LINKÖPINGS UNIVERSITET

TDP005: Objektorienterat system

Designspecifikation

Författare

Kasper Nilsson, kasni325@student.liu.se
Andrei Plotoaga, andpl509@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2020
Version 1.0

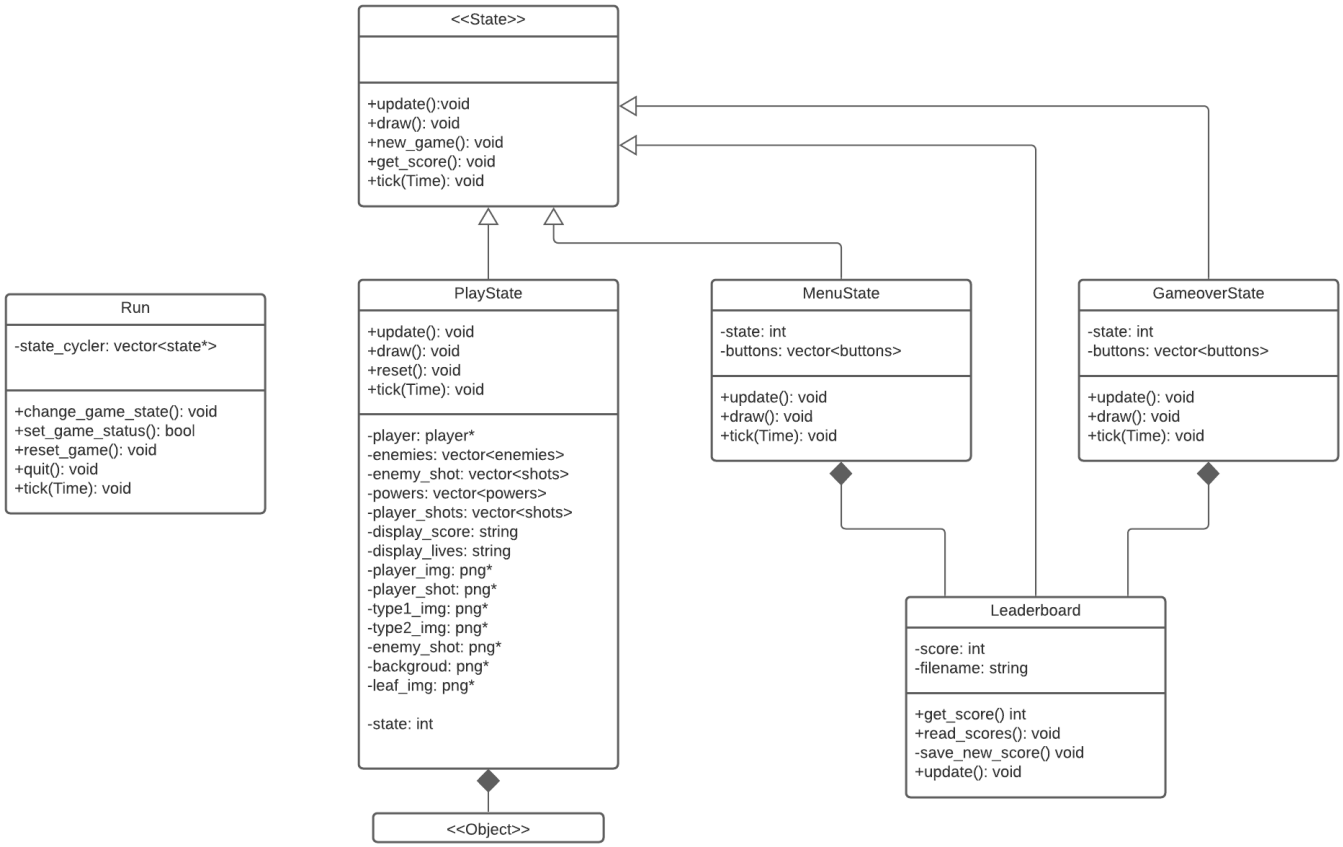
2020-11-27

1 Revisionshistorik

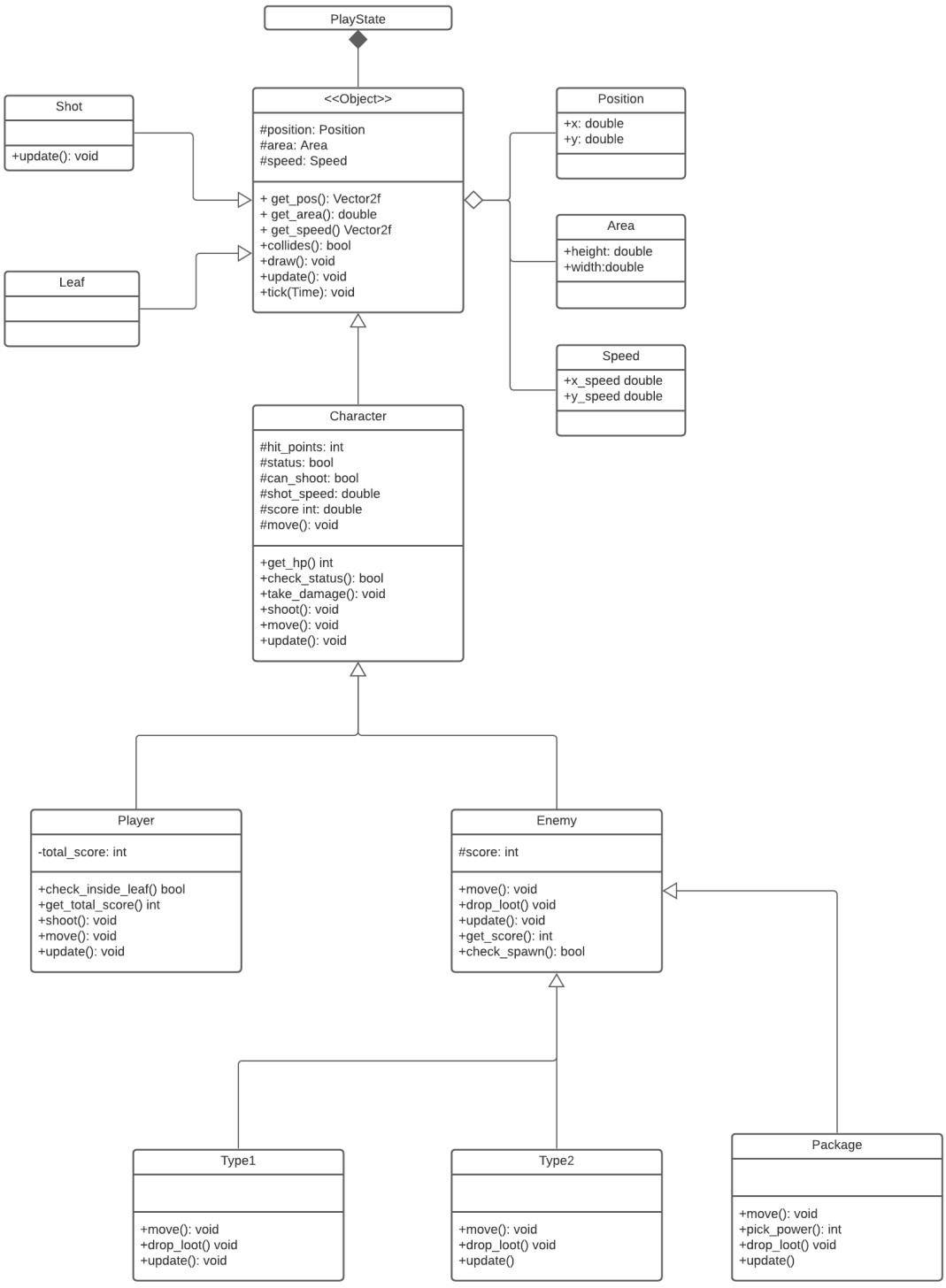
Ver.	Revisionsbeskrivning	Datum
1.0	Första utkast	27/11-20

