze och Player och sen kör `start()`-metoden som jag nämnde ovan.

Player

Player-klassen hanterar spelarens stats såsom health, strength, defence osv. och deras interaktioner med spelet. Några av de centrala metoderna är:

- addItemToInventory(Item item): Denna metoden lägger till ett Item i spelarens inventory. Implementeringen av detta fungerar så här: I PlayerMovementHandler finns metoden handleItemPickup() som har fyra parametrar. två av dessa är x, y. Först deklareras Item item = maze.getItemAt(x, y). Sedan kommer det en if-sats, och om allt är okej så sker player.addItemToInventory(item);. På så sätt bestäms det om positionen har ett item, vilken typ av item det är och sedan läggs den till i inventory.
- move(int dx, int dy): Syftet med denna metod är att uppdatera spelarens position beroende på rörelseriktning. Den jobbar i samarbete med getX() och getY() som också finns i Player klassen. Men det är i PlayerMovementHandler som den används ordentligt. Där finns metoden movePlayer(int dx, int dy) som både ger player en ny koordinat (om isFreeSpace = true) men samtidigt kontrollerar ifall det sker någon typ av kollision med Obstacles eller Items.
- showStats(): Visar spelarens nuvarande stats, såsom health, strength, name och defence. Denna uppdateras i samband med att någon av dessa skulle ändras. Tex så används Player.takeDamage i CombatHandler. Vilket innebär att this.health -= damage; och health variabeln uppdateras direkt och ifall spelaren skriver show stats så kommer detta reflekteras korrekt.
- hasItem(String itemName): Denna är lite intressant då det är en metod med ett specifikt syfte, att kontrollera ifall spelaren har dragon gem i sitt inventory för att kunna trigga ett speciellt event vid combat. Den är väldigt simpel egentligen, den itererar över inventory och ifall den hittar gem så returnerar den true. Detta sker ej varje combat utan endast vid combat när monster.isDragon = true;

Namngivning och Parametrar

Jag har försökt att använda beskrivande namn på metoder och parametrar för att tydligt visa deras syfte och funktion. Till exempel använder jag addObstacles för att *lägga till* hinder, och movePlayer för att *flytta* spelaren. Parametrarna i dessa metoder är valda för att vara intuitiva men också generella för att kunna återanvändas så mycket som möjligt, till exempel dx och dy för att representera förändringar i spelarens x- och y-positioner och damage för att beskriva mängden skada ett monster gör. Men den bestämmer ej typ av skada tex, vilket gör den återanvändningsbar ifall man vill vidareutveckla Combat systemet.

Kopplingar mellan Klasser

Flera klasser i projektet känner till varandra, vilket är nödvändigt för att spelet ska fungera korrekt. Till exempel måste Maze-klassen känna till Player och ObstacleHandler för att kunna hantera spelarrörelser och väggar/monster. På samma sätt måste CombatHandler känna till både Player och Monster för att kunna hantera combat.

Det finns relativt många kopplingar mellan klasserna, jag har försökt minimera kopplingarna genom att använda handlers som ObstacleHandler och PlayerMovementHandler. Dessa klasser bidrar till att hålla huvudklasserna som Maze, Game och Player mer fokuserade och mindre beroende av andra klasser. En ytterligare förbättring skulle kunna vara att använda ett mer omfattande designmönster, jag har läst att det finns något som heter MVC som kanske skulle kunna vara relevant här för att ytterligare separera logiken och minska kopplingarna mellan klasser?

Min personliga uppfattning av vad god *seperation of concerns* innebär (med javascript som utgångspunkt) är att i ett vakuum så ska varje enskild metod gå att implementera "*as is*" med nya parametrar. Där är jag inte med denna uppgift, men jag tycker ändå jag vidtagit vissa steg för att förbättra den.