

Projektbeskrivning

<Geometric Tower Defense>

2018-05-06

Projektmedlemmar:

Edwin Vikström Johansson <edwjo109@student.liu.se>

Namn Efternamn <marno874@student.liu.se>

Handledare:

Handledare <mariusz.wzorek@liu.se>

Table of Contents

1. Introduktion till projektet.....	2
2. Ytterligare bakgrundsinformation.....	2
3. Milstolpar.....	2
4. Övriga implementationsförberedelser.....	4
5. Utveckling och samarbete.....	4
6. Implementationsbeskrivning.....	6
6.1. Milstolpar.....	6
6.2. Dokumentation för programkod, inklusive UML-diagram.....	8
6.3. Användning av fritt material.....	10
6.4. Användning av objektorientering.....	10
6.5. Motiverade designbeslut med alternativ.....	12
7. Användarmanual.....	14
8. Slutgiltiga betygsambitioner.....	16
9. Utvärdering och erfarenheter.....	16

Planering

1. Introduktion till projektet

Vi ska göra ett Tower Defense-spel som är inspirerad av "Bloons Tower Defense". Tower Defense är en kategori av spel med rötter till 1980-talet som går ut på att försvara en punkt (ofta symboliserat som en bas eller by) från fiender som går genom en bana. Detta görs genom att bygga olika sorters torn som ska försvara din bas genom att eliminera fienden. Fienderna går längst en förbestämd väg, där det inte går att placera ut torn. Spelaren förlorar om ett visst antal fiender når banans slutpunkt.

Vi använder inspirationsprojektet "TDDD78: Tower Defense".

2. Ytterligare bakgrundsinformation

Se Wikipedias artikel https://sv.wikipedia.org/wiki/Tower_defense.

3. Milstolpar

#	Beskrivning
1	Skapa spelplanen som representeras av ett rutnät där varje cell aningen är fiendernas väg, ett torn eller byggbar mark. Det ska finnas en grafisk komponent för att visa spelplanen. Man ska kunna stänga spelets fönster med en knapp.
2	Skapa en fiende och funktionalitet så att fienden kan visas på spelplanen och röra sig i en riktning.
3	Skapa checkpoints för fienders rörelse och funktionalitet för fiender så de kan röra sig längst en bana.
4	Skapa ett torn som kan placeras ut i en ledig cell genom musklick.
5	Funktionalitet så att torn kan skjuta på en fiende med jämna mellanrum om fienden befinner sig inom räckhåll. Om en fiende blir träffad så tas den bort från spelplanen. Skottet visas som ett sträck mellan tornet och fienden. Sträcket försvinner nästa frame.
6	Fixa så att fiender skapas med jämna mellanrum vid startpunkten.
7	Lägga till hp till fienden så att det krävs minst ett skott för att döda dem. Tornen gör för tillfället 1 hp skada per skott.
8	Skapa ett resurssystem så att man får resurser när ett torn skjuter en fiende (x skada ger x resurser). Torn ska kosta resurser för att kunna byggas. Resurser visas

	som text någonstans på skärmen.
9	Skapa funktionalitet så att man förlorar då tillräckligt många fiender når banans slutpunkt. Då ska en ruta komma upp som frågar om man vill starta om eller avsluta.
10	Lägga till två torn. Det ena skadar alla fiender som är runt om den och det andra skjuter en missil som gör "splash damage" (skada på flera fiender samtidigt).
11	Lägga till grafik för att visa byggbara torn i en meny. Man ska kunna klicka på tornens symboler så de är valda och då blir placerbara (om man har resurserna).
12	Fienderna ändrar färg och fart beroende på hur mycket hp de har. Ju mindre hp, desto långsammare blir de.
13	Lägga till så att fiender skapas i ronder som blir fler och fler för varje rond. Lägga till en knapp som startar en ny rond. Rondnummer visas någonstans på skärmen.
14	Lägga till så man kan spara highscore när man förlorat (score = antal ronder man klarat) tillsammans med ett alias. Highscore-listan visas efter man angett ett alias.
15	Lägga till så man kan se ett torns statistik när man klickar på den. Statistiken visas under menyn där man köper torn.
16	Lägga till så man kan uppgradera torn. Uppgradering kostar resurser och förbättrar t.ex. skada och räckvidd. När ett torn uppgraderas så får den ett nytt utseende. Uppgradering sker genom att man trycker på ett torn och sedan klickar på en knapp under tornets statistik.
17	Göra så man kan sälja ett torn genom att trycka på tornet och sedan en knapp vid sidan om uppgradera-knappen. Man får tillbaka hälften av kostnaden för tornets totala kostnad med eventuella uppgraderingar.
18	Lägga till fler fiendetyper. En kan bara ta skada av bomb-tornet (den som gör splash-damage) och är långsam jämfört med den vanliga fienden. En annan kan bara ta skada av torn som uppgraderats minst två gånger och rör sig snabbt. Dessa fiender skapas inte under de första ronderna.
19	Lägga till så att en boss skapas var tionde rond. Under dessa ronder skapas bara bossen. Bossen har väldigt mycket hp och är långsam.
20	Lägga till ljudeffekter, t.ex. när man köper nya torn och när man skjuter.
21	Lägga till så man kan välja att ett torn prioriterar vissa fiender. Man ska kunna välja mellan först, sist, mest hp och minst hp. Man väljer prioritering för ett torn med knappar som finns under uppgradera-knappen.
22	Lägga till fler banor med olika svårighetsgrad. En bana ska ha en väg som splittras i två så fienderna tar båda vägarna. En annan bana ska ha två punkter där fiender

skapas. Man väljer bana innan ett spel startas.