2 Klassdiagram

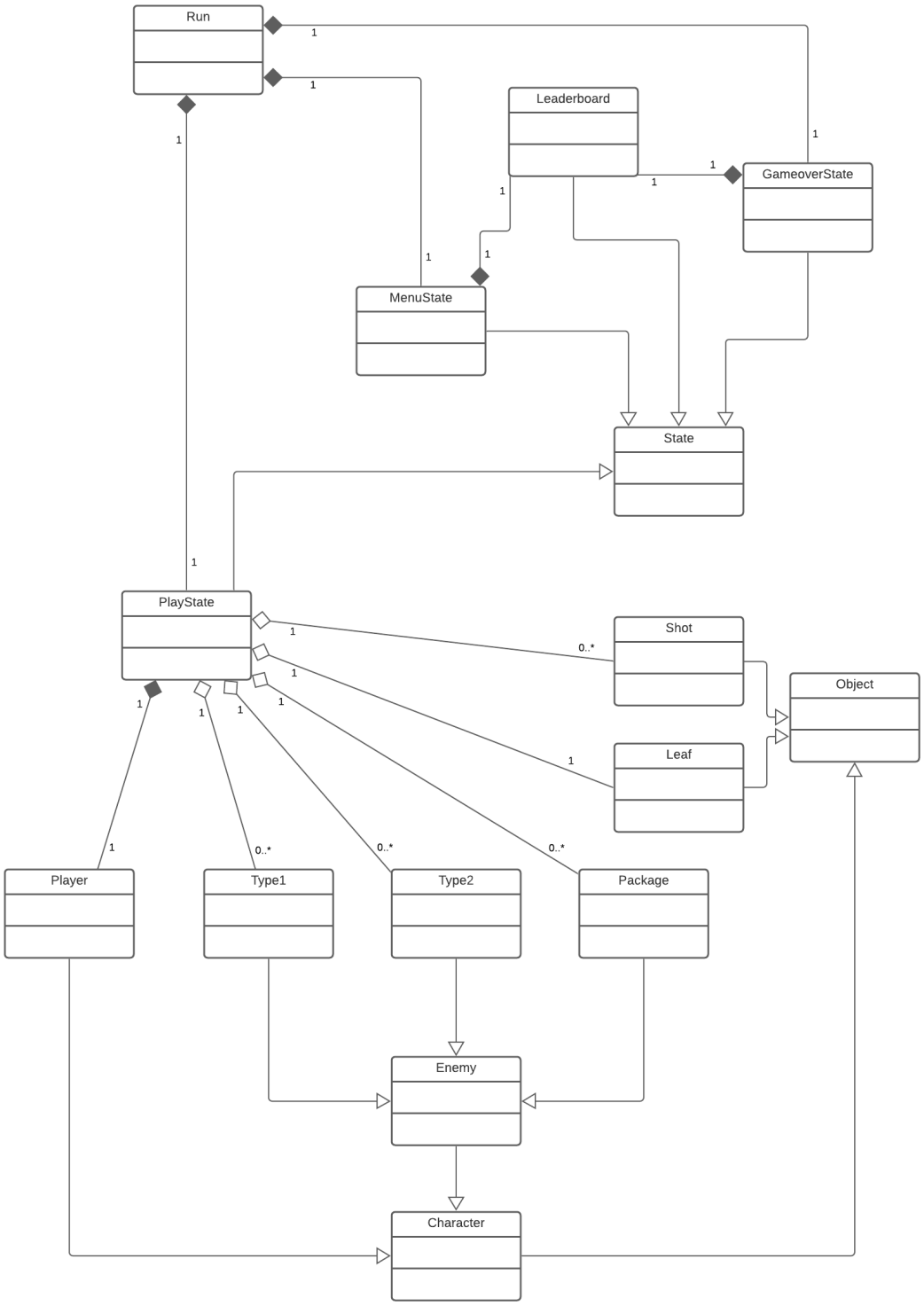
Figur 1: Klassdiagram med dess medlemmar del 1



Figur 2: Klassdiagram med dess medlemmar del 2



Figur 3: Klassdiagram med mulitplicitet



3 Detaljbeskrivning av två centrala klasser

3.1 Player

En instans av klassen Player är ett objekt som efterliknar en nyckelpiga och syftet med klassen är att användaren ska ha förmåga att kunna integrera med spelet. Ett objekt av Player ska kunna röra sig i upp, ned, vänster och höger med hjälp av piltangenterna. Notera att objektet inte kan rotera sig. Dessutom ska objektet kunna skjuta projektiler med mellanslag. Objektet kommer ha som medlemsvariabler en viss position i form av (x,y) koordinater, antal liv, rörelsehastighet i x- och y-led, eldhastighet i x- och y-led, totala poäng och status (död eller levande).

Objektet är begränsat att röra sig under en viss angiven yta, denna yta ska föreställa ett blad utritat på skrämen så det blir tydligt var gränserna för rörelse befinner sig.

Ett objekt av klassen Player kommer att integrera med objekt av klassen Shot för att kunna använda sin egna skjutfunktionaliet samt för att avgöra om fiender har träffat spelaren och vice versa. För att kunna detektera ifall spelaren kolliderar med objekt av någon av klasserna Enemy_Type_1 och Enemy_Type_2 behövs även där en sammankoppling. Det finns även kontroller för att se till att spelaren inte kan röra sig utanför den utritade bladytan via funktionen check_inside_lead(), så det behövs en sammankoppling mellan Lead klassen.

