Projektbeskrivning

<Geometric Tower Defense>

2018-05-06

Projektmedlemmar:
Edwin Vikström Johansson <edwjo109@student.liu.se>
Marcus Nolkrantz <marno874@student.liu.se>

Handledare:
Handledare <mariusz.wzorek@liu.se>

Table of Contents

Planering

1. Introduktion till projektet

Vi ska göra ett Tower Defense-spel som är inspirerat 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 attackerar spelets fiender. Fienderna går längs 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

Under inledningsfasen av projektet gjorde vi milstolpar för att hålla reda på vad som bör finnas I spelet.

Tabell 1. Visar vad som ska implementeras.

#	**Beskrivning**
1	Skapa spelplanen som representeras av ett rutnät där varje cell antingen 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 så fiender kan röra sig längs en bana.
4	Skapa ett torn som kan placeras ut i en ledig cell genom ett musklick.
5	Implementera funktionalitet så att torn kan skjuta på en fiende med jämna mellanrum om fienden befinner sig inom räckhåll. Om en fiende blir träffad så ska den tas bort från spelplanen. Skottet ska visas som ett streck mellan tornet och fienden. Strecket försvinner nästa frame.
6	Lägga till funktionalitet så att fiender skapas med jämna mellanrum vid startpunkten.
7	Lägga till hp (hälsopoäng) till fiender 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 kräver 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 torntyper. Den ena skadar alla fiender inom en viss radie och den andra skjuter en missil som exploderar och skadar alla fiender inom explosionen.
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 tillräckligt med resurser).
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 (poäng = 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 till exempel. 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	Lägga till så man kan sälja ett torn genom att först trycka på tornet och sedan trycka på en knapp vid sidan om uppgraderingsknappen. Man får tillbaka hälften av kostnaden för tornets totala kostnad med eventuella uppgraderingar.
18	Lägga till fler fiendetyper. En kan bara ta skada av missiltornet 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, till exempel. 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 uppgraderingsknappen.
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 en spelomgång startas.

23	Välja svårighetsgrad. Reglerar hur mycket torn kostar och hur ofta fiender skapas.
24	Lägga till så att spelet visas relativt fönstrets storlek (till exempel om man förstorar fönstret såtar spelplanen alltid upp hela fönstret).

4. Övriga implementationsförberedelser

Koden ska delas upp så att:

Tabell 2. Visar den tänkta klasstrukturen.

- En klass håller reda på vad som finns på spelplanen och på vilken position med hjälp av en tvådimensionell 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 abstrakt klass för funktionalitet och variabler som torn 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 längs x- och y-axlarna mellan utsatta checkpoints oberoende av rutnät.

5. Utveckling och samarbete

Såhär har vi tänkt angående samarbetet under projektet:

Tabell 3. Visar hur samarbetet planeras se ut.

- 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

När projektet färdigställts har dessa milstolpar uppnåtts:

6.1. Milstolpar

Tabell 4. Visar vilka av de milstolparna som listades i tabell 1 som har avklarats.

Skapa spelplanen som representeras av ett rutnät där varje cell antingen 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.

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

Avklarat.

Skapa checkpoints för fienders rörelse och funktionalitet så fiender kan röra sig längs en bana.

Avklarat.

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.

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 streck mellan tornet och fienden. Strecket försvinner nästa frame.

Avklarat, dock visas skotten under tre ticks.

Lägga till funktionalitet så att fiender skapas med jämna mellanrum vid startpunkten.

Avklarat.

Lägga till hp (hälsopoäng) till fiender så att det krävs minst ett skott för att döda dem. Tornen gör för tillfället 1 hp skada per skott.

Avklarat.

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

Avklarat.

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.

Lägga till två torntyper. Den ena skadar alla fiender inom en viss radie och den andra skjuter en missil som exploderar och skadar alla fiender inom explosionen.

Avklarat.

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 tillräckligt med resurser)..

Avklarat.

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

Ej implementerat.

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.

Lägga till så man kan spara highscore när man förlorat (poäng = 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.

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.

Lägga till så man kan uppgradera torn. Uppgradering kostar resurser och förbättrar till exempel. skada och räckvidd. När ett torn uppgraderas så får den ett nytt utséende. 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.

