Systemdesign dokument för

Författare: Jenny Carlsson, Daniel Persson, Joel Hilmersson & Elin Forsberg

Datum: 2021-09-24
Version 1.0

1 Introduktion

Målet med denna rapport är att beskriva strukturen av mjukvarusystemet för spelet *Chalmers Defense*. Detta kommer att göras genom UML-diagram och beskrivande text.

Chalmers Defense är en desktopapplikation i form av ett spel. Spelet är en typ av "tower defense" spel men med ett Chalmers-tema där målet är att eliminera coronavirus av olika varianter innan de tar sig till IT-sektionens sektionslokal Hubben 2.1.

1.1 Designmål

Målet med designen är att vi skall ha en applikation med hög sammanhållning och få sammanbindningar. Vi vill att applikationen skall vara testbar och ha goda möjligheter för expansioner. Till sist vill vi ha en design som återanvänder kod och har få beroenden.

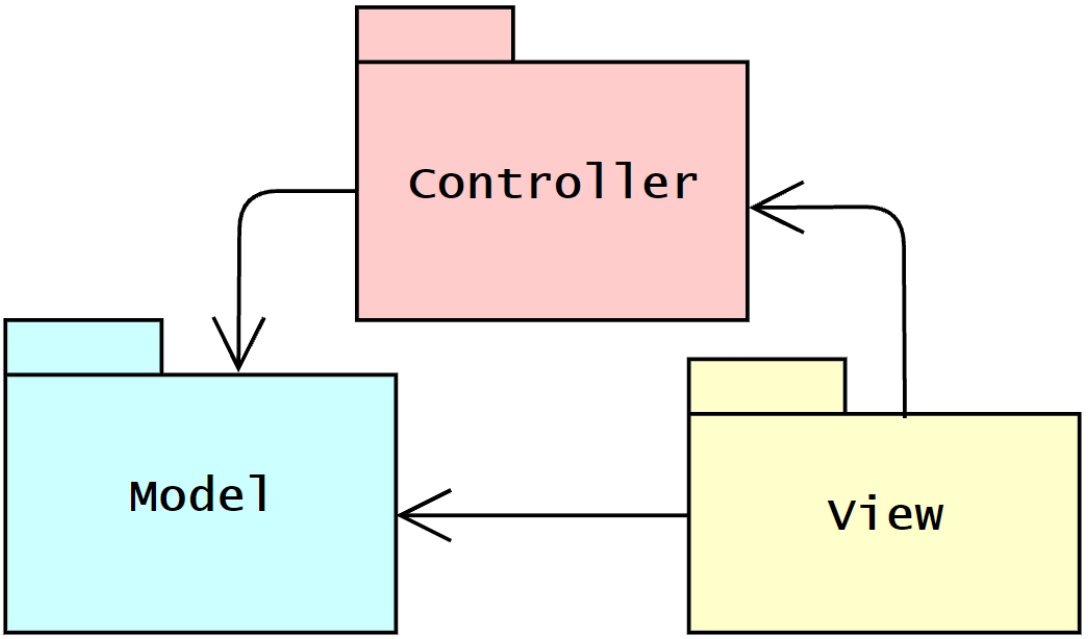
1.2 Definitioner, akronymer, och förkortningar

- JUnit
    - Inbyggda ramverket för enhetstestning av programmeringsspråket Java.
- Chalmers Defense
    - Namnet på vårt program.
- JSON
    - Javascript object notation.
- Javadoc
    - Verktyg för att generera dokumentation i HTML-format utifrån kommentarer i källkod.
- Huvudbransch
    - Den officiella "grenen" för projektets källkod där själva programmets färdigskrivna kod finns.
- Power-ups
    - Funktion i spelet som ger spelarens torn en förhöjd förmåga av något slag, exempelvis extra stark i sina attacker.
- Spawn
    - Ett uttryck för att skapa eller generera något.
- Rendera
    - Används för att beskriva när programmet visar något på skärmen.
- Targeting