23 Välja svårighetsgrad. Reglerar hur mycket torn kostar och hur ofta fiender skapas.

24 Fixa så att spelet visas relativt fönstrets storlek (t.ex. om man förstörar fönstret så tar spelplanen alltid upp hela fönstret).

4. Övriga implementationsförberedelser

Koden ska delas upp så att:

- En klass håller reda på vad som finns på spelplanen och på vilken position med hjälp av en två-dimensionell array (likt Tetris), exempelvis tom mark, torn eller fiendernas bana. Ska även innehålla en array som håller koll på fiendernas position på banan och funktionalitet för att bedöma om spelet är igång eller om spelaren har förlorat.
- En klass som hanterar fiendeobjektet.
- En fabriksklass som skapar fienden.
- En klass som sköter utritningen av själva spelplanen, d.v.s. allt som ska uppdateras kontinuerligt.
- En klass som sköter utritningen av själva fönstret och håller koll på keyEvents.
- En Singleton-klass som hanterar highscore listan.
- En klass som hanterar startmenyn.
- Ett interface för tornen.
- En abstractklass för de som tornen har gemensamt.
- Klasser för varje enskilt torn.
- Enumklass med fält som beskriver vad som finns kan finnas på varje plats på banan.
- En klass där spelet startas som även ska innehålla spelklockan.
- Ett interface med en metod som ska kallas på när spelplanen förändras som alla klasser som ska uppdateras ska implementera.
- En klass som innehåller matriser med information om hur varje bana ser ut.

Fienderna ska röra sig längst x- och y-axlarna mellan utsatta checkpoints oberoende av rutnät.

5. Utveckling och samarbete

- Vi kommer att arbeta en hel del tillsammans, så vi kan diskutera struktur, upplägg av kod, funktionalitet med mera. Sedan kommer vi utöver det att arbeta en del enskilt men vi kommer då att se till att båda har tydligt separerade arbetsområden och att vi hela tiden uppdaterar varandra med eventuella problem eller liknande.
- Om någon i gruppen har skrivit kod själv kommer vi gå igenom koden och se till att alla gruppmedlemmar förstår exakt vad koden gör och vad den har för syfte.
- När vi arbetar med projektet kommer nog variera ganska mycket beroende på hur det ser ut med andra kurser och vad som sker vid sidan av studierna. Som det ser ut just nu kommer nog en stor del av arbetet ske dels under arbetstid men även på kvällar och helger.

- Vi går in i det här projektet med höga ambitioner och vi är överens inom gruppen att vi ska satsa mot ett så högt betyg som möjligt.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

1. Skapa spelplanen som representeras av ett rutnät där varje cell aningen är fiendernas väg, ett torn eller byggbar mark. Det ska finnas en grafisk komponent för att visa spelplanen. Man ska kunna stänga spelets fönster med en knapp.

- Avklarat.

2. Skapa en fiende och funktionalitet så att fienden kan visas på spelplanen och röra sig i en riktning.

- Avklarat.

3. Skapa checkpoints för fienders rörelse och funktionalitet för fiender så de kan röra sig längst en bana.

- Avklarat.

4. Skapa ett torn som kan placeras ut i en ledig cell genom musklick.

- Delvis implementerat. Torn placeras inte i celler utan de placeras valfritt på banan så länge de inte överlappar fiendernas väg eller andra torn.

5. Funktionalitet så att torn kan skjuta på en fiende med jämna mellanrum om fienden befinner sig inom räckhåll. Om en fiende blir träffad så tas den bort från spelplanen. Skottet visas som ett sträck mellan tornet och fienden. Sträcket försvinner nästa frame.

- Avklarat, dock visas skotten under tre ticks.

6. Fixa så att fiender skapas med jämna mellanrum vid startpunkten.

- Avklarat.

7. Lägga till hp till fienden så att det krävs minst ett skott för att döda dem. Tornen gör för tillfället 1 hp skada per skott.

- Avklarat.

8. Skapa ett resurssystem så att man får resurser när ett torn skjuter en fiende (x skada ger x resurser). Torn ska kosta resurser för att kunna byggas. Resurser visas som text någonstans på skärmen.

- Avklarat.

9. Skapa funktionalitet så att man förlorar då tillräckligt många fiender når banans slutpunkt. Då ska en ruta komma upp som frågar om man vill starta om eller avsluta.

- Avklarat.

10. Lägga till två torn. Det ena skadar alla fiender som är runt om den och det andra skjuter en missil som gör "splash damage" (skada på flera fiender samtidigt).

- Avklarat.

11. Lägga till grafik för att visa byggbara torn i en meny. Man ska kunna klicka på tornens symboler så

de är valda och då blir placerbara (om man har resurserna).

- Avklarat.

12. Fienderna ändrar färg och fart beroende på hur mycket hp de har. Ju mindre hp, desto långsammare blir de.

- Ej implementerat.

13. Lägga till så att fiender skapas i ronder som blir fler och fler för varje rond. Lägga till en knapp som startar en ny rond. Rondnummer visas någonstans på skärmen.

- Avklarat.

14. Lägga till så man kan spara highscore när man förlorat (score = antal ronder man klarat) tillsammans med ett alias. Highscore-listan visas efter man angett ett alias.