Lägga till så man kan sälja ett torn genom att först trycka på tornet och sedan trycka på en knapp vid sidan om uppgraderingsknappen. Man får tillbaka hälften av kostnaden för tornets totala kostnad med eventuella uppgraderingar.

Avklarat.

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.

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.

Lägga till ljudeffekter, till exempel. när man köper nya torn och när man skjuter.

Ej implementerat.

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.

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 en spelomgång startas.

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

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

Ej implementerat.

Lägga till så att spelet visas relativt fönstrets storlek (till exempel 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

letar 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 ritar svartvita boxar i fönstret. Klasserna SettingsFrame, HelpFrame och HighscoreFrame

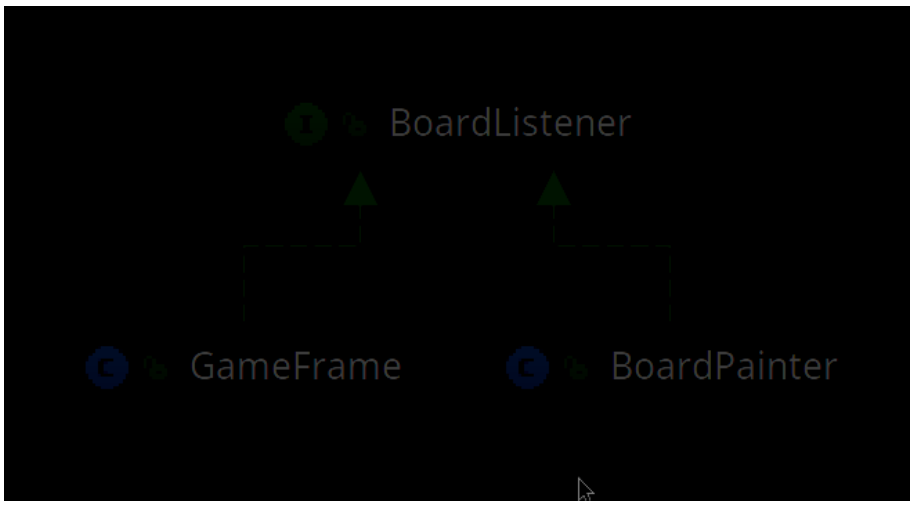
använder samma logik när det kommer till knapptryckningar, och de ritas upp med klasserna

SettingsPainter, HelpPainter och HighscorePainter.

När användaren är i SettingsFrame och klickar på en av knapparna (som väljer bana) så skapas en instans av Board 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.

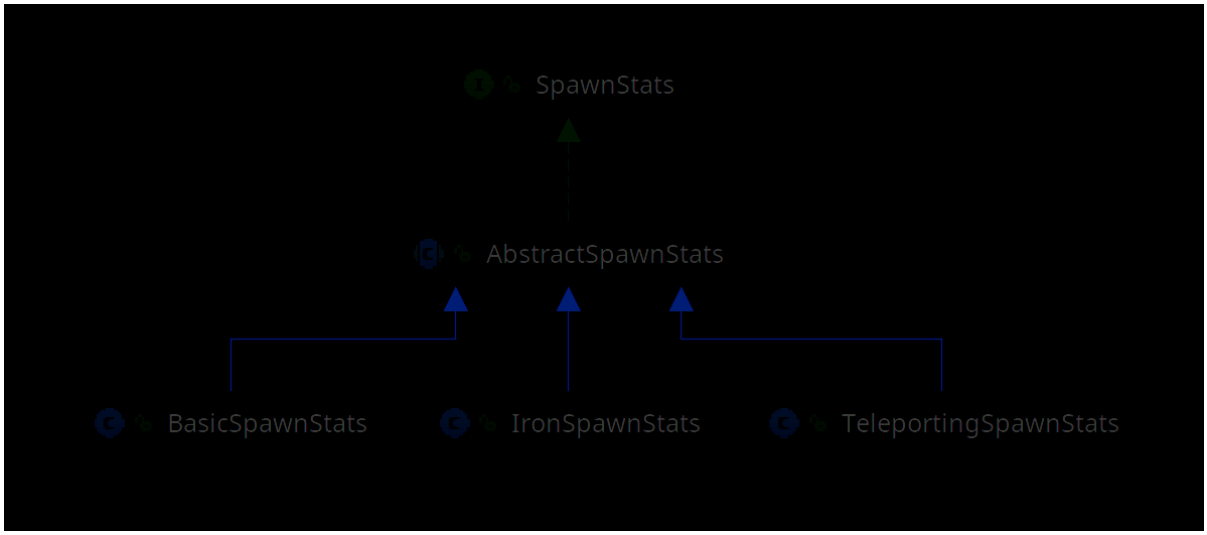
Bild 1. Klassdiagram som visar hur klasserna GameFrame och BoardPainter implementerar gränssnittet BoardListerner.



