

TDP005 Projekt: Objektorienterat system

Designspecifikation

Författare

Jakob Norberg, jakno825@student.liu.se

Linus Sachs, linsa177@student.liu.se

Elias Roos, eliro825@student.liu.se

Thomas Sjödin, thosj050@student.liu.se

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första utkast	29/11-24
1.1	Uppdaterat allt	05/12-24
1.2	Uppdaterat utefter slutprodukten	18/12-24
1.3	Uppdaterat designval	6/1-25



MovableThing ärver i sin tur av basklassen Entity som samlar de gemensamma funktionerna för alla objekt som ska finnas i spelet. Figur 1 visar hur Player och Enemy ärver från MovableThing som i sin tur ärver från Entity.

Player tar emot 3 parametrar i dess konstruktor. Dessa parametrar är:

- en `sf::Vector2f` som representerar positionen där objektet ska skapas
- en `std::string` som anger objektets namn (t.ex. `player`)
- och en `sf::IntRect` som specificerar vilken del av karaktärens spritesheet som ska användas

Konstruktor skickar dessa parametrar till `MovableThing`'s konstruktor som i sin tur skickar dem till `Entity`. I `Entity` hämtas texturen från en map där alla texturer ligger sparade, mapen är uppbyggd av strängar som nycklar och texturer som värden. Det gör det möjligt för `Player` att, med hjälp av `std::string`, hämta rätt textur. `Player`'s konstruktor initierar också en del variabler som är specifika för `Player`. De variabler som är specifika för `Player`-klassen listas i tabell 2.

För att sammanfatta strukturen så skapar detta en struktur som är lättläst och enkel att vidareutveckla.

Funktioner	Åtgärd
<code>void update()</code>	Uppdaterar spelarens rörelse, gravitation och animation
<code>void addHealth()</code>	Specifik funktion för spelaren då spelaren är det enda objekt som ska kunna få liv
<code>void handleCollision()</code>	Hanterar kollision mellan spelaren och fiender
<code>sf::RectangleShape getHitbox()</code>	Hämtar spelarens hitbox för kollision
<code>void resetCooldown()</code>	Nollställer cooldown för när en attack kan ske
<code>void updateHitbox()</code>	Uppdaterar området där spelaren kan kollidera, fixerar den på spelaren
<code>bool canAttack()</code>	Returnerar true ifall spelaren kan attackera
<code>void movement()</code>	Styr spelarens rörelse
<code>void changeVelocity()</code>	Ändrar spelarens hastighet
<code>sf::Vector2f getVelocity()</code>	Returnerar spelarens hastighet
<code>void changeJump()</code>	Ändrar så spelaren kan hoppa igen

Tabell 1: Player-funktioner

Variabler	Åtgärd
<code>int speed</code>	Spelarens hastighet
<code>sf::RectangleShape playerHitbox</code>	Spelarens kollisionsruta
<code>float reductionFactorX</code>	Justeringsfaktor för X-axeln för detektionsboxen
<code>float reductionFactorY</code>	Justeringsfaktor för Y-axeln för detektionsboxen
<code>float lastAttack</code>	Tid sedan senaste attacken
<code>float attackCooldown</code>	Hur lång tid spelaren inte kan attackera på
<code>float jumpHeight</code>	Höjden spelaren kan hoppa
<code>Animation animation</code>	Animation för spelaren
<code>int collisionDamage</code>	Skada spelaren tar vid kollision
<code>int direction</code>	Vilket håll spelaren går åt
<code>bool canJump</code>	Säger om spelaren kan hoppa eller inte
<code>sf::Vector2f velocity</code>	Spelarens rörelsehastighet

Tabell 2: Player variabler

2.1 MovableThing

Syftet med klassen är att fungera som en gemensam bas för alla objekt som kan röra sig, har hälsa och som kan ge eller ta skada.