- Delvis. Highscore och ett alias sparas när man förlorat, dock visas inte highscore-listan direkt utan man måste själv navigera sig till highscore-menyn i startmenyn. Dessutom beror inte ens poäng bara på hur många ronder man klarat av, utan även hur mycket skada man tilldelat fienderna.

15. Lägga till så man kan se ett torns statistik när man klickar på den. Statistiken visas under menyn där man köper torn.

- Delvis implementerat. Den enda statistiken som visas är tornets nivå.

16. Lägga till så man kan uppgradera torn. Uppgradering kostar resurser och förbättrar t.ex. skada och räckvidd. När ett torn uppgraderas så får den ett nytt utseende. Uppgradering sker genom att man trycker på ett torn och sedan klickar på en knapp under tornets statistik.

- Delvis implementerat. Tornet får inte ett nytt utseende när de uppgraderas.

17. Göra så man kan sälja ett torn genom att trycka på tornet och sedan en knapp vid sidan om uppgradera-knappen. Man får tillbaka hälften av kostnaden för tornets totala kostnad med eventuella uppgraderingar.

- Avklarat.

18. Lägga till fler fiendetyper. En kan bara ta skada av bomb-tornet (den som gör splash-damage) och är långsam jämfört med den vanliga fienden. En annan kan bara ta skada av torn som uppgraderats minst två gånger och rör sig snabbt. Dessa fiender skapas inte under de första ronderna.

- Delvis. Den sistnämnda fienden byttes ut mot en fiende som teleporteras mellan vägens olika hörn.

19. Lägga till så att en boss skapas var tionde rond. Under dessa ronder skapas bara bossen. Bossen har väldigt mycket hp och är långsam.

- Ej implementerat.

20. Lägga till ljudeffekter, t.ex. när man köper nya torn och när man skjuter.

- Ej implementerat.

21. Lägga till så man kan välja att ett torn prioriterar vissa fiender. Man ska kunna välja mellan först, sist, mest hp och minst hp. Man väljer prioritering för ett torn med knappar som finns under uppgradera-knappen.

- Ej implementerat.

22. Lägga till fler banor med olika svårighetsgrad. En bana ska ha en väg som splittras

i två så fienderna tar båda vägarna. En annan bana ska ha två punkter där fiender skapas. Man väljer bana innan ett spel startas.

- Delvis implementerat. Det finns olika banor men de beter sig på samma sätt.

23. Välja svårighetsgrad. Reglerar hur mycket torn kostar och hur ofta fiender skapas.

- Ej implementerat.

24. Fixa så att spelet visas relativt fönstrets storlek (t.ex. om man förstorar fönstret så tar spelplanen alltid upp hela fönstret).

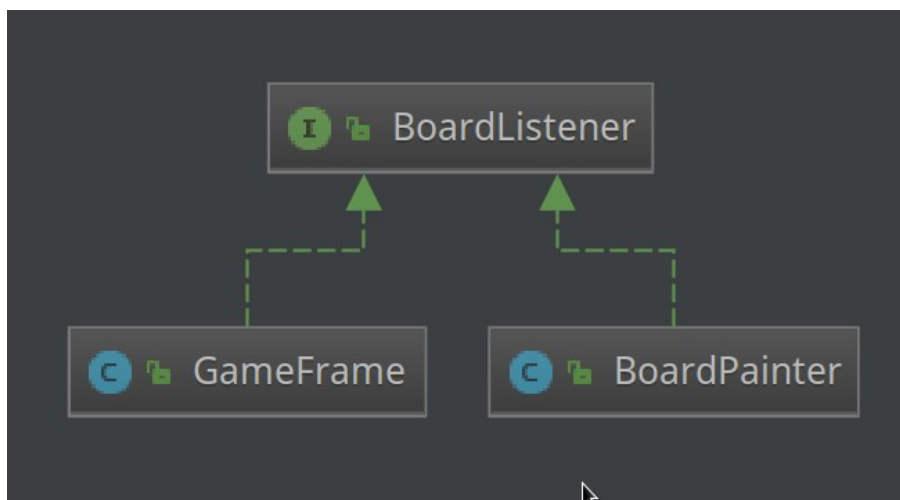
- Delvis implementerat. Fönstrets storlek går inte att ändra.

6.2. Dokumentation för programkod, inklusive UML-diagram

Programmet startas upp i klassen MenuFrame. Den innehåller funktionalitet för att navigera till programmets resterande fönster. Istället för att använda JButtons för att byta fönster kollar programmet istället efter knapptryckningar, och om koordinaterna för en knapptryckning är inom ett förbestämt intervall så tas det gamla fönstret bort och ett nytt fönster öppnas. Dessa intervall ritas upp med hjälp av klassen MenuPainter, som ritat svartvita boxar i fönstret. Klasserna SettingsFrame, HelpFrame och HighscoreFrame använder samma logik när det kommer till knapptryckningar, och de ritas upp med klasserna SettingsPainter, HelpPainter och HighscorePainter.

