Projektbeskrivning

Tower Defence

2018-03-06

Projektmedlemmar:
Oscar Sandberg <oscsa949@student.liu.se>
Arvid Sundqvist <arvsu170@student.liu.se>

Handledare:
Teodor Riddarhaage <teori199@ida.liu.se>

1. Introduktion till projektet 3

2. Ytterligare bakgrundsinformation 3

3. Milstolpar 3

4. Övriga implementationsförberedelser 5

5. Utveckling och samarbete 6

6. Implementationsbeskrivning 6

6.1. Milstolpar 6

6.2. Dokumentation för programkod, inklusive UML-diagram 8

Övergripande programstruktur 8

Översikter över relaterade klasser 8

Programmet är uppdelat i tre delar: Grafik, spelmekanik och entiteter. Deras sammanhang förklaras mer ingående i varje del. 8

Grafik 8

Spelmekanik 8

Entiteter 9

HealthEntity 10

6.3. Användning av fritt material 11

6.4. Användning av objektorientering 11

Objekt/klasser 11

Typhierarkier och Subtypspolymorfism 12

Ärvning med overriding 12

6.5. Motiverade designbeslut med alternativ 13

Vi har tagit flera designbeslut under projektets gång som diskuteras och motiveras nedanför. 13

Designbeslut 1: Abstrakta klasser 13

Designbeslut 2: Slopning av Enum-klasser 13

Designbeslut 3: Expandera tick till Entity 14

Designbeslut 4: Uppdelandet av grafik, spelmekanik och entiteter. 14

Designbeslut 5: Spelplan och bakgrund 14

Designbeslut 6: Levels 14

Designbeslut 7: Alternativ till tick 15

Designbeslut 8: Fienders rörelse mot basen 15

7. Användarmanual 16

8. Slutgiltiga betygsambitioner 19

9. Utvärdering och erfarenheter 19

1. Introduktion till projektet

Under projektet ska vi utveckla ett Tower Defence-spel och har valt att utgå från inspirationsprojekt Tower Defence.

Första Tower Defence-spelet utvecklades av Atari 1990 och har därefter varit ett populärt tema på spel med många framgångsrika spel inom genren. Tower Defence går ut på att fiender rör sig genom en bana mot ett mål samtidigt som torn byggs upp av spelaren och skjuter fienderna. Genren kan delas upp i två kategorier.

- Fienderna rör sig på en väg och spelaren bygger torn utmed vägen som skjuter ner fienderna. Tornen kan inte placeras på vägen utan endast bredvid.
- Fienderna rör sig från ena sidan av spelplanen till den motsatta sidan och spelaren kan fritt bygga torn över spelplanen och tornen skjuter ner fienderna. I den här kategorien kan inte fienderna röra sig igenom torn.

2. Ytterligare bakgrundsinformation

För att vårt Tower Defence-spel ska fungera måste vissa grundregler finnas.

- En sida av spelplanen måste skyddas från fiender av spelaren.
- Fienderna kommer i rundor, där varje runda blir svårare för spelaren att klara och försvara sin sida.
- Placering av torn eller hinder kan placeras överallt på spelplanen, men inte på varandra.

Se mer information på Wikipedias artikel: https://en.wikipedia.org/wiki/Tower_defense

3. Milstolpar

#	Beskrivning
1	Skapa tom spelplan med ett koordinatsystem. Spelplanen består av en enum typ “ground” med möjlighet att lägga till fler i senare steg.
2	En grafisk komponent kan visa spelplanen. Det går att stänga fönstret genom att

	klicka på kryssknappen. En testklass skapas för att kunna testa spelet.
3	Skapa en ny enum klass för fiender samt egna typer för varje fiende. Vi börjar med en enkel för att sedan lägga till fler. Varje fiende har olika beteenden och därför olika klasser. Använder en klass för att skapa alla fiender.
4	Fiender får ett utseende och visas på spelplanen.
5	Fiender spawnar på x = 0 och y = random, de rör sig mot spelplanens bredd.
6	Skapa en ny enum klass för torn samt egna typer för varje torn. Vi börjar med en enkel för att sedan lägga till fler. Varje torn har olika beteenden och därför olika klasser. Använder en klass för att skapa alla torn.
7	Torn får ett utseende och visas på spelplanen.
8	Fiender får HP för att de ska kunna dö.
9	Implementera så att torn kan göra skada på fiender.
10	Lägga till så att fiender kan dö och då försvinna från spelplanen.
11	Torn får HP för att de ska kunna dö.
12	Implementera så att fiender kan göra skada på torn.
13	Lägga till så att torn kan dö och då försvinna från spelplanen.
14	Skapa “bas” som fienderna rör sig mot, har HP.
15	Implementera Game Over för spelet. MVP uppnådd.
16	UI Bar som kommer visa t.ex. stats och shop för torn.
17	Möjlighet för spelaren att köpa torn. Gratis för tillfället.
18	Placera torn m.h.a muspekaren på spelplanen.
19	När ett torn dödar en fiende erhåller spelaren pengar som kommer kunna användas för att köpa fler torn.
20	Torn får ett pris som begränsar spelaren från att kunna köpa oändligt många torn.
21	Levels. Vi skapar så att fienden spawnar i omgångar och varje omgång/level blir svårare och svårare.
22	Vi skapar olika typer av fiender som har olika beetenden och egenskaper, men som delar samma gränssnitt.
23	Vi skapar olika typer av torn som har olika beetenden och egenskaper, men som delar samma gränssnitt.