Det kommer enbart att finnas ett Player objekt per spel. Det innebär att det objektet måste sparas i en instans av Playstate som syftar till att erhålla samtliga spelobjekt som finns utritade på skärmen.

Följande är medlemsfunktioner som utmärker klassen Player som klassen internt kommer använda sig utav:

- **bool check_inside_leaf()** - kollar att spelaren befinner sig i på rörelseytan (bladets area).
- **get_total_score()** - uppdaterar spelaren totala poäng.
- **shoot()** - avfyrar projektil.
- **move()** - flyttar position.
- **update()** - kommer uppdatera position, och status (död eller levande) för spelaren.

Följande är intressanta medlemsfunktioner som Player kommer att ärva ifrån Character-klassen:

- **get_hp()** - returnerar spelarens nuvarnade liv.
- **check_status()** - kolla ifall spelaren har slut på liv, då avslutas spelet.
- **take_damage()** - minskar spelarens liv och ska orsaka en synlig reaktion att spelaren blivit träffad.

3.2 Playstate

En instans av klassen Playstate syftar till att innehålla det som ska ritas ut på spelplanen. Alltså denna klass ska innehålla ett spelar-objekt från Player klassen, flertal fiender-objekt från Type1/Type2 klasserna, ett löv-objekt från Leaf-klassen, samtliga projektiler som finns på spelplanen från klassen Shot, menyer som visar antal liv och antal poäng.

Följande är de mest intressanta interna medlemmar:

- **vector enemies** - är en vektor behållare av x,y coordonater av alla fiender. en av användningsområdena för detta är att det kan berätta för oss när banan inte har några fiender kvar, då kan vi avsluta spelet.
- **vector enemy_shot** - innehåller x- och y-koordinaterna för alla fiendens projektiler. På samma sätt finns det en annan vektorbehållare för spelarens skott.
- **reset()** - funktion som återställer alla värden i PlayState till deras standardstatus.

- **player** - pekaren till den aktuella spelaren

Följande är de mest intressanta medlemsfunktioner som PlayState kommer ärva från State:

- **tick()** - är ett sätt att hålla reda på tiden som har gått mellan händelserna.
- **draw()** - syftar till att rita ut de objekt som ska finnas på skärmen.
- **new_game()** - kommer starta ett nytt spel.
- **get_score()** - hämtar poäng topplistan från en textfil.
- **update()** - hanterar händelser så rätt logik flyter på i spelet.

4 Fördelar och nackdelar med designen

Det bör nämnas att denna design är i en tidig utvecklingsfas och kan komma att förändras. Vi upplever att designen inte är helt fullständig och ses mer som en grund för vidare påbyggnad allteftersom när vi kommer igång med realiseringen. Hur som helst så kan vi identifiera ett par fördelar som vi hoppas kan bestå sig, en av dessa är att det finns tydliga klasser som fyller en viss specifik funktionalitet. Alltså det upplevs som om abstraktionen är i rätt riktining.

När det kommer till förbättringsområden är en del att vissa klasser är till stor del beroende utav övriga klasser, exempelvis Playstate. En konsekvens av detta är ifall vi skulle vilja göra en ändring av en klass kan vi få oönskat beteende på annat håll i källkoden som vi dessutom kommer behöva ändra osv. Vi har försökt fundera ut alternativa lösningar till detta, men lyckas inte. En viss coupling kommer alltid att finnas, men vi strävar åt att hålla den så låg som möjligt. Ett annat förbättringsområde gäller ifall designen gör det svårt för oss att utöka spelet med nya banor. Det upplevs som om vår design kommer leda till en viss upprepning av kod, vilket inte är önskvärt.

5 Externa filformat

Spelet ska kunna visa spelarens främsta poäng vid avslutad spelsession (game-over eller vinst). Dessa presenteras i form av en top 10 lista. För att kunna spara undan poängresultatet kommer en txt-fil att användas, alltså efter avslutat spelsession ska poängresultatet jämföras mot de som redan finns angivna i denna textfil. Om spelaren lyckas slå någon av sina tidigare topresultat kommer listan att uppdateras allteftersom. Nackdel med att använda en txt-fil för detta är att användaren kommer kunna ha möjlighet att skriva i txt-filen som möjligtvis skulle kunna leda till inläsningsfel för spelet eller fusk.

För textur av diverse objekt används png-filer.