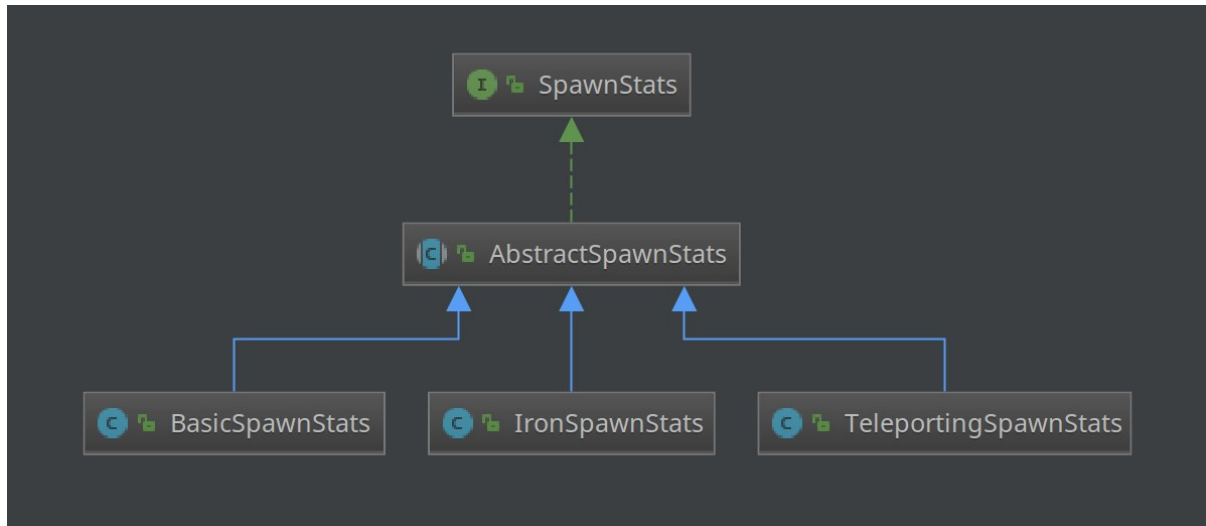
När användaren är i SettingsFrame och klickar på en av knapparna (som väljer bana) så skapas ett Board-objekt med parametrar om den valda banan och en klocka startas med startClock(). Denna klocka kommer kalla på funktionen tick() i Board var 20:e millisekund.

Funktionen tick() styr spelet framåt. Först skapas fiender (Enemy) enligt SpawnStats (diskuteras senare i denna del av rapporten), sedan attackerar torn (Tower) fiender och till sist så rör sig fienderna.

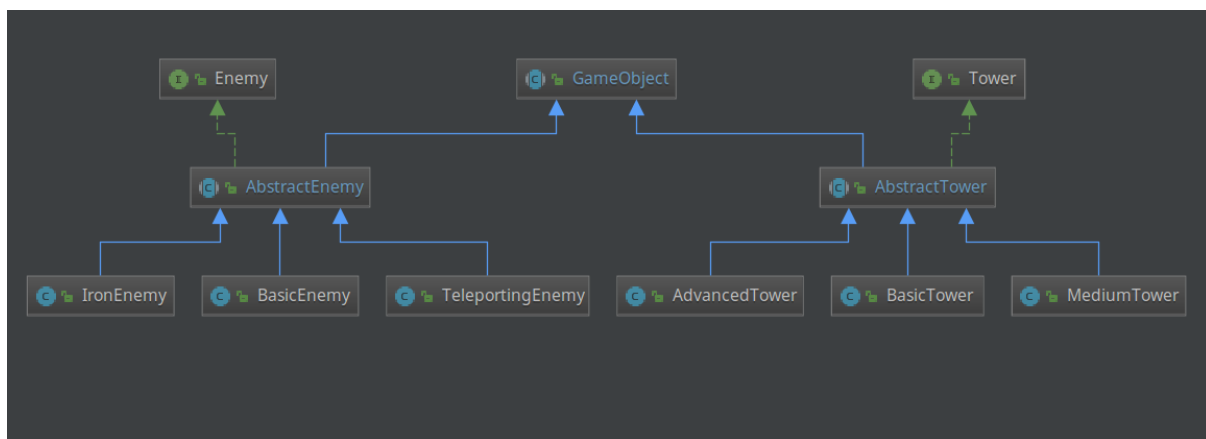


Klasserna GameFrame och BoardPainter implementerar gränssnittet BoardListener för att kunna visa spelets information i realtid. När någonting sker med spelplanen, t.ex. att ett torn

skapas eller fiender rör sig, kommer Board kalla på funktionen `notifyListeners()` för att notifiera de klasser som implementerar `BoardListener` om att spelplanen uppdaterats. `NotifyListeners()` kallar alla `BoardListeners` funktion `boardChanged()`. `GameFrame` kommer genom `boardChanged()` uppdatera etiketterna om liv, resurser, nuvarande runda o.s.v. Dessutom kommer den visa en inmatningsruta som frågar om namn och sparar poäng om användaren förlorat. `BoardPainter` kommer genom `boardChanged()` att rita upp hela spelplanen igen, dvs brädet, alla fiender, alla torn och tornens attacker.



Klasser som implementerar `SpawnStats` är klasser som beskriver för Board hur spelets fiender ska skapas, dvs hur många, hur ofta och hur mycket liv de ska ha för varje runda. Klasserna `BasicSpawnStats`, `IronSpawnStats` och `TeleportingSpawnStats` utvidgar den abstrakta klassen `AbstractSpawnStats`, som i sin tur implementerar gränssnittet `SpawnStats`. `AbstractSpawnStats` innehåller gemensamma fält och metoder för dess subclasser. Eftersom alla tre subclasser implementerar `SpawnStats` så kan man lägga de i en gemensam lista (`List<SpawnStats> spawnStats`). Då kan man t.ex. kalla på alla `SpawnStats` `newRound()`-funktion i listan med en for-loop istället för att explicit behöva skriva `BasicSpawnStats.newRound()`, `IronSpawnStats.newRound()`...