Konstruktorn för klassen tar emot tre parametrar:

- en `sf::Vector2f` som representerar positionen där objektet ska skapas
- en `std::string` som anger objektets namn (t.ex. `player`)
- och en `sf::IntRect` som specificerar vilken del av karaktärens spritesheet som ska användas

Dessa parametrar skickas vidare till basklassen `Entity`, som ansvarar för att initiera dem. Konstruktorn för `MovableThing` initierar även ett antal medlemsvariabler som kan användas av subclasserna. Dessa variabler beskrivs mer detaljerat i tabell 4.

Från `MovableThing`-klassen ärver `Player` följande funktioner och variabler:

Funktioner	Åtgärd
<code>void removeHealth()</code>	Tar bort health för spelaren
<code>bool checkDeath()</code>	Returnerar true ifall spelaren är död
<code>bool canAttack()</code>	Returnerar true ifall spelaren kan attackera

Tabell 3: `MovableThing`-funktioner

Variabler	Åtgärd
<code>bool isDead</code>	False om spelaren lever
<code>int health</code>	Spelarens liv
<code>float fallingspeed</code>	Hur snabbt spelaren faller
<code>float gravity</code>	Gravitation
<code>float lastAttack</code>	Tid sedan senaste attacken
<code>float attackCooldown</code>	Hur lång tid spelaren inte kan attackera på

Tabell 4: `Player` variabler

2.2 Entity

Entity-klassen är basklassen för alla objekt i spelet. Den samlar de egenskaper som är gemensamma för allt i spelet, såsom position och textur.

Konstruktorn för klassen tar emot tre parametrar:

- en `sf::Vector2f` som representerar positionen där objektet ska skapas
- en `std::string` som anger objektets namn (t.ex. "player")
- och en `sf::IntRect` som specificerar vilken del av karaktärens spritesheet som ska användas

Entity initierar dessa värden baserat på de parametrar som skickas in. Entity har även en `std::map` där objekten kopplas ihop med rätt textur.

Från Entity-klassen ärver Player följande variabler:

Från Entity-klassen ärver Player följande funktioner och variabler:

Funktioner	Åtgärd
<code>void render()</code>	Ritar ut spelaren
<code>void setPosition()</code>	Placerar ut spelaren på en given position
<code>sf::FloatRect getGlobalBounds()</code>	Returnerar spelarens hitbox
<code>sf::Vector2f getPosition()</code>	Returnerar spelarens position i x- och y-led
<code>sf::Texture</code>	<code>getTextures()</code>
Returnerar spelarens textur	

Tabell 5: Entity-funktioner

3 Detaljb beskrivning av GameState

Syftet med GameState är att kalla alla funktioner som får spelet att köra. GameState kallar funktioner från andra klasser för att uppdatera spelplanen, läsa in objekt och kalla klasserna gameover/youWin när spelet är över. GameState ärver från klassen state men har själv inga underklasser.

Funktioner	Åtgärd
void update()	Tillkallar alla relevanta objekt, override vid en annan instans av spelet
void render()	Ritar ut textur, override vid en annan instans av spelet
void checkCollition()	Kontrollerar kollisionsdetektering mellan objekt.
void checkEnemyCollision()	Kontrollerar kollision mellan fiender och spelaren.
void checkPlayerAttack()	Kontrollerar spelarens attack och eventuella träffar.
void spawnObjects()	Skapar och placerar objekt på spelkartan.
void removeEnemies()	Tar bort fiender som har besegrats.
void removeEnemies()	Tar bort fiender som har besegrats.
void checkDetection()	Kontrollerar om spelaren har blivit upptäckt av fiender.
void checkSwordAttack()	Kontrollerar spelarens svärdattack.
void spawnArrow()	Skapar en pil vid en viss position.
void spawnArrow()	Skapar en pil vid en viss position.
void checkArrowCollision()	Kontrollerar kollisioner mellan pilar och objekt.
void checkForDeath()	Kontrollerar om spelaren eller fiender har dött.
void initTileMap()	Initialiserar objektkartan som innehåller spelobjekt.
void overLap()	Kontrollerar om sprite hitbox överlappar med tiles.
void initBackground();	Initierar bakgrundsbilden för gamestate.

