System design document for Group 29

Sally Chung, Amanda Dyrell Papacosta,
Eric Erlandsson Hollgren, Jonatan Jageklint, Zoe Opendries

Oktober
version 2

1 Introduktion

Syftet med dokumentet är för att ge en teknisk bakgrund till projektet. Projektet i fråga är ett spel som blivit inspirerat av ”age of war” och ”tower defense” liknande spel, vilket riktas till att användaren får möjligheten att utveckla sina spel- och strategikunskaper. Spelet är nybörjarvänligt och ger en möjlighet för spelaren att utvecklas inom givna områden. Det som skiljer projektets spel från givna spel är att användaren kan styra sin 2D karaktär, vilket ytterligare utmanar spelarens strategitänk.

I följande avsnitt kommer beskrivning för projektets testnings metodlogi, mjukvaruarkitektur, val av mjukvarudesign och mer beskrivas.

1.1 Definitions, acronyms, and abbreviations

Definitions etc. probably same as in RAD

- Subscribers: Prenumeranter/lyssnare, används ofta i förhållande till observer-pattern
- MVC: Model-View-Controller, ett mönster som strukturerar upp system arkitekturen för projektet
- UML: Unified Modeling Language, visar visuellt beteendet och strukturen i ett system eller en process.
- Domänmodellen beskriver en översiktlig bild av modellen.
- Enum: en datatyp vars värden består av en fixerad mängd konstanter.

2 System architecture

I detta avsnitt kommer mjukvaruarkitekturen att beskrivas samt beskrivning kring hur spelet fungerar.

2.1 MVC - Model view Controller

Spelet är strukturerat kring arkitekturen MVC. MVC har tre större delar som beskrivs nedan i detalj.

2.1.1 Model

Kärnan i programmet är modellen. Modellen är där all funktionalitet, logik och data existerar. Den ska vara en modul som inte är beroende av något utanför sig självt för att fungera.

2.1.2 View

Views ger användaren en inblick i vad som sker i modellen. Varje view visar upp en egen del av modellen men är så pass abstraherade att de skulle kunna bytas ut både vid kompileringstid och körtid. Allt som användaren ska interagera med eller kunna se visas upp genom en view.

2.1.3 Controller

Controllers används för att koppla ihop vissa delar av views med modellen. Finns det en knapp spelaren kan trycka på eller det ska hända något när användaren trycker på piltangenterna är det controllernas uppgift att prata med och manipulera en del av modellen.

2.2 Applikationsflöde

Under en körning av applikationen börjar det med att en meny skapas i View, där kan spelaren välja att spela spelet eller avsluta. Väljer man att spela kommer Model och Controller att instansieras och sedan en View som använder referensvärden till instansierade Model-objekt och skapar en illustration av Model-komponenten.

När Model har instansierats så startar modellen en game-loop. Då models tillstånd ändras konstant uppdateras även View konstant genom att View-komponenten kollar hur Model uppdateras. Applikationens vy som spelaren ser finns även ett gränssnitt med knappar som spelaren kan klicka på. Dessa lyssnar View-komponenten till och när spelaren interagerar med dessa notiferas vyns Controller-komponenten att spelaren har gjort en handling. Därmed kan Controller tar in informationen om vad för handling som gjorts och informerar Model-komponenten att ändra sitt tillstånd.

När detta sker så är det View som illustrerar Models nya tillstånd.

För vissa handlingar som spelaren har tillgång till finns även villkor som måste uppfyllas innan modellen kan ändras. Exempelvis om spelaren vill uppgradera en komponent behövs spel-valuta som kan fås ackumulativt under spelets gång. Om spelaren inte har tillräcklig valuta så kommer, när modellen försöker hantera denna handling, inte någon ändring av delens tillstånd i komponenten ske. Modellen kommer att uppdatera en lista av felmeddelanden med ett meddelande varför villkoret inte var uppfyllt. Detta ser View och illustrerar denna information.

Gameloopen kan avslutas av sig självt, och det är meningen att spelarens syfte är att förhindra detta från att hända genom att spela spelet. Om Gameloopen avslutas så kommer detta observeras av View och en Game-Over-skärm visas.

3 System design

I detta avsnitt kommer designmönster som har använts i projektet att beskrivas.