24	Vi skapar topplista i spelet.
25	Man kan köpa hinder och placera ut dessa på spelplanen.
26	Möjlighet att uppgradera tornen spelaren äger, vilket förbättrar tornets egenskaper.
27	En pathfinder läggs till för fiender för att de ska kunna gå runt vissa hinder och liknande.
28	Ljud adderas till spelet. Spel när torn skjuter, när fiender skadar torn eller basen, samt bakgrundsmusik.
29	Powerupgränssnitt skapas för att kunna addera powerups enklare.
30	Powerup nuke. Lägga till en powerup som tar bort alla fiender på hela spelplanen.
31	Powerup speedDown/freeze. Lägga till en powerup som gör att alla fiender går långsammare/stannar.
32	Powerup damageUp. Lägga till en powerup som ökar skadan för alla torn.
33	Powerup moneyUp. Lägga till en powerup som ökar pengarna som erhålls från varje dödad fiende.
34	Powerup scoreUp. Lägga till en powerup som ökar poängen.
35	Powerdown-gränssnitt skapas för att kunna addera powerdowns enklare
36	Powerdown speedUp. Lägga till en powerdown som ökar fiendernas hastighet.
37	Powerdown damageDown. Lägga till en powerdown som minskar tornens skada på fiender.
38	Svårighetsgrad - Lätt/Medel/Svår : Ju svårare dess då fler fiender med mer HP och snabbare spawnar.
39	Startmeny där man kan välja svårighetsgrad och diverse inställningar.
40	Vi adderar fler block så att spelplanen kan se ut på fler sätt och få olika egenskaper.
41	Implementera en MapMaker som gör det möjligt att göra egna banor.
42	Addera ett option i startmenyn att välja MapMaker.

4. Övriga implementationsförberedelser

Block (Spelplanen): Spelplanen består av block som är olika enum types.

Fiender: Det finns ett gränssnitt och olika typer av fiende tillhör olika klasser som implementerar gränssnittet. Fienderna attackerar torn eller hinder om de kolliderar.

Torn: Det finns ett gränssnitt och varje olika typ av torn är en klass som implementerar gränssnittet. Tornet väljer ut en fiende som den skjuter.

Ljud: Egen klass

Powerup: Vi skapar ett gränssnitt som varje unik powerup är en klass som implementerar gränssnittet.

Powerdown: Vi skapar ett gränssnitt som varje unik powerdown är en klass som implementerar gränssnittet.

Svårighet: En enumklass med olika enum types.

Pathfinder: Egen klass.

Highscore: En highscore-klass som skapar highscore-objekt och ytterligare en klass som skapar en lista av highscores.

Startmeny: Samma frame som spelplanen.

5. Utveckling och samarbete

- Vi arbetar på ett agilt arbetssätt med små milstolpar för att enkelt kunna stämma av hur bra en implementering är. Koden ska ha tydliga kommenterar och varje person ska kolla igenom den andras kod för att förstå för all kod i projektet. Det ger möjlighet till feedback och för att kunna förbättra koden. Vi använder Gitlab för att kunna jobba enskilt och kunna versionshantera projektet. Målet är att varje milstolpe ska vi arbeta från en separat gren i Gitlab som slås ihop med den ursprungliga koden när all kod i den separata grenen fungerar som den ska.
- Liknande milstolpar delas upp för att öka effektiviteten samtidigt som båda får en bra förståelse för koden då koden kommer fungera på snarlikt sätt. Med milstolpar som är mer komplicerade kommer arbetet ske tillsammans för att båda ska få en bra förståelse samtidigt som vi kan bolla ideer och lösningar.
- Arbetet på projektet kommer ske på schemalagda tider, kvällar och om det behövs på helger.
- Vi satsar på betyget 5.

6. Implementationsbeskrivning

I detta avsnitt redovisas uppnådda milstolpar, dokumentation av programkod, vilket material som använts, hur objektorientering använts samt vilka designbeslut som tagits under projektets gång.

6.1. Milstolpar

Vi har implementerat alla milstolpar i samma ordning som i projektbeskrivningen.

1. Milstolpen är implementerad. Under projektets gång ändrade vi hur spelplanen skulle

vara uppbyggd. Istället för en spelplan som är uppbyggd av små kvadrater är den uppbyggd av en stor bild och koordinatsystemet består av pixlar. Motivering till ändringen kan ses i avsnitt 6.5.

2. Milstolpen är implementerad. Grafisk komponent och en testklass, Launcher, finns i projektet för att visa spelplanen och testa spelet.
3. Milstolpen är delvis implementerad. Under projektet ändrade vi uppbyggnaden av fiender, motivering till ändringen kan ses i avsnitt 6.5.
4. Milstolpen är implementerad. Fiender får ett utseende och kan visas på spelplanen.
5. Milstolpen är implementerad. Fienderna spawnar längst till vänster på spelplanen (x=0) och random y och rör sig mot spelplanens högra sida.
6. Milstolpen är delvis implementerad. Under projektet ändrade vi uppbyggnaden av torn, motivering till ändringen kan ses i avsnitt 6.5.
7. Milstolpen är implementerad. Torn får ett utseende och kan visas på spelplanen.
8. Milstolpen är implementerad. Fiender får HP och kan dö.
9. Milstolpen är implementerad. Torn kan skada fiender.
10. Milstolpen är implementerad. Fiender försvinner från spelet när de dör.
11. Milstolpen är implementerad. Torn får HP och kan dö.
12. Milstolpen är implementerad. Fiender kan skada torn.
13. Milstolpen är implementerad. Torn försvinner från spelet när de dör.
14. Milstolpen är implementerad. Spelet får en bas som ska förstöras av fienderna.
15. Milstolpen är implementerad. Game over när basen har slut på HP.
16. Milstolpen är delvis implementerad. UI bar för stats och shop.Motivering kan ses i avsnitt 6.5.
17. Milstolpen är implementerad. Lägga till fler torn (gratis).
18. Milstolpen är implementerad. Placering av torn.
19. Milstolpen är implementerad. Reward när fiender dödas.
20. Milstolpen är implementerad. Torn får ett pris.
21. Milstolpen är delvis implementerad. Spelet blir svårare och svårare. Motivering kan ses i avsnitt 6.5.
22. Milstolpen är implementerad, men utan gränssnitt. Finns olika fiender med olika beteenden och egenskaper. Motivering kan ses i avsnitt 6.5.
23. Milstolpen är implementerad, men utan gränssnitt. Finns olika torn med olika beteenden och egenskaper. Motivering kan ses i avsnitt 6.5.
24. Milstolpen är implementerad. Highscore
25. Milstolpen är implementerad. Spelare kan köpa hinder.
26. Milstolpen är inte implementerad.
27. Milstolpen är inte implementerad.
28. Milstolpen är inte implementerad.
29. Milstolpen är inte implementerad.
30. Milstolpen är inte implementerad.
31. Milstolpen är inte implementerad.
32. Milstolpen är inte implementerad.
33. Milstolpen är inte implementerad.
34. Milstolpen är inte implementerad.
35. Milstolpen är inte implementerad.
36. Milstolpen är inte implementerad.
37. Milstolpen är inte implementerad.
38. Milstolpen är inte implementerad.
39. Milstolpen är inte implementerad.