    - Uttryck för målinriktning som i detta fallet syftar på ett torns inställning att prioritera målen de skall attackera.
- UML-diagram
    - "Unified Modeling Language"- diagram är ett diagram som representerar strukturen av ett program på ett objektorienterat språk för modellering.
- IT-sektionen
    - Syftar på IT-programmet på Chalmers tekniska högskola.
- SMART
    - All domän logik bör ligga i modellen, därför brukar den kallas för smart.
- DUM
    - En vy ska inte utföra egna beräkningar, bara rendera baserat på direktiven från andra. Därför kallas den ofta för dum.
- TUNN
    - En kontroller ska endast hantera extern inmatning, som oftast är inmatning från användare. Det bör vara ett tunt lager mellan användare och program, därför kallas kontroller för tunn.
- Runda
    - En runda i Chalmers Defense börjar med ett klick på startknappen. Då börjar virus spawna in och åka igenom banan. Det finns alltid ett fixerat antal virus per runda, så rundan är avklarad när alla har spawnat och antingen dött eller kommit till slutet.
- Separation of concern
    - Är en designprincip som innebär att ett modul ska endast ha en väldefinierad uppgift.
- GUI
    - GUI (Graphical User Interface) är ett användargränssnitt där användaren kan interagera med programmet med hjälp av visuella indikatorer.
- HUD
    - I spel så är HUD (Heads-up display) en metod för att visa visuell information till användaren som en del av spelets GUI.
- Overlays
    - I Chalmers Defense används en overlay för att visa vinstvyn och förlorarvyn. En overlay innebär att man lägger något visuellt ovanpå den aktuella skärmen men täcker inte allt. I detta fall läggs vinn- och förlorarvyn som en ruta i mitten av skärmen som användaren kan interagera med.
- libGDX

    - Vårt primära bibliotek för att rendera spelet på skärmen.
- LabelStyle
    - En libGDX class som ger varje textelement en specifik stil beroende på font storlek, font stil och färg.
- Sprite
    - En libGDX klass som håller information över position, rotation samt storlek över en viss bild (kallad "Sprite") som kan renderas till fönstret.
- Custom exception
    - En egenskapad undantags klass för användning inom ett visst syfte när javas standard undantag inte passar.
- Tower defense-spel
    - Ett strategispel som går ut på att försvara en slutdestination mot fiender genom att bygga torn av olika slag. Spelaren kan köpa nya torn med hjälp av pengar och om en fiende tar sig igenom banan förlorar spelaren liv.

2 Systemarkitekturen



Figur 2.1: Arkitekturen av programmet

Programmet är uppdelat i fyra delar enligt MVC:

- Model: är SMART och står för logiken för programmet & klasserna. Denna är oberoende av de andra.
- View: är DUM och inkluderar det som presenteras utåt, det vill säga gränssnitt etc. Den hämtar endast listor från Model och agerar på dessa.
- Controller: är TUNN och hanterar användarinput som sedan direkt skickas till Model för vidare behandling inom programmet.

2.1 Programflödet

You will to describe the 'flow' of the application at a high level. What happens if the application is started (and later stopped) and what the normal flow of operation is. Relate this to the different components (if any) in your application.

Programmet startas genom klassen *Chalmers Defense*. Den initialiserar allt som behövs för att få upp den första vyn som spelaren ser, vilket är huvudvyn. Klassen för huvudvyn fortsätter sedan med att bygga upp knappar och bilder som ska renderas. När användaren klickar på startknappen går programmet vidare genom att lägga upp en ny vy (spelvyn) och renderar de olika komponenterna som behövs på denna vy.

Programmet väntar nu på signaler från kontroller klasserna. När spelaren interagerar med en av knapparna i GUI:t så kommer kontroller klasserna att reagera genom att delegera till *model* som tar in vad som har blivit klickat på och agerar därefter. Antingen hanteras det själv av *model* eller så delegeras det vidare till andra klasser inom modulen *Model*.

En runda startas genom att användaren klickar på startknappen. När detta händer skickar kontroller startknappsanropet till *model* för vidare hantering. Model startar i sin tur upp uppdateringsklockan i klassen *GameTimer* som kontinuerligt kallar på updateringsmetoden i *Model* så att alla subkomponenter kan uppdateras. Direkt efter klockuppstarten anropas också *SpawnViruses* klassen med den aktuella rundan från *Player* till att börja spawna en förprogrammerad virussekvens av virus. Dessa virus börjar sedan att röra sig utefter banan som finns i spelet.

En runda avlutas när det inte finns några virus kvar på banan. Virus kan antingen ha blivit dödade av torn som spelaren har placerat eller så har viruset kommit igenom hela banan. När rundan är avlutad säger *model* till *gameTimer* att sluta kalla på "*updateModel*" vilket resulterar i att allt pausas.

Spelaren kan både under rundor och mellan rundor när som helst välja att dra ut ett nytt torn från högermenyn och placera dessa på spelplanen. Tornen kommer sedan beroende på om rundan är aktiv eller ej att direkt börja fungera som tänkt utan några problem.

Om spelaren önskar att avsluta spelet kan detta göras när som helst genom att navigera tillbaka till huvudmenyn och därifrån klicka på avsluta spelet vilket tar användaren tillbaka till skrivbordet.