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

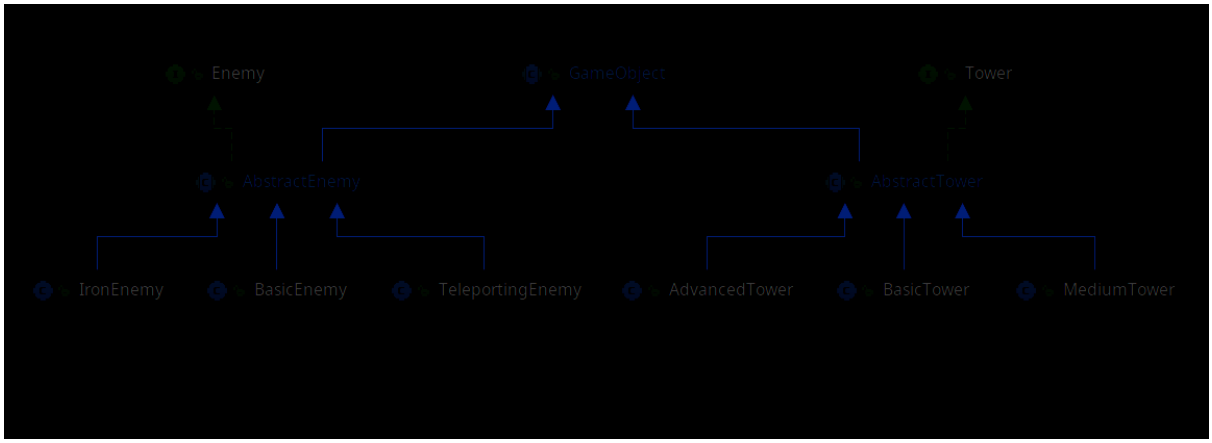
skapas eller fiender rör sig, kommer Board kalla på funktionen notifyListeners() för att notifiera de klasser som implementerar BoardListener om att spelplanen uppdaterats. NotifyListeners() kallar alla BoardListeners funktion boardChanged().
GameFrame kommer genom boardChanged() uppdatera etiketterna om liv, resurser, nuvarande runda och så vidare Den kommer även visa en inmatningsruta som frågar om namn och sparar poäng om användaren förlorat. BoardPainter kommer genom boardChanged() att rita upp hela spelplanen igen, det vill säga brädet, fiender, torn och attacker.

Bild 2. Klassdiagram som visar strukturen för SpawnStats (klasser som håller reda på hur varje fiende ska skapas).



Klasser som implementerar SpawnStats är klasser som beskriver för Board hur spelets fiender ska skapas, det vill säga 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 subklasser. Eftersom alla tre subklasser implementerar SpawnStats så kan man lägga de i en gemensam lista (List<SpawnStats> spawnStats). Då kan man till exempel. kalla på alla SpawnStats newRound()-funktion i listan med en for-loop istället för att explicit behöva skriva BasicSpawnStats.newRound(), IronSpawnStats.newRound()...

Bild 3. Klassdiagram som visar strukturen för GameObjects (fiender och torn).



Gränssnittet Enemy implementeras av AbstractEnemy, som har subklasserna 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 avséende 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 sparat 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.

Tabell 5. Visar hur objektorientering har använts i projektet.

- 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 abstactTower och abstractEnemy där funktionalitet som är gemensam för alla torntyper respektive fiendetyper är samlad. Längs 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 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 enumklass och olika villkorssatser.