40. Milstolpen är inte implementerad.
41. Milstolpen är inte implementerad.
42. Milstolpen är inte implementerad.

6.2. Dokumentation för programkod, inklusive UML-diagram

Övergripande programstruktur

Spelet drivs framåt med hjälp av en timermetod *GameClock* som tickar ca 60 gånger per sekund. I varje tick kallas huvudklassen *Game's* egen tick funktion som hanterar nödvändig funktionalitet. Alla entiteter (förklaras mer utförligt senare) har också en tick-funktion, detta innebär att de är relativt självständiga efter de har skapats. I slutet av varje tick i huvudklassen *Game* uppdateras även grafiken som ritas ut på skärmen.

Översikter över relaterade klasser

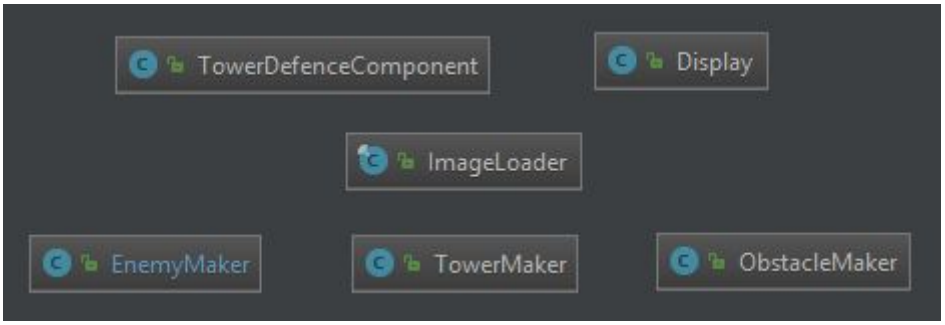
Programmet är uppdelat i tre delar: Grafik, spelmekanik och entiteter. Deras sammanhang förklaras mer ingående i varje del.

Grafik

De klasser som har hand om grafiken följer nedan, en grafisk representation av deras struktur kan ses i UML-Diagram 1.

- *Display*
  - Skapar fönstret där alla grafiska komponenter finns.
  - En menubar för köp av torn och hinder.
  - Hanterar user input.
- *TowerDefenceComponent*
  - Ritar ut alla entiteter med HP-bar
- *ImageLoader*
  - Laddar in texturer från resursmappen med hjälp av en path i String format.

Grafiken hanteras även delvis av maker-klasserna som tillhandahåller information om var texturresurserna finns.



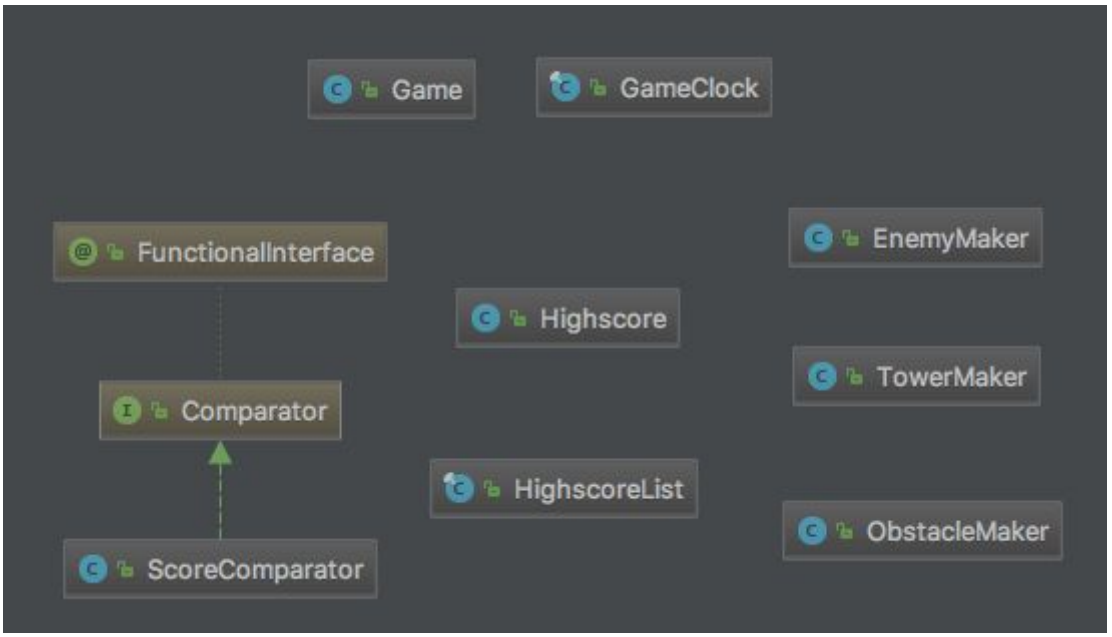
UML-Diagram 1, illustrerar klasshierarki för de grafiska komponenterna.

Spelmekanik

Klasser som hanterar spelmekaniken följer nedan, en grafisk representation av deras struktur kan ses i UML-Diagram 2.

- *GameClock*
  - Har en timer som kallar på tick-funktionen i *Game* ca 60 gånger i sekunden.