Gränssnittet `Enemy` implementeras av `AbstractEnemy`, som har subclasserna `IronEnemy`, `BasicEnemy` och `TeleportingEnemy`. Gränssnittet `Tower` implementeras av `AbstractTower`,

som har subklasserna BasicTower, MediumTower och AdvancedTower. Klasser av typ Enemy och klasser av typ Tower är alla subklasser till GameObject.

GameObject innehåller information som både Enemy:s och Tower:s använder. Eftersom både Tower:s och Enemy:s finns på spelplanen så innehåller GameObject fält för x- och y-led, samt getters för dessa fält.

När spelaren har förlorat skapas ett Highscore objekt med spelarens namn och poäng. HighscoreObjektet skickas sedan in som parameter till funktionen addScore() i klassen HighscoreList. Efter att ett HighscoreObjekt har lagts till i listan sorteras listan med hjälp av klassen ScoreComperator med avseende på poäng. HighscoreList objektet sparas sedan till en fil med hjälp av metoden saveHighscores() i klassen FileManager. HighscoreList objektet kan sedan enkelt kommas åt med funktionen readHighscores från samma klass. Varje gång det skapas en ny spelplan tilldelas fältet highscores i board HighscoreList objektet som finns sparad sedan tidigare. Skulle det inte finnas något objekt sedan tidigare eller sker något fel vid inläsning skapas ett nytt objekt.

6.3. Användning av fritt material

Vi har bara använt java 1.8 och MigLayout. Inspiration till funktionen addComponent i klassen SettingsFrame är tagen från videon https://www.youtube.com/watch?v=ya_kEknUNIE&t=8s av Derek Banas.

6.4. Användning av objektorientering

Följande objektorienterade egenskaper har använts i vårt projekt.

- Objekt och klasser har använts för att samla information och funktionalitet. Till exempel har vi klassen GameObjects som är en abstrakt klass där all funktionalitet som är gemensam för torn och fiender är samlad. Under GameObjects finns i sin tur de abstrakta klasserna abstractTower och abstractEnemy där funktionalitet som är gemensam för torn respektive fiender är samlad. Längst ner i hierarkin finns separata klasser för alla torn och fiender där de egenskaper som är unika för just den typen av objekt finns specificerad.
Ett annat exempel är att det för varje fönster i spelet finns en fönsterklass som är ansvarig för uppritningen av själva fönstret, dess komponenter samt mus- och knapptryck. När ett skifte sker från ett fönster till ett annat är det enkelt att förstöra det första objektet och sedan skapa ett nytt av önskad typ.

I språk som inte är objektorienterade på samma sätt hade det varit svårare att samla gemensamma egenskaper, funktionalitet och fält i en klass, vilket då leder till att utvecklaren är tvungen att skriva samma sak flera gånger i koden. Det skulle inte heller gått att skifta mellan fönster på samma sätt utan skulle troligen behövts lösas med någon form av enum klass samt olika villkorssatser.

- Konstruktörer har använts för att enkelt initialisera fält. Det gör det enkelt att dela information mellan olika klasser genom att helt enkelt ha en konstruktor som tar in de parametrar som behövs och sedan tilldelar informationen till klassens privata fält. En klass kan sedan ha flera konstruktörer som gör olika saker beroende på de parametrar som anges när objektet skapas.

Som exempel har konstruktören använts till exempel i klassen `board` för att initialisera en mängd olika privata fält, vilket inte hade varit möjligt i ett språk utan objektorientering. Där hade varje klass behövt vara mer oberoende eller haft högre synlighet på fält, vilket minskar säkerheten.

- Typhierarkier har använts en hel del. Till exempel finns gränssnitt som grupperar objekt med liknande egenskaper tillsammans. Det finns bland annat listor med torn och fiender som enkelt kan loopas igenom när spelplanen ritas ut. Trots att alla objekt inte är av samma typ kan en metod anropas för alla objekt i den listan eftersom de alla har implementerat ett gränssnitt som försäkrar att alla objektet innehåller vissa metoder. Det här använts på många ställen till exempel i klassen `BoardPainter` där klassen `paintComponent` bland annat anropar metoderna `getRadius` för alla fiender och `getImage` för alla torn. Även i klassen `Board` finns ett exempel på typhierarkier där en lista med `BoardListeners` itereras över och för varje objekt i listan anropas metoden `Boardchanged`, vilket är möjligt enbart för att det finns ett gränssnitt som alla objekt i listan implementerar.

I språk utan objektorientering hade det här inte varit möjligt. Då hade det enklaste troligen varit att bara ha en klass för fiender och sen låta till exempel `getRadius` metoden i den klassen ta in typen av fiende som parameter och sedan ha olika returvärde beroende på vilken typ av fiende det är.

- Inkapsling har använts bland annat i klasserna `AbstractTower` och `AbstractEnemy` där vi låter de flesta fält vara "protected" för att det ska vara tillgängliga från subklasser. I andra språk utan objektorientering får man antingen låta fälten vara globala eller publika och på så sätt minska säkerheten eller istället föra med fälten som parametrar till enskilda metoder.
- Att låta en abstraktklass innehålla en standard representation av en funktion och sedan skriva över den med `@override` i de subklasser där den är annorlunda har använts bland annat för fiender. Där finns en standard representation av funktionen `Move` men eftersom `IronEnemy` ska röra sig annorlunda skriver den över den funktionen med en egen `move` funktion.
I ett språk som inte är objektorienterat hade det inte gått att göra på samma sätt utan varje klass hade behövt ha en egen representation av funktionen, vilket hade lett

till en hel del upprepad kod.