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

- Som exempel har konstruktorn använts i klassen board för 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 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 används på många ställen, till exempel i klassen BoardPainter där klassen PaintComponent 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 den enklaste lösningen varit att ha en klass för fiender och sen låta till exempel getRadius 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 använt bland annat i klasserna AbstactTower och AbstractEnemy där vi låter de flesta fält vara “protected” för att de ska vara tillgängliga från subklasser. I andra språk utan objektorientering får man antingen låta fälten vara publika och på så sätt minska säkerheten, eller föra med fälten som parametrar till enskilda metoder.

- Att låta en abstrakt klass innehålla en standardrepresentation av en funktion och sedan ha en subklass som skriver över funktionen sker till exempel i klasserna AbstractEnemy och TeleportingEnemy. I AbstractEnemy finns en standardrepresentation för funktionen move(), men eftersom TeleportingEnemy ska röra sig annorlunda skriver den klassen över 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 upprepad kod.

6.5. Motiverade designbeslut med alternativ

Dessa designbeslut har vi tagit i koden:

Tabell 6. Visar de designbeslut som har tagits och motiverar varför.

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 SpawnStat-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 runda. Förändringarna av fälten sker i varje SpawnStats-klass funktion newRound(), som anropas 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 SpawnStats typ skapas. Denna process återupprepas tills currentAmount är lika med noll för alla SpawnStats.

En alternativ lösning till SpawnStats 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.

Vi tycker att vår lösning är bättre i det avseendet att den är mer objektorienterad, men sämre då rundorna inte blir lika balanserade som om man skulle hårdkodat 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 fönsterklass och en uppritningsklass för varje fönster. De klasser vi har är MenuFrame, HelpFrame, ScoreFrame, HighscoreFrame, SettingsFrame och GameFrame. Varje gång man kommer till en ny meny stängs det nuvarande fönstret, och en instans av det nya fönstrets klass skapas. Ett alternativt sätt vi hade kunnat välja är att ha olika uppritningsklasser men bara en fönsterklass. Man hade då kunnat ha enumvärden för de olika fönstrena och använt olika uppritningsklasser beroende på nuvarande enumvärde.

Vi valde inte den lösningen eftersom den var svårare att implementera. Det beslutet gav oss mer tid att fokusera på viktigare aspekter av projektet, som objektorientering.

Designbeslut 3: Representera torn i spelbrädet

Vi ville få ett sätt att representera spelets torn i spelbrädet. Vår lösning är att det finns 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 sätt vi valde att representera torn på 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 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 torn har funktionalitet som beskriver hur de ska bete sig. Dessutom 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 en fil och sedan gå att läsa av i efterhand. Vi valde att ha klassen HighscoreList som innehåller tre listor, en för varje bana, med Highscoreobjekt. Varje gång användaren avslutar en spelomgång sparas ett nytt objekt i listan i HighscoreList som motsvarar den spelade banan. Listan sorteras och den uppdaterade versionen av HighscoreList skriver över den gamla listan i filen.

En alternativ lösning är att HighscoreList bara innehåller listor med de nio bästa poängen och en flagga för det nuvarande lägsta poängen 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.

En annan lösning är att ha tre klasser där var och en innehåller en lista med toppoängen för en bana. Man kan då implementera ett gränssnitt och en abstrakt klass som innehåller gemensamma metoder och fält. 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 i.

Anledningen till att vi inte valde någon av de alternativa lösningarna var att vi inte tjänade på att ändra implementationen vi redan hade. Det gav oss mer tid till att arbeta med andra delar av projektet.

Designbeslut 5: Utseende

Trots att grafiken inte ligger i fokus i denna kurs ville vi ändå att spelet skulle se ganska snyggt ut, då vi tycker att det ger ett starkare helhetsintryck. Därför har vi i våra menyer en grafisk komponent som ritar ut fina boxar med text, istället för att använda Jbuttons.

En alternativ lösning är att med hjälp av MigLayout placera ut JButtons under varandra istället för att rita ut dem med en grafisk komponent. Med denna lösning hade vi behövt skriva mycket mindre kod, men samtidigt haft samma funktionalitet. Vi valde inte denna lösningen eftersom vi tycker att JButtons ser tråkiga ut.