Tabell 6: GameState-funktioner

Variabler	Åtgärd
Player player	Spelarens objekt.
Fountain fountain	Fontän-objekt i spelvärlden.
Door door	Dörr-objekt i spelvärlden.
MeleeAttack sword	Spelarens svärdsattack
std::vector<std::unique_ptr < Enemy >> enemies	Vektor med unika pekare till fiender
std::vector<std::unique_ptr < Spikes >> spike	Vektor med unika pekare till spikar.
std::vector<std::unique_ptr < Projectile >> arrows	Vektor med unika pekare till pilar
sf::RectangleShape background	Bakgrundsbild för spelvärlden.
sf::Texture bgTexture	Textur som används för bakgrunden.
TileMap map	Spelkartan.
bool isDead	Indikerar om spelaren är död.

Tabell 7: GameState-variabler

4 Designval

Vi är nöjda med hur vi har strukturerat klasserna i vårt spel. Varje klass har ett tydligt syfte, vilket gör det enkelt att navigera och förstå koden. Den tydliga uppdelningen minskar risken för att ändringar i en klass påverkar andra delar av systemet, vilket bidrar till att hålla koden organiserad och lätt att underhålla.

En av styrkorna med vår design är hur basklasser används för att samla gemensamma funktioner som delas av flera typer av objekt. Detta gör det enkelt att lägga till nya objekt, exempelvis en ny fiendetyp, utan att behöva modifiera basklasserna. Genom att undvika kodupprepning och hålla funktionalitet gemensam för flera objekt i basklasserna, blir systemet flexibelt och enkelt att bygga ut.

Dock har vi identifierat en viktig svaghet i vår nuvarande design: GameState-klassen bär ett för stort ansvar. I början av projektet ansvarade den inte bara för alla objekt i spelet, utan även för kollisioner och spelmekanik som är kopplade till dessa. Detta gör GameState komplex och svår att arbeta med, särskilt när vi vill lägga till nya funktioner eller fler objekt.

Vi insåg att ett bättre sätt att lösa detta är att dra mer nytta av polymorfism och flytta ansvaret för logiken från GameState till spelobjekten själva. I stället för att GameState direkt hanterar varje objekts kollisioner eller specifika beteenden, kan varje objekt implementera en `update()`-funktion för att hantera sin egen logik. GameState behöver då bara iterera över alla objekt och anropa deras `update()`-metoder. Detta tillvägagångssätt minskar ansvaret för GameState, vilket gör klassen mer hanterbar och låter varje objekt sköta sin egen logik. På så sätt sprids inte ansvaret för spelmekaniken över flera filer utan samlas i de klasser som representerar objekten på spelplanen. Detta ger en tydligare separation av ansvar och gör systemet mer modulärt och lättare att bygga vidare på.

Sammanfattningsvis tycker vi att vår design med tydlig klassuppdelning och basklasser är en styrka som gör koden flexibel och underhållbar. Genom att integrera mer polymorfism och flytta ansvaret för logiken till objekten själva ser vi också en möjlighet att adressera problemet med GameState och skapa en mer skalbar och hanterbar arkitektur.

5 Extern filformat

Vår plan är att använda oss av en txt-fil för att skapa banorna. Tanken är att vi ska ladda in våra banor från en extern fil, detta ökar flexibiliteten och formbarheten av banorna man skapar. Eftersom vi har jobbat med att ladda in txt-filer tidigare så är det den typ av filer vi känner oss bekvämast med.

Vi är dock medvetna om att txt-filer kan vara begränsande ifall spelet vidareutvecklas och blir mer komplext, då txt-filer är svårlästa och allmänt krångliga att jobba med. Men för de aktuella kraven vi har på spelet anser vi att txt-filer räcker för att uppfylla detta.