3.1 MVC

I vårt projekt har fokuset varit kring att arbeta med MVC-pattern, med andra ord att varje implementation av klass ska ha en koppling till Model-View-Controller (modell, vy och kontroller). Varje instantiering av ett objekt sker i vår app-klass vilket är den delen av projektet som i vårt gradle-system skickas till en ”Desktop Launcher”. Denna i sin tur är det som visar upp spelets vyer vilket spelaren får tillåtelse att interagera med spelet. Genom att klicka på tangenter skickas signaler från kontrollen till modellen att något har skett. Därefter lyssnar vyn på modellen och renderar förändringen.

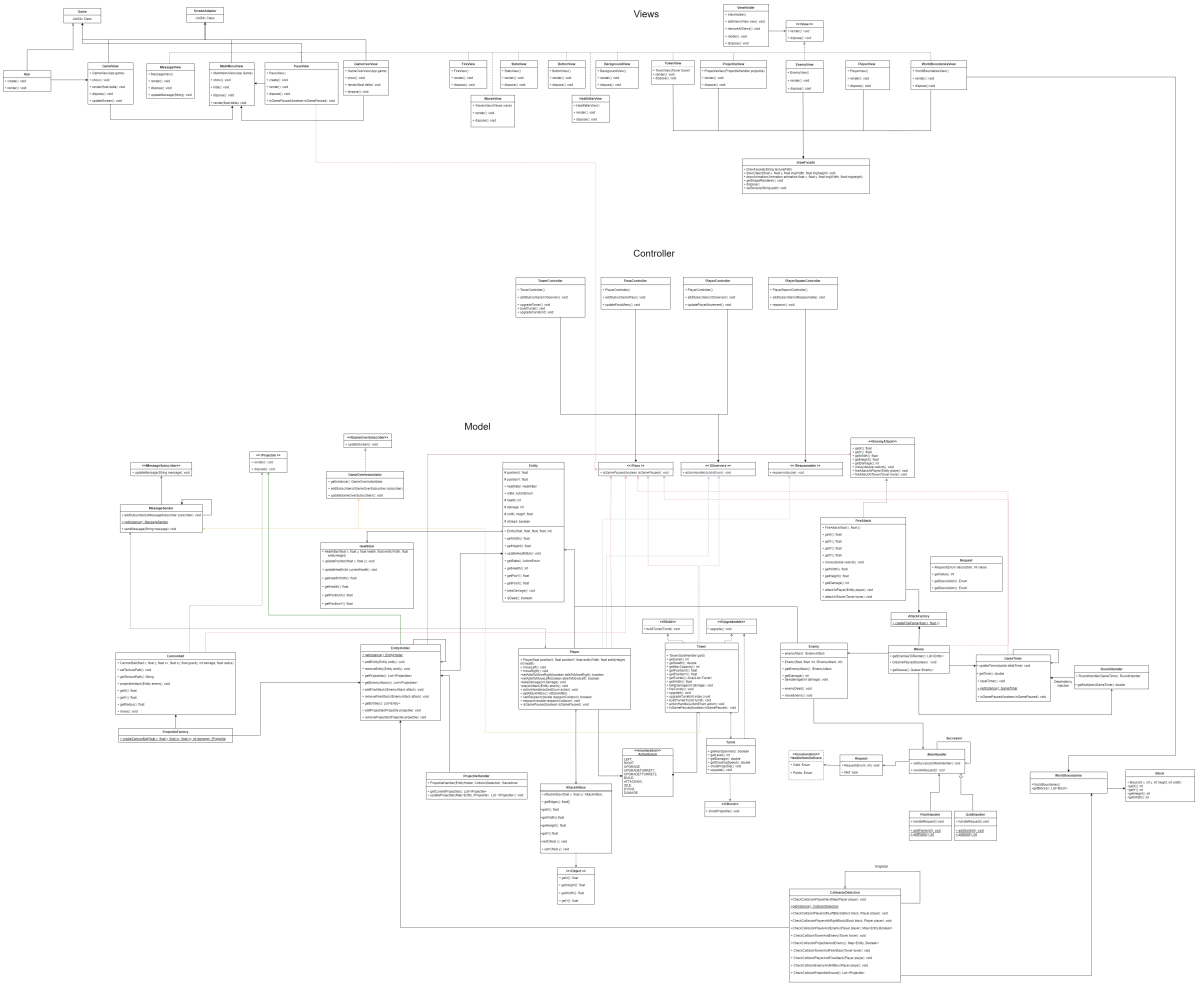

Figure 1: UML diagram över vårt spel

För en bättre bild av vårt UML diagram, fig. 1, hänvisar vi till denna länk genom att klicka här.