7. Användarmanual

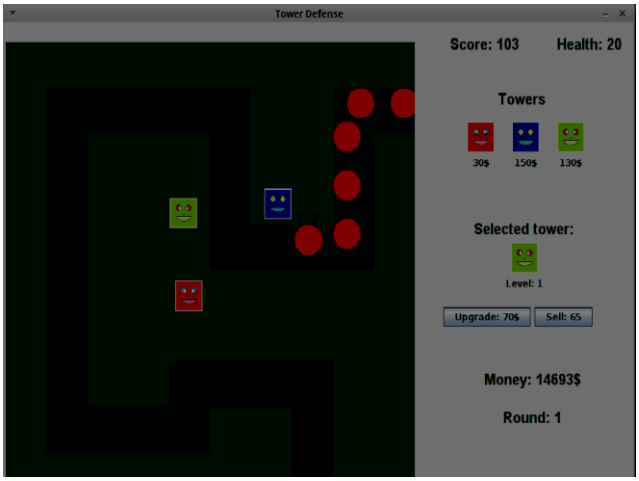
När spelet startas öppnas en huvudmeny. Där får spelaren välja mellan fyra olika alternativ: starta ett nytt spel, se vilka spelare som har högst poäng, få hjälp eller avsluta.

Bild 4. Visar startmenyn.

Om ett nytt spel påbörjas öppnas en ny meny upp där spelaren får välja bana. Det finns tre olika banor att välja mellan, och det som skiljer dem åt är fiendernas väg. När en bana valts startas en spelomgång. I panelen till höger visas spelarens liv, poäng, pengar, nuvarande rond och byggbara torn. Längs ner i panelen finns även knappar för att starta en ny rond, pausa och avsluta.

När en ny rond startas börjar fiender röra sig längs den bruna vägen. Målet är att förstöra fienderna genom att strategiskt placera ut olika sorters torn på det gröna området. Det görs genom att klicka på ett torn i sidopanelen och sedan placera tornet på önskad plats med hjälp av musen, förutsatt att man har tillräckligt med pengar. Skulle spelaren ha för lite pengar går tornet inte att placera ut och avmarkeras genom att klicka någonstans utanför spelplanen. När ett torn skadar en fiende får spelaren pengar och poäng.

Bild 5. Visar ett pågående spel.



Det finns tre olika sorters torn att välja mellan. Det rosa tornet är billigt och skjuter en fiende i taget. Det blåa tornet skjuter en missil som kan skade flera fiender samtidigt. Det gula tornet skadar alla fiender inom en viss räckvidd.

Utöver att köpa nya torn kan spelaren uppgradera respektive säja 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 som räckvidd, skada per skott och skjuthastighet. Ett torn kan som högst uppgraderas nio gånger.

Spelet har även tre fiendetyper som beter sig på olika sätt. Den röda fienden går normalt längs den bruna vägen. Denna fiendetyp förekommer varje rond. Den gråa fienden kan bara ta skada av missiltornet. Den sista fiendetypen teleporteras mellan vägens hörn, istället för att röra sig normalt.

Skulle en fiende lyckas ta sig förbi alla torn förlorar spelaren lika många liv som fienden har hp 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 toppoäng, som man enkelt navigerar till från startmenyn.

Bild 6. Visar de högsta poängen från bana 1.

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

Såhär har vi valt att utvärdera projektet:

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 oftast inte 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 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 tänk, och dels att vi stressade igenom en del moment lite för mycket, som vi senare var tvungna att justera.

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 genom att leta på nätet, kolla på videor 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.

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 punk 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 nämndes tidigare så följde vi inte vår arbetsplan till hundra procent utan lämnade rum för idéer som uppkom under projektets 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?
Börja i tid med projektet och se till att man har en bra objektorienterad bas innan man går in på områden som till exempel design.

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 hade också varit bra om det fanns lite mer information om 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 inlärning?
Vi är båda väldigt nöjda med upplägget av kursen och känner att vi har lärt oss väldigt mycket båda två.