6.5. Motiverade designbeslut med alternativ

Designbeslut 1: SpawnStats

Det vi ville åstadkomma med SpawnStats var att få ett sätt att skapa fiender för varje runda. Vi gjorde det genom att ha en SpawnStats-klass för varje Enemy-klass. Dessa SpawnStats-klasser har fält för hur många fiender som ska skapas, med vilken periodtid de ska skapas och hur mycket liv de ska ha. SpawnStats klasserna innehåller också fält med flyttal som talar om hur mycket vissa fält ska öka varje gång en ny rond startas. Förändringarna för fälten sker i varje SpawnStats-klass funktion newRound(), som kallas när Board kallar på sin newRound()-funktion. Algoritmen som används i en SpawnStats-klass newRound() är unik för den klassen. Det är i Boards funktion tick() som fienderna skapas, genom att kolla i varje spawnStats om ett heltal currentTime (ökar med ett för varje tick) är lika med periodtiden avrundat uppåt. Då minskar ett heltal currentAmount med ett och en fiende av SpawnTypes typ skapas. Denna process återupprepas tills currentAmount är lika med noll för alla SpawnStats.

En alternativ lösning till SpawnStats vi hade kunnat välja är att hårdkoda all statistik om fienderna i en klass, och sedan ha en funktion newRound(int round) i den klassen, som med en switch-sats av round returnerar en hårdkodad lista av fiender.

Vår lösning är bättre i det avseendet att den är mer objektorienterad, men sämre då rundorna inte blir lika rättvisa som om man skulle hårdkoda dem. Om man hårdkodar kan man testa sig fram och se vad som passar för varje runda, men att skapa en algoritm är svårare, och vi ville inte spendera för mycket tid på det.

Designbeslut 2: Fönster

Det vi ville åstadkomma med fönster var ett sätt att visa spelet och menyer. Vi implementerade detta genom att ha en fönster-klass och en uppritnings-klass per fönster. De klasser vi har är MenuFrame, HelpFrame, ScoreFrame, HighscoreFrame, SettingsFrame och GameFrame. Varje gång man kommer till en ny meny så stängs det nuvarande fönstret ner, och en instans av det nya fönstrets klass skapas. Ett alternativt sätt vi hade kunnat välja hade kunna varit att ha olika uppritnings-klasser men bara en fönster-klass. Man hade då kunnat haft enumvärden för de olika fönsterna och använt olika uppritningsklasser beroende på nuvarande enumvärde.

Vi valde att inte ha det på det sättet eftersom vi tyckte det var svårare att implementera, då vi hade olika muslyssnare för varje fönsterFrame. Det gav oss då mer tid att fokusera på viktigare aspekter av projektet. Dessutom så är den grafiska aspekten inte viktig i jämförelse med den objektorienterade i detta projekt.

Designbeslut 3: Representera torn i spelbrädet

Det vi ville åstadkomma var att få ett sätt att representera spelets torn på i spelbrädet. Vi har klasser för olika typer av torn: BasicTower, MediumTower och AdvancedTower. Dessa klasser

är subklasser till den abstrakta klassen AbstractTower, som i sin tur implementerar gränssnittet Tower. I board finns fältet List<Tower> towers, som innehåller spelbrädets alla torn. Detta är det sättet vi valde att representera torn i spelbrädet.

En alternativ lösning vi tänkte på var att lägga till enum-värden i Tile för att representera tornen. Då skulle varje torn ta upp en hel cell ruta på spelplanen och sparas i Boards fält tiles. Vi valde att inte använda detta sätt eftersom det är mycket lättare att behandla ett torn som en klass än ett enum-värde, då tornen har mycket funktionalitet kopplade till dem. Dessutom så blir vårt projekt mindre objektorienterat om tornen beskrivs som enum-värden.

Designbeslut 4: Poänglistan

Det vi ville åstadkomma med poänglistan var att poängen från varje rond skulle sparas i någon slags fil och sedan gå att se i efterhand. Vi valde att ha klassen HighscoreList som innehåller tre stycken listor, en för varje bana, med Highscoreobjekt. Varje gång någon avslutar ett spel sparas ett nytt objekt i listan i HighscoreList som motsvarar den spelade banan. Listan sorteras sedan och den uppdaterade versionen av HighscoreList läses ned till filen och stryker därmed över den gamla

Alternativa lösningar hade varit att HighscoreList bara innehöll listor med de nio bästa poängen och hade en flagga för nuvarande lägsta poäng på topplistan. Man hade då inte behövt spara ett lika stort objekt och inte heller skriva till filen lika ofta utan bara när någon slog sig in på lista.

Man skulle också kunna tänka sig att det hade varit smart att istället för att ha en HighscoreLista med en tre listor, ha tre klasser där var och en innehöll en list med topp poängen för en bana. Man hade då kunnat lagt gemensamma metoder i en abstrakt klass och på så sätt inte behövt skriva så mycket mer kod. Koden hade då blivit mer lättläst och vi hade inte behövt använda oss av switch-satser för att kontrollera vilken bana som spelas och i vilken lista poängen ska läggas till.

Anledningen till att vi inte valde dessa implementationer var dels att vi inte kände att vi tjänade så mycket på att ändra det vi redan hade, då det fungerade bra och dels att det inte egentligen är något som projektet kräver. Därför höll vi oss till den implementation vi hade gjort och fokuserade mer på objektorienteringen i andra moment.

Designbeslut 5: Utséende

Trots att grafiken verkligen inte ligger i fokus i denna kurs vill vi båda att vårt spel ska se åtminstone relativt snyggt ut, då vi tycker att det ger ett starkare helhetsintryck. Därför har vi i bland annat våra menyer valt att använda en grafisk komponent och rita ut boxar med text, istället för att använda färdiga knappar. Det gör att vi varje gång det inträffar ett musklick är tvungna att kolla musen position på skärmen för att veta vad som ska hända.

Ett annat alternativ är att med hjälp av till exempel MigLayout placera ut färdiga knappar under varandra, det hade gjort att vi behövt skriva väldigt mycket mindre kod och ändå haft samma funktionallitet. Anledningen till att vi inte gjorde detta är som sagt ovan att vi tyckte att det såg väldigt tråkigt ut och därför valde vi att lösa det som vi gjorde.

7. Användarmanual

När spelet startas öppnas en huvudmeny. Där möts spelaren av fyra olika alternativ, starta ett nytt spel, se vilka spelare som har fått högst poäng på varje bana, få hjälp eller avsluta. Spelaren gör enkelt sitt val genom att klicka på önskat alternativ.



Om ett nytt spel påbörjas öppnas en ny meny upp där spelaren får välja bana. I nuläget finns tre olika banor att välja mellan, där det som skiljer dem åt är utséendet. När spelet väl startas visas en stor spelplan på vänstersidan och en lite mindre sidopanel på högersidan. I sidopanelen visas bland annat kvarvarande liv, poäng, pengar och nuvarande rond. Längst ner i panelen finns även knappar för att starta en ny rond, pausa eller avsluta.

När en ny rond startas börjar ballonger röra sig längst en förutbestämd väg. Målet är att förstöra ballongerna innan de har hunnit ta sig till slutet av bana genom att strategiskt placera ut olika sorters torn. Det görs genom att klicka på ett torn i sidopanelen och sedan placera det på önskat plats med hjälp av musen. När tornet placeras ut dras respektive torns kostnad från spelarens pengar. Skulle spelaren ha för lite pengar går tornet inte att placera ut och avmarkeras då enkelt genom att klicka någonstans utanför spelplanen. Efter varje rond och när ett torn skadar eller har sönder en ballong belönas spelaren med pengar och poäng. Pengarna använts sedan till att köpa nya eller uppgradera befintliga torn.



Det finns tre olika sorters torn att välja mellan, alla har olika egenskaper och är bra respektive dåliga på olika saker. Det första tornet, det rosa, är väldigt enkelt. Det skjuter en ballong i taget och är bra då det är billigt och har lång räckvidd. Det kan däremot inte skada vissa typer av ballonger.

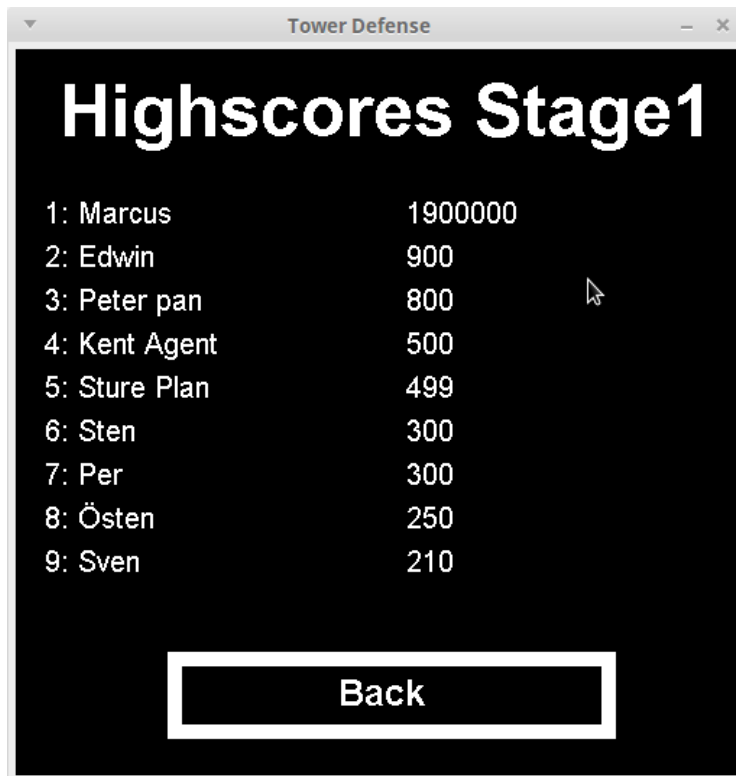
Det andra tornet, det blåa, är lite annorlunda. När det skjuter sker en explosion och den skadar då flera ballonger samtidigt. Det sista tornet, det gula, har liknande egenskaper som det blåa tornet med skillnaden att explosionen inte följer ballongerna, utan hela tiden sker runt tornet. Det är mycket användbart i till exempel hörn, då tornet på så sätt kan skada flera olika grupper av ballonger samtidigt. Både det gula och det blåa tornet kan skada alla olika typer av ballonger.

Utöver att köpa nya torn kan spelaren uppgradera respektive sälja de torn som redan finns på banan. Det görs genom att klicka på ett torn, som då visas upp i sidopanelen, och sedan klicka på önskat alternativ. När ett torn uppgraderas förbättras egenskaper såsom räckvidd, skada per skott och skjuthastighet. Ett torn kan som högst vara i level tio vilket betyder att det maximalt kan uppgraderas nio gånger.

Som nämnts ovan finns det likt torn olika sorters ballonger. Den simplaste och mest förekommande är den röda ballongen. Den rör sig med kontinuerlig fart från start till mål, har relativt lite liv, och kan skadas av alla typer av torn. Från rond tre och framåt förekommer även två andra typer av ballonger. Dessa är gjorda av metall och kan enbart ta skada från det gula och det blå tornet. Den ena av dem uppför sig väldigt likt den första ballongen med skillnaden att den har mer liv och rör sig snabbare. Den andra har dock lite speciella egenskaper, istället för att röra sig kontinuerligt teleporteras den mellan varje hörn på vägen. Det gör att placeringen av tornen blir extra viktig.

Skulle en ballong lyckas ta sig förbi alla torn förlorar spelaren lika många liv som ballongen har kvar. När alla spelarens liv är slut har spelaren förlorat och får skriva in sitt namn i en liten ruta. Namnet och spelarens poäng sparas sedan och har spelaren fått tillräckligt mycket poäng visas statistiken i listan med topp poäng, som man enkelt navigerar till från

startmenyn.



8. Slutgiltiga betygsambitioner

Vi siktar mot att uppnå minst betyget fyra då vi anser att vi har uppfyllt de kraven som finns listade.

9. Utvärdering och erfarenheter

Vad gick bra och vad gick mindre bra?

Det här projektet har varit väldigt lärorikt. Vi kom igång med utvecklingen av spelet rätt sent men när vi väl hade börjat så flöt det på ganska så bra. Speciellt har samarbetet och kommunikationen inom gruppen varit en bidragande faktor. Vi jobbade nästan aldrig tillsammans utan delade hela tiden upp arbetet mellan varandra, vilket gjorde att vi behövde vara väldigt tydliga med vad vad vi skulle och vad vi inte skulle göra.

En sak som inte gick fullt lika bra var planeringen och framför allt att vi inte började arbeta med spelet tidigare. Sista veckorna innan inlämning blev därför extremt intensiva, vilket resulterade i att vi dels inte riktigt hann implementera all funktionalitet som vi hade tänkt och dels att vi stressade igenom en del moment lite för mycket, som vi senare var tvungna att gå tillbaka och justera alternativt göra om.

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 har båda två gått på alla föreläsningarna och läst det mesta av studiematerialet från

kurshemsidan. Utöver det har vi lärt oss det mesta genom att leta på nätet, kolla på videos från youtube eller helt enkelt prövat oss fram för att hitta en lämplig lösning. Vi har inte varit på särskilt många handledda labbar utan vi föredrog att hämta information från andra källor som vi i lugn och ro kunde utvärdera och testa och det tyckte vi båda fungerade väldigt bra.

Har ni lagt ned för mycket/lite tid?

Vi har nog definitivt lagt ner tiden som krävs, åtminstone för att bli godkända med projektet. Vi borde dock ha satt igång lite tidigare så vi hade haft mer tid att lägga på små detaljer och förbättringar.

Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?

Vi fördelade arbetet jämt mellan oss där båda hade tydliga delar de var ansvariga för och på så sätt såg vi till att vi gjorde ungefär lika mycket av arbetet.

Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?

Vi har inte följt projektbeskrivningen till punkt och pricka utan den har hela tiden fått följa med i bakgrunden som en slags grund för hur vi ska strukturera arbetet men aldrig stått i vägen för våra idéer och tankar. Jag tror att det viktigaste med projektbeskrivningen var att den startade en tankeprocess hos oss där vi började fundera över hur utvecklingen av spelet faktiskt skulle gå till innan vi började att skriva kod.

Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?

Arbetet har fungerat bra och som nämnades tidigare så följde vi inte vår arbetsplan till hundra procent utan lämnade rum för idéer som uppkom under tidens gång. Det tror jag helt klart gjorde vårt projekt bättre.

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.

Vi har inte haft så mycket problem vid sidan av projektet. Vi båda har arbetat mestadels på egna datorer och allting har flutit på förvånansvärt bra.

Vilka tips skulle ni vilja ge till studenter i nästa års kurs?

Vi skulle tipsa om att börja i tid och att verkligen lägga ner den tiden som behövs. Dels för att projektet kräver det, dels för att man lär sig väldigt mycket och dels för att det faktiskt är väldigt roligt att ha ett färdigt program som man faktiskt kan känna sig nöjd över.

Har ni saknat något i kursen som hade underlättat projektet?

Vi tycker att upplägget av kursen har varit bra. Möjligen att det hade underlättat med lite mer information kring hur man läser in och sparar objekt till filer och en kort genomgång i hur man till exempel lägger till externa bibliotek i IDEA. Det kanske också hade varit bra om det

fanns lite mer information kring hur man läser in till exempel bilder och ljud och var filerna ska vara sparade i för mapp med mera.

Har ni saknat något i kursen som hade underlättat er egen inläring?

Vi är båda väldigt nöjda med upplägget av kursen och känner att vi har lärt oss väldigt mycket båda två.