- *Game*
  - Den huvudsakliga tick-funktionen.
    - Hanterar entiteter , poäng, pengar och multipliers som skalar spelets svårighetsgrad under dess gång.
    - Spawnar fiender.
    - Kallar på alla entiteter egen tick-funktion. Detta låter dem vara mer självgående.
  - Hanterar skapandet av nya torn, fiender och hinder och håller koll på de existerande.
- *Highscore*
  - Tar hand om poänghantering.
  - *ScoreComparator*
    - Jämför highscores.
  - *HighscoreList*
    - Förvarar highscores i en lista.
- Entity makers
  - Makers "tillverkar" de olika fiender, torn och hinder åt de funktioner som behöver arbeta med dessa för att antingen kolla kollisioner eller placera på spelplanen.
  - Kan lätt användas för att skapa fler och nya entiteter .
  - *TowerMaker*
  - *EnemyMaker*
  - *ObstacleMaker*



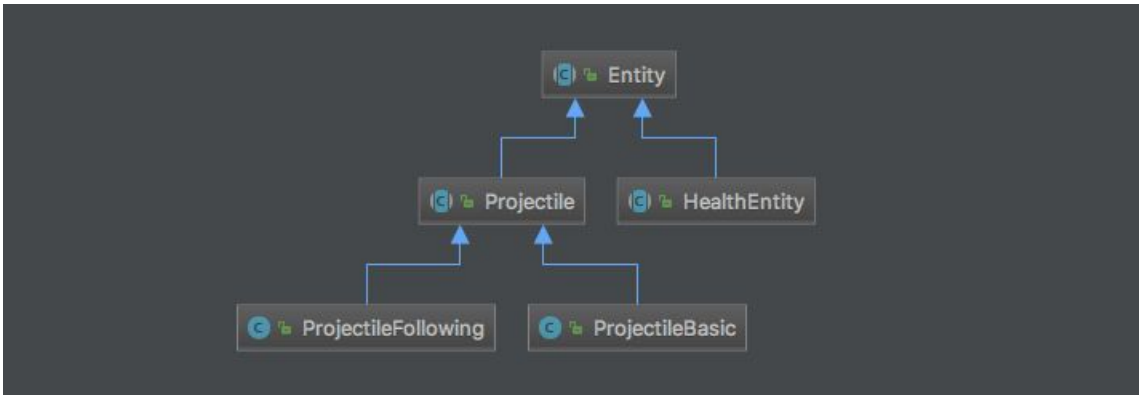
UML-Diagram 2, illustrerar klasshierarki för spelmekanik.

Entiteter

Enitity är en superklass för alla objekt som ritas ut på spelplanen, dessa har en bild, x- och y-position samt kan påverka spelet samt metoder för dessa. I UML-Diagram 3 visas de subklasser som ärver från Entity.

- *HealthEntity*
  HealthEntity är en underklass till *Entity* och hanterar alla objekt som har liv som torn, fiender och hinder. I klassen finns metoder som är gemensamma för alla dessa objekt.
  - Utritning av HP-bar
  - Metod för att kolla efter kollisioner.

- *Projectile*
  Projectile är en abstrakt klass till *Entity* för alla projektiler som skjuts från tornen.
  - Metod för att skada fiender.
  - Metod för målsökande projektilers rörelse.
  - Metod för att att kolla efter kollision.
    - *ProjectileBasic*
      ProjectileBasic är en underklass till *Projectile* och hanterar egenskaperna för objekt inom klassen.
      - Metod för icke målsökande projektilers rörelse.
    - *ProjectileFollowing*
      ProjectileFollowing är en underklass till *Projectile* och hanterar egenskaperna för objekt inom klassen.
      - Ärver samtliga metoder från *Projectile*.



UML-Diagram 3, en grafisk representation av Entity's klasshierarki.

HealthEntity

HealthEntity är en underklass till *Entity* och hanterar alla objekt som har liv som torn, fiender och hinder. I klassen finns metoder som är gemensamma för alla dessa objekt. Visuell representation av hierarkin kan ses i UML-Diagram 4.

- *Enemy*
  Enemy är en abstrakt klass till *HealthEntity* och hanterar gemensamma fält och metoder för alla fiender i spelet.
  - Metod för kollision för att testa kollision med torn och hinder.
  - Attackera torn.
  - Grundläggande rörelse
    - *EnemyBasic*
      EnemyBasic är en underklass till *Enemy* och hanterar egenskaperna och beteendet för fienden Enemy Basic.
    - *EnemySeeker*
      EnemySeeker är en underklass till *Enemy* och hanterar egenskaperna och beteendet för fienden Enemy Seeker.
      - Metod för att hitta närmaste torn.
      - Metod för att röra sig mot närmaste torn.

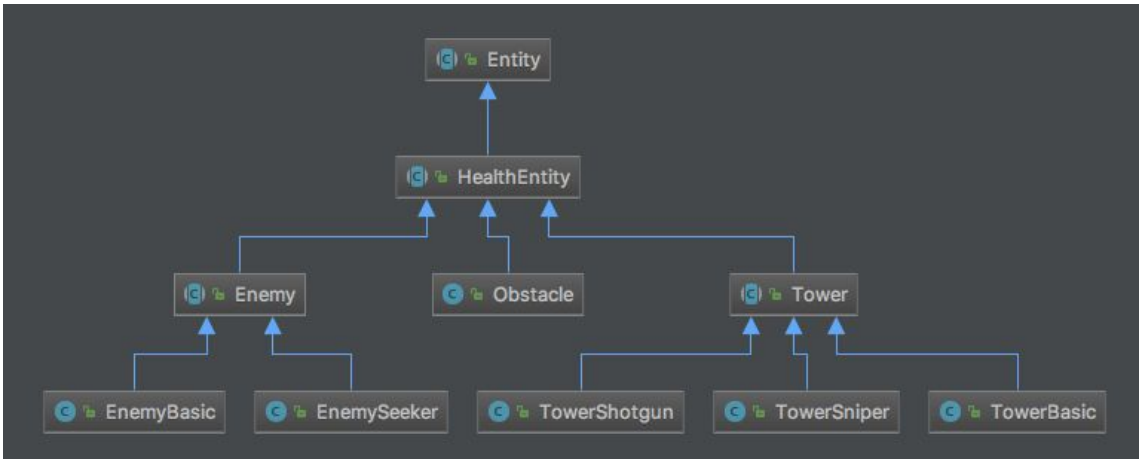
- *Tower*
  Tower är en abstrakt klass till *HealthEntity* och hanterar gemensamma fält och metoder för alla torn i spelet.
  - Metod för att skjuta fiender.
  - Metod för att ta bort torn som dött och basen.
    - *TowerBasic*
      TowerBasic är en underklass till *Tower* och hanterar de specifika

egenskaperna för tornen Allround och Short Range.
- Metod för icke målsökande projektil.