3.2 Relationen mellan UML diagrammet och domänmodellen

Både UML diagrammet och domänmodellen skapar en grafisk representation på arkitekturen av hur programmet är uppbyggt. Domänmodellen ger en övergripande blick på hur modellen fungerar i UML diagrammet. Den gömmer både controller samt view delen av MVC, men även hur systemet är uppbyggt och visar endast de centrala delar som användaren kommer att interagera med. Modellen i domänmodellen speglar relationerna i UML diagrammet som en helhet. UML diagrammet visar dessa relationer men gömmer inga delar så att varje klass syns. Här ser man de olika objektorienterade lösningar som vi har använt oss av.

3.3 Observer

Observers tillämpas i mån om att ha observatörer för när ett objektens status förändras. Detta ska kunna ske dynamiskt vilket tillämpar sig väl för våra controllers; PausController, PlayerSpawnController, TowerController och PlayerController använder sig båda av observer-pattern. Både TowerController och PlayerController implementerar interfacet IObservable vilket är en actionHandle av typen void som tar in enums av action.

Syftet är beroende på vilken ”action” en spelare väljer att göra, så ska den i sin controller-klass uppdatera objektets status. I sin controller klass är det därför fördelaktigt att låta den hantera vad för typ av inputs som sker när spelaren exempelvis väljer att gå höger eller vänster. Vidare beskrivet är det controller klassen som ser till att koppla samband action enumet för den typen av input spelaren ger. Exempelvis, i Player klassen finns en actionHandle metod som hämtar olika typer av enums som skickas till controllern. Dessa märker sedan av vilka knappar som spelaren håller ner, och vyn kommer därmed uppdatera genom att lyssna på modellen och renderar det för exempelvis player.

3.4 Singleton

Singleton används för en del klasser i modellen som vi bara vill att det ska finnas en instans av. Dessa är RoundHandler, GameTimer och EntityHolder. Gametimer används av olika delar av modellen syftet med GameTimer är att den ska räkna tiden som spelaren lyckats försvara basen samt att RoundHandler ska använda instansen till att beräkna bur mycket liv som fiender har. Därför bör det bara finnas en instans av GameTimer. EntityHolder har som uppgift att hantera listor med vilka objekt som

finns i modellen vid en given tidspunkt och bör därför också bara kunna instansernas en gång.

3.5 Chain of Responsibility

I projektet har chain of responsibility använts för att uppdatera spelet om poäng och guld. Enkelt beskrivet är det en kedja av requests som skickas igenom kedjan tills den tas emot. I kedjan finns det olika hanterare, ”handlers”, som bekräftar att den kan/inte kan ta emot requesten som skickats. I vårt projekt har vi tillämpat detta genom att implementera en MainHandler som en abstrakt klass vilka GoldHandler och PointHandler ärver av. De sistnämnda klasserna är så kallade ”handlers” som hanterar en specifik request. Skillnaden mellan dessa är att de kräver olika request beskrivningar; GoldHandler uppdaterar guldet genom att en request om ”GOLD” har specificerats och PointHandler fungerar likadant fast istället kräver den ”POINTS”, i sin request beskrivning.

Kedjan i sig fungerar genom att i exempelvis Enemy-klassen implementera MainHandler i sin klass. Därefter instantierar den en ny GoldHandler och PointHandler. För att detta ska fungera som en kedja, sätts en av dessa till en ”successor” av den andra. Till exempel kommer GoldHandler få point-requesten och skicka vidare till PointHandler för att hanteras. När en enemy dör ska då kedjan sättas igång, där både ”GOLD” och ”POINTS” ska uppdateras med det värde som tillkommer en enemy (när en enemy dödas fås x-antal poäng och guld).

3.6 Factory Pattern

Factory pattern är ett creational design pattern som används för att underlätta skapandet av nya objekt som är relaterade till varandra. I vårt program används detta mönster vid skapandet av projektiler och fiendeattacker.