■ *TowerShotgun*
TowerShotgun är en underklass till *Tower* och hanterar de specifika egenskaperna för tornet Shotgun.
- Metod för sju icke målsökande projektiler.

■ *TowerSniper*
TowerSniper är en underklass till *Tower* och hanterar de specifika egenskaperna för tornet Sniper.
- Metod för målsökande projektil.

❖ *Obstacle*
Obstacle är en underklass till *HealthEntity* och hanterar alla egenskaper och beteende för alla hinder.
➢ Metod för att skada fiender.
➢ Metod för att ta bort hinder som dött.



UML-Diagram 4, illustration av ärvning från HealthEntity.

6.3. Användning av fritt material

Vi har använt klasser som ingår i Java 8 och inte några externa bibliotek. Utöver det har den mesta av koden för att skapa en topplista i spelet hämtats från Oscar Sandbergs projekt i Tetris.

6.4. Användning av objektorientering

Här beskriver vi hur javas objektorientering använts för att skapa spelet.

Objekt/klasser

Objekt och klasser är en central del i vår kod för primärt fiender, torn, hinder och projektiler som alla är objekt i spelet. Olika objekt har olika egenskaper och beteenden och då används olika klasser för att samla ihop liknande objekt och separera olika objekt.

Alla våra objekt tillhör klassen *Entity* och delar metoderna *getX*, *getY*, *getImg*, *drawImg*, *hasCollision* samt *tick*. Metoderna ger ett objekts position, objektets bild, om objektet kolliderar med andra objekt samt *tick* som kallar på objektens beteenden. Objekten delas sedan upp i två kategorier, de som har HP tillhör klassen *HealthEntity* och de som inte har HP. Det är endast projektiler i spelet som inte har HP och de tillhör en abstrakt klass *Projectiles* som är en subklass till *Entity*. Samtliga projektiler delar metoden *damage* som sänker

fiendernas HP och metoden *hasCollision* för att detektera en kollision. Projektiler behöver en egen metod för *hasCollision* då den försvinner direkt vid kollision till skillnad från torn och fiender som inte ska försvinna när de kolliderar med varandra. *Projectiles* har även en metod *move* som ger projektilerna en målsökande rörelse. Varje projektil delas upp i ytterligare två klasser *ProjectileFollowing* och *ProjectileBasic*. *ProjectileFollowing* ärver samtliga metoder från *Projectile*. Vi har också projektiler som inte är vara målsökande och de tillhör klassen *ProjectileBasic*. I *ProjectileBasic* finns en override av *Projectile*'s metod *move* för att projektilen inte ska vara målsökande.

För våra andra objekt torn, fiender och hinder har vi använt samma system och kan ses i UML-diagramet för *HealthEntity* i avsnitt 6.2.

Om vi inte hade använt objektorientering hade det behövts en klass för varje projektil med olika beteende eller egenskaper och att samtliga metoder och fält finns i varje klass. Klassen *ProjectileFollowing* hade behövt samtliga metoder i *Entity* och *Projectile* och det hade blivit duplicerad kod i projektet.

De olika lösningarna påverkar inte den som spelar spelet, men det får stor påverkan på strukturen och all kod. Genom att använda objektorientering kan vi enklare lägga till fler projektiler med andra egenskaper. Vi behöver endast skapa en ny klass som ärver samtliga metoder och fält från *Projectile* och göra en override av den metoden vi vill ändra eller lägga till en ny metod för det vi lägga till för att ändra beteendet. Nackdelen är att koden blir utspridd och det kan vara svårare att förstå vad varje subklass egentligen gör.

Typhierarkier och Subtypspolymorfism

Vi använder typhierarkier till vår fördel när vi ska hantera entiteter. Vi har med den abstrakta klassen *Entity* garanterat vissa egenskaper och kan således lägga till alla olika entiteter i en och samma lista, exempel på användning:

I klassen *Game* och metoden *tick* itererar vi över listan med entiteter och kallar på deras "tick" funktion som ser till att de utför den tänkta funktionaliteten. Ett problem vi stötte på med denna implementation var när objekt skulle tas bort eller läggas till, det leder till ett concurrent modification exception. Detta löstes genom att skapa temporära listor (*tempAdd, tempRemove)* som används för att uppdatera *entities*-listan utanför iterationens körningstid.

I klassen *TowerDefenceComponent* och metoden *paintComponent* används även listan *enteties* för att rita objekten på skärmen då *Entity* garanterar att alla som ärver av den har nödvändiga metoder och egenskaper för att kunna ritas ut.

Att lösa detta utan objektorientering skulle t.ex. innebära att vi var tvungna att för varje objekt kalla på just dennes *tick* eller *drawImg* funktion. Detta hade lett till extra kod i både *TowerDefenceComponent* och *tick* men även i de enskilda subklasserna. Man vinner även en klar struktur och hierarki mellan de olika klasserna som gör att man enklare kan få en övergripande bild hur spelet hänger ihop. Eftersom vår nuvarande implementation är just objektorienterad är det svårt att säga hur den mäter sig med en t.ex. funktionell approach.

Ärvning med overriding

För att kunna tillhandahålla en effektivitet och flexibilitet i skapandet av nya betéenden för olika enteties används ärvning och overriding av, i första hand *Entity* och *HealthEntiy* men sedan de huvudsakliga klasserna för skapandet av fiender, projektiler, torn och hinder. Alla dessa följer samma struktur, alla subtyper av t.ex. fiender ärver från *Enemy*-klassen där den

mest grundläggande funktionaliteten finns. I fallet för fienden *EnemySeeker* gör den en override på *move*-metoden i *Enemy* för att uppnå ett annat rörelsemönster. Metoden som gör en override ändrar fiendens rörelsemönster till att fiender söker upp torn för att förstöra dem istället för att gå rakt fram mot basen.

För att uppnå ett liknande betéende utan OO skulle man kunna tänka sig ett antal if-satser avgör vilken typ av *move*-funktion som ska användas. Fördelen med detta är att man kanske skulle kunna få lite mer variation på betéendet hos en fiende med färre klasser, i nuläget måste en helt ny klass skapas för att ändra en detalj i betéendet men samtidigt är det stor risk att koden blir grötig och svår att tolka när den är fylld med if-satser.

Inkapsling

Vi använder olika typer av inkapsling i olika delar av koden. Till exempel i klassen *Game* är det viktigt att bara vissa klasser kan komma åt *money*-fältet, som är private, genom getters och setters. Därmed skulle vi kunna göra felkontroller innan värdet sätts. Utöver det är det viktigt med inkapsling då många fält har samma namn men får inte blandas ihop.

Om inte inkapsling hade varit möjlig hade alla variabler och metoder varit tillgänglig för samtliga klasser. Vi har då behövt göra kontroller och begränsningar i många klasser för att begränsa möjligheten att manipulera data.

Genom att använda inkapsling vinner vi mycket funktionalitet gällande att enkelt göra fält och metoder synliga från vissa men inte andra klasser eller helt begränsa synligheten till nuvarande klass. Med inkapsling krävs det mindre kod samtidigt som det är enklare att få en överblick överblick över vad som är synligt för andra klasser. Vid mindre projekt kan det vara enklare att inte använda inkapsling för att slippa använda sig av t.ex getters och setters.

6.5. Motiverade designbeslut med alternativ

Vi har tagit flera designbeslut under projektets gång som diskuteras och motiveras nedanför.

Designbeslut 1: Abstrakta klasser

Vi strävade efter att åstadkomma ett enkelt sätt att öka antalet torn, fiender och hinder. En del av objekten ska dela vissa egenskaper och beteenden med varandra, men också ha möjligheten att skilja sig från varandra på dessa punkter. Vi valde att använda oss av abstrakta klasser istället för gränssnitt som vi först hade tänkt, för att underklasserna kan utökas och beteenden och egenskaper kan ändra eller läggas till.

T.ex har den abstrakta klassen *Enemy* en metod *move* som styr en fiendes rörelse. Underklasserna *EnemyBasic* och *EnemySeeker* utökar *Enemy*. *EnemyBasic* ärver metoden *move* från *Enemy*, medans *EnemySeeker* dominerar “override” metoden *move* för att objekt som tillhör den klassen ska få ett eget rörelsemönster.

Ett alternativ är att använda gränssnitt istället för abstrakta klasser. Det var också den ursprungliga idén när vi skrev våra milstolpar. Vi valde abstrakta klasser för att det ger oss möjlighet att ärva samtliga metoder och fält från abstrakta klassen men också dominera “override” metoder från den abstrakta klassen för att ändra beteenden och egenskaper. Designbeslutet tycker vi är rätt även om det hade fungerat med gränssnitt.

Designbeslut 2: Slopning av Enum-klasser

När vi började skriva kod upptäckte vi att användningen av Enum-klasser inte behövdes. Enum-klasser för det som senare blev entiteter verkade överflödigt för det spelet vi skulle skapa, nämligen hantering av objektens egenskaper och utseende. Funktionaliteten vi var ute efter fick vi från de abstrakta klasserna *Tower, Enemy, Projectile,* och *Obstacle* samt våra olika *Maker*-klasser för dessa. Vi hade kunnat använda Enum-klasser men nu kom vi undan med ett par färre klasser samt en mer modulär lösning som gjorde att utvecklingen av spelet gick snabbare.

Designbeslut 3: Expandera *tick* till *Entity*

Från början var det tänkt att klassen *Game* skulle ta hand om allt som behövde göras under ett tick och vi körde på den implementationen ett tag men det gjorde alla våra klasser väldigt sammanbundna och vi saknade flexibilitet och självständighet i fiender och torn. Därför valde vi att utöver *tick* i game ge *Entity* en *tick*-funktion som skulle ta hand om det just den entiteten behövde göra. I slutändan ledde detta till att *Game* kunde bli centrerad på spelmekanik och *Entity* blev självständig efter skapandet.

Designbeslut 4: Uppdelandet av grafik, spelmekanik och entiteter.

Från projektet i Tetris lärde vi oss att det är viktigt att dela upp programmet i rimliga delar. Därför tog vi beslutet att dela upp projektet i grafik, spelmekanik och entiteter och försöka hålla dem separerade för att skapa en tydlig bild av vad de olika klasserna är till för. Istället för denna approach hade man kunnat tänka sig att sudda ut gränserna mellan grafik, spelmekanik och entiteter en aning för att tillåta en mer dynamisk användning av delarna.

Till exempel hade klassen *Game* kunnat stå för skapandet av rutan som spelet befinner sig i istället för klassen *Display.* Men i slutändan valde vi att göra som vi gjorde för att upprätthålla den tydlighet och läsbarhet vi var ute efter.

Designbeslut 5: Spelplan och bakgrund

Till en början var det tänkt att spelplanen skulle byggas upp med hjälp av kvadrater med olika texturer. Vi valde bort det alternativet då vi ännu inte implementerat en map editor och då bedömde vi att det tappade sitt syfte.

Vi ville först och främst ha ett spelbart spel och valde då att använda oss av en bakgrundsbild som täcker hela spelplanen och ritas ut varje tick av *TowerDefenceComponent*. Eftersom att bakgrunden endast är ett objekt blir det mer effektivt än att rita ut många små, men det hade öppnat upp för mera möjligheter när det kommer till terräng och hur dynamiskt spelet skulle kunna bli.

Båda dessa lösningar är rimliga men passar olika bra i olika applikationer och den vi valde är passande för vårt spel då kvadrater skulle bli mer komplicerat.

Designbeslut 6: Levels

För att spelet skulle bli svårare och svårare att spela ville vi att spelet skulle ha levels där spelet blev svårare och svårare för varje level.

Vi valde att implementera en multiplikator som ökar desto längre man spelat (var 2000e tick i spelet). Varje fiendes HP multipliceras med multiplikatorn, vilket gör att det blir svårare och svårare att döda fienderna.

En alternativ lösning är att man har omgångar där det är fördefinierat vilka fiender som ska

ingå i varje omgång. För varje omgång blir det fler och fler fiender och svårare och svårare. Det hade kunnat implementeras genom att varje omgång består av en lista som beskriver antalet fiender och vilken typ av fiende i varje omgång.

I början var vår tanke att implementera omgångar och inte multiplikatorn som vi tillslut valde. Men vi valde multiplikatorsystemet för det är skalbart och vi behöver inte hårdkoda varenda omgång. Att hårdkoda varenda omgång hade inneburit att det fanns ett slut på spelet som begränsas av hur många omgångar vi hade hårdkodat. Nu bestäms slutet istället av hur länge hur spelaren överlever.

Designbeslut 7: Alternativ till tick

Vi behövde något som driver fram spelet dvs att fienderna rör sig framåt, attackerar torn osv. Vi valde därför att implementera en timermetod *GameClock* som tickar ca 60 gånger per sekund för att driva framåt spelet. I varje tick kallas huvudklassen *Game's* egen tick funktion som hanterar nödvändig funktionalitet. Alla entiteter har också en tick-funktion, detta innebär att de är relativt självständiga efter de har skapats. I slutet av varje tick i huvudklassen *Game* uppdateras även grafiken som ritas ut på skärmen.

Ett alternativ hade varit att spelet drivs framåt av en while-loop som slutar när basen i spelet dödats. För att få en rimlig fart i spelet kan man lägga till *Thread.sleep(1)*.

```
public void gameLoop()
{
	while(baseHealth >= 0) {
		updateGame();
		repaint();
		Thread.sleep(1);
	}
}
```

Att göra som kodexemplet ovanför hade fungerat okej, men det kan uppstå problem om processorn arbetar med andra trådar samtidigt. Då är det inte säkert att spelets tråd startar igen när den ska. Vår lösning är mycket mer stabil och säker då den inte riskerar samma problem som om man pausar en tråd.

Designbeslut 8: Fienders rörelse mot basen

Fienderna behöver röra sig från spelplanens vänstra sida mot basen för att försöka döda basen och vinna över spelaren.

Vi valde att ha en öppen spelplan där fienderna kan röra sig fritt över spelplanen. Fienderna dyker upp på spelplanens vänstra sida men var i höjdled är slumpat. Sedan kan fienderna röra sig enligt olika rörelsemönster mot basen. Om en fiende kolliderar med ett torn eller hinder stannar fienden och attackerar tornet eller hindret.

Ett alternativ är att alla fiender dyker upp från en position och sedan följer en väg mot basen. Det alternativet är vanligt för denna genre av spel t.ex Bloons Tower Defence-spelen är uppbyggt på det sättet.

Anledningen till att vi valde att ha en öppen spelplan var för att kunna göra spelet mer flexibelt. Med en öppen spelplan kan vi göra att olika fiender rör sig på olika sätt, vilket gör att spelet inte ses som lika enformigt. Vi har en fiende som endast kan röra sig i x-led, medans en annan som även kan röra sig i y-led och letar upp närmaste torn framför sig som den attackerar. Det hade inte varit möjligt om vi valt den alternativa lösningen. Den alternativa lösningen kan vara smidig då man inte behöver hantera vad som händer med kollisioner och oftast i de spelen kan inte fienderna attackera tornen. Det hade förenklat en del kod i vårt spel, men rörelsemönstret att följa en väg är svårare än med alternativet vi valde.

7. Användarmanual

Skärmbild 1, hur spelet ser ut.

Spelet är ett 2D strategispel (se Skärmbild 1) inom kategorin Tower Defence. Det går ut på att man ska samla ihop poäng, desto mer poäng desto bättre. Det gör man genom att överleva länge och döda fiender. Fienderna rör sig från vänstra sidan av spelplanen till den högra sidan mot en bas. Fienderna kan inte gå igenom torn eller hinder utan stannar och attackerar tornet eller hindret. Om fienderna förstör basen är spelet slut. För att överleva längre kan man köpa torn och hinder som skadar eller hindrar fiender från att överleva och döda basen. Basen kan inte skada fiender.

Hur startar man spelet?

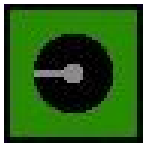
Spelet startar man genom att öppna projektet i en IDE och sedan köra klassen *Launcher*.

Avsluta spelet

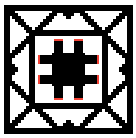
Man avslutar spelet antingen genom att klicka på röda krysset längst upp i hörnet eller när man förlorat i spelet och då välja att inte starta ett nytt spel.

Torn

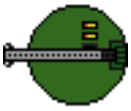
I spelet finns det fyra olika typer av torn, Allround, Short Range, Sniper och Shotgun. De olika typerna av torn har olika egenskaper och beteenden. Egenskaper för alla torn finns i tabell 1.



Allround, är det billigaste tornet i spelet och är medelbra i förhållande till de andra tornen. Den skadar minst av alla torn, men har kort omladdningstid. Tornet har även näst längst sikte, har mest HP och är billigt. Det gör tornet att tornet fungerar till det mesta.



Short Range, är unikt med sitt korta sikte stora skada. Omladdningstiden är lika kort som för tornet Allround, men Short Range har 50 % mindre HP. Short Range är det näst billigaste tornet i spelet.



Sniper, har ett extremt långt sikte som kan täcka nästan hela spelplanen om den placeras bra. Varje skott ger mycket skada och är även målsökande för att ge högre träffsäkerhet, men tornet lång omladdningstid och lägst HP av alla torn. Sniper är det dyraste tornet i spelet.

Shotgun, är det näst dyraste tornet i spelet, men har näst mest HP. Sju skott skjuts när Shotgun skjuter, varje skott ger inte mycket skada och omladdningstiden är längst av alla torn. Tornet kan sikta nästan lika långt som Allround.

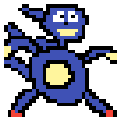
Tabell 1, tornens egenskapersvärden.

Egenskaper	Allround	Short Range	Sniper	Shotgun
HP	100	50	25	75
Skada	1	10	6	2
Längd på siktet	200	50	700	175
Omladdningstid	20	20	100	250
Pris	50	150	300	250

Fiender

I spelet finns två olika fiender med olika egenskaper och beteenden. Samtliga fienders HP och ersättning ökar desto längre tid man spelar, varje 2000:de tick i spelet ökas multiplikatorn med ett. Fienderna kan inte röra sig igenom torn eller hinder utan stannar om en kollision uppstår och attackerar tornet eller hindret. Fiendernas egenskaper finns i Tabell 2.

Enemy Basic, är den ven vanligaste fienden och rör sig från spelplanens vänstra sida till den högra sidan och basen. Fienden kan endast röra sig i x-led, men dyker upp på olika y-position för varje ny fienden som dyker upp på skärmen.



Enemy Seeker, dyker upp var 15:e fiende. Den dyker upp en sumpad y-position varje gång och Enemy Seeker rör sig mot närmaste torn istället för mot basen men aldrig bakåt. Om inga torn är placerade rör den sig endast i x-led mot basen och rörelsen

påverkas aldrig av om hinder är utplacerade.

Tabell 2, fiendernas egenskapsvärden.

Egenskaper	Enemy Basic	Enemy Seeker
HP	7 x multiplikator	3 x multiplikator
Skada	10	5
Attackhastighet	50	15
Ersättning	10 x multiplikator	20 x multiplikator
Hastighet	2	4

Hinder

Det finns två hinder i spelet, Nails och Blocker. Egenskaperna finns i tabell 3.

Nails, skadar fiender när de kolliderar med spikarna och fiendens hastighet sänks till 0 tills spikarna dödats.

Blocker, skadar inte fiender något utan blockerar endast deras väg och sänker fiendens hastighet till 0 till blockern dödats.

Tabell 3, hindrens egenskapsvärden.

Egenskaper	Nails	Blocker
HP	15	20
Skada	1	-
Attackhastighet	10	-
Pris	30	20

Pengar

Spelaren får pengar när en fiende dödas, hur mycket beror på vilken typ av fienden som dödats. Poängen beror även på hur länge spelaren överlevt. Desto längre spelaren överlever desto mer poäng och ersättningen för varje dödad fiende ökar också. Spelarens pengar ökar även med ett var 40de tick i spelet. Med pengarna kan man köpa torn eller hinder för att överleva längre.

Poäng

I spelet finns ett poängsystem, spelaren får en poäng var 10de tick i spelet. Desto längre tid man överlever desto mer poäng och därmed större chans att komma in på highscorelistan.

Köpa torn eller hinder

Om man har tillräckligt med pengar kan man köpa torn eller hinder för att överleva längre. Det görs genom att välja "Towers" eller "Obstacles" i menyraden längst upp till höger och därefter klicka på det tornet eller hindret man vill köpa. Därefter placerar man ut tornet eller hindret genom att klicka där på spelplanen man vill placera det. Det går inte att placera torn eller hinder på varandra.

8. Slutgiltiga betygsambitioner

Våra betygsambitioner är en 5:a och vi anser att vi uppfyller samtliga krav för att få det betyget.

9. Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
    - Övergripande har det mesta gått bra! Det tog lite tid att komma igång med själva programmerandet då vi inte riktigt var säkra på hur vi faktiskt skulle göra i detalj. Vi hade i projektbeskrivningen tänkt oss att använda Enum och gränssnitt vilket vi gjorde i början men övergick senare till abstrakta klasser, något vi skulle gjort från början.
- *Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken? Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har "hjälpt" bäst? Vi vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!*
    - Material: Vi har inte använt mycket material förutom att vi har tagit en viss inspiration från tetrisprojektet.
    - Föreläsningarna var tyvärr för komplicerade/abstrakta och tekniska för att vi skulle kunna dra nytta av dem.
    - Labbarna har varit bra att gå på.
    - Oracles egna dokumentation och tutorials har varit till stor hjälp.
- *Har ni lagt ned för mycket/lite tid?*
    - Vi lade för lite tid i början på att strukturera koden och hur allt skulle hänga ihop. Det tog lång tid innan vi hade en struktur som vi var bekväma med.

    - Vi känner inte att vi har lagt för mycket tid på något. Vi hade en feature freeze i mitten av projektet vilket gav oss tid att strukturera och förbättra vårt arbetssätt och vad vi skulle fokusera på.
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
    - Det som är viktigt för ett bra samarbete och inte bara arbetsfördelning är kommunikation. Det har fungerat bra, även om vi i olika perioder jobbat olika mycket.
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
    - Grunden av projektbeskrivningen var bra även om en del detaljer som vi ändrade. Projektbeskrivningen gav oss något att luta oss mot för när koden skrevs
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
    - Vi har följt arbetsmetodiken vi utformade i projektbeskrivningen. Vi har jobbat på schemalagda pass, men mycket utanför schemalagd tid och på kvällar. Helger har vi endast jobbat nu när slutinläningen ska in.
- *Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.*
    - Rapportskrivandet var det mest problematiska pågrund av lite erfarenhet av det och svårt att veta förväntansnivån.
    - Det har inte varit svårt att hitta plats att vara på.
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
    - Underskatta inte tiden det tar att tänka ut allt, programmera, felsöka osv. Det underlättar väldigt mycket om man börjar tidigt, det kommer ge en mer tid att polera koden och hantera buggar som dyker upp.
- *Har ni saknat något i kursen som hade underlättat projektet?*
    - Kraven på koden för att få vissa betyg var utspritt på olika ställen, samt att det var svårt att tolka och förstå en del av kraven. Mycket information var utspritt på olika ställen, vilket gjorde det svårt när information gällande kod och projektbeskrivning skulle matchas med kraven.
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
    - Föreläsningarna var tidigt under kursen och kan vara svårt att vid projektet komma ihåg vad som sades under föreläsningar. Om föreläsningarna varit utspridda under längre tid eller att det funnits fler labbassistenter som kunde hjälpa till då de ofta under sista projektet inte hann med att hjälpa alla.