I och med att spelet i nuläget är relativt grundläggande, med endast en typ av fiendeattacker och projektiler, kan mönstret kanske ses som onödigt. Däremot ger det möjligheten att enkelt utveckla spelet, dvs. implementera fler attacker och projektiler i framtiden, vilket är anledningen till att mönstret används. Genom att låta en klass ha ansvar för att skapa alla projektiler (ProjectileFactory) eller fiendeattacker (AttackFactory) gynnas det framtida utvecklandet av spelet. Hade ytterligare en projektil av annat slag behövt implementeras skulle det kunnat göras genom en ny create-metod i ProjectileFactory och sedan hänvisat till denna när en sådan projektil ska instansieras någonstans i projektet. Detta kan vara användbart om fiender ska ha flera olika attacker (exempelvis vid uppgradering).

3.7 SOLID

Solid principerna är något som alltid funnits i bakhuvudet under projektets gång. Principerna är följande; Single responsibility principle, open closed principle, Liskovs substitution principle, interface segregation principle och dependency inversion principle. Under utvecklingen av projektet har varje utvecklare haft eget ansvar över att följa ovanstående principer så gott det går och det har resulterat i att projektet har en stabil grund i SOLID principerna.

3.7.1 Single responisbility principle

Varje klass har i programmet endast ansvar över en process. Några exempel Player klassen som endast har ansvar över allt som angår spelaren, CannonBall som har asnvar över allt som angår en cannonball. Detta innefattar visserligen rörelse, storlek och skada vilket är olika saker men som alla hänger ihop med cannonball och därför kan defineras som inom samma ansvarsområde.

3.7.2 Open closed principle

Programmet ska vara öppet för utökning men stängt för modifiering. Detta har vi implementerat genom att så många variabler och referenser som möjligt ska vara privata och därmed inte kunna modifieras utifrån. Vi har även tänkt på att progammet ska vara lätt att utöka i framtiden därför har vi många abstrakta beroenden och inte konkreta implementationer. Chain of responsibility används också ifall vi i framtiden vill att fler saker ska hända när en fiende dör.

3.7.3 Liskovs substitution principle

I spelet som det ser ut just nu så finns det bara en typ av projektil men det finns möjlighet att lägga till fler därför finns det en abstrakt implementation av projektiler som heter IProjectile. Detta för att programmet under körtid inte ska bry sig om vilken typ av projektil som används utan att dessa kan bytas ut under körtid.

3.7.4 Interface segregation principle

Projektet har inga objekt som använder sig av interfaces där alla metoder i interfacet inte används.

3.7.5 Dependency inversion principle

Projektet använder sig mycket av dependency inversion principle eftersom observer pattern implementeras på många ställen.

4 Persistent data management

I detta kapitel beskrivs datahanteringen som har använts i projektet.

4.1 Images

Bilder som karaktärer, spelets bakgrund och objekt har använt finns i projektmappen ”assets”. Dessa bilder nås genom att instantiera drawFacade i en views konstruktor som kräver en sträng som parameter av ”new Texture” vilket är från LibGdx egna ramverk. Detta hämtar filen som refereras i strängen vilket sedan ritas upp i spelet.

5 Quality

För att säkerhetsställa kvalitén kring programmets funktion har tester skrivits i JUnit. I detta avsnitt kommer kvalitén angående testerna att beskrivas.

5.1 Tests and Issues

Vi har använt oss av JUnit för att skriva enhetstester till modellen. Dessa är lokerade i src/test/java

5.1.1 Issues

- Endast en turret skjuter projektiler
- När man trycker på ”restart game” knappen i paus menyn återställs inte fienderna.

5.2 Project dependencies

Test:

- JUnit ver. 5.1.1 the type is jar

Compile:

- Inga beroende finns utanför projektet från modellen. Däremot är vyerna och kontrollerna beroende av ramverket libGDX ver. 1.10.0

5.3 PMD

På programmet körde vi en PMD best practises analys och fick violations men dessa var antingen insignifikanta eller bara rent av fel.

5.4 Quality of testing

För att få en hög kvalité av spelets funktionalitet har vi sett till att skriva tester som är utom modellen. Till exempel har vi gjort en test vy för att testa alla observers som används i modellen.

Vid testning av kod har vi använt oss av metoden ”OAA” genom att följa strukturen: Ordna, agera och bekräfta. Den förstnämnda beskriver att vi konfigurerar våra objekt som ska testas, agera den typ av metod som vi exempelvis ska testa och konfirmera att det tillståndet som önskas antingen gäller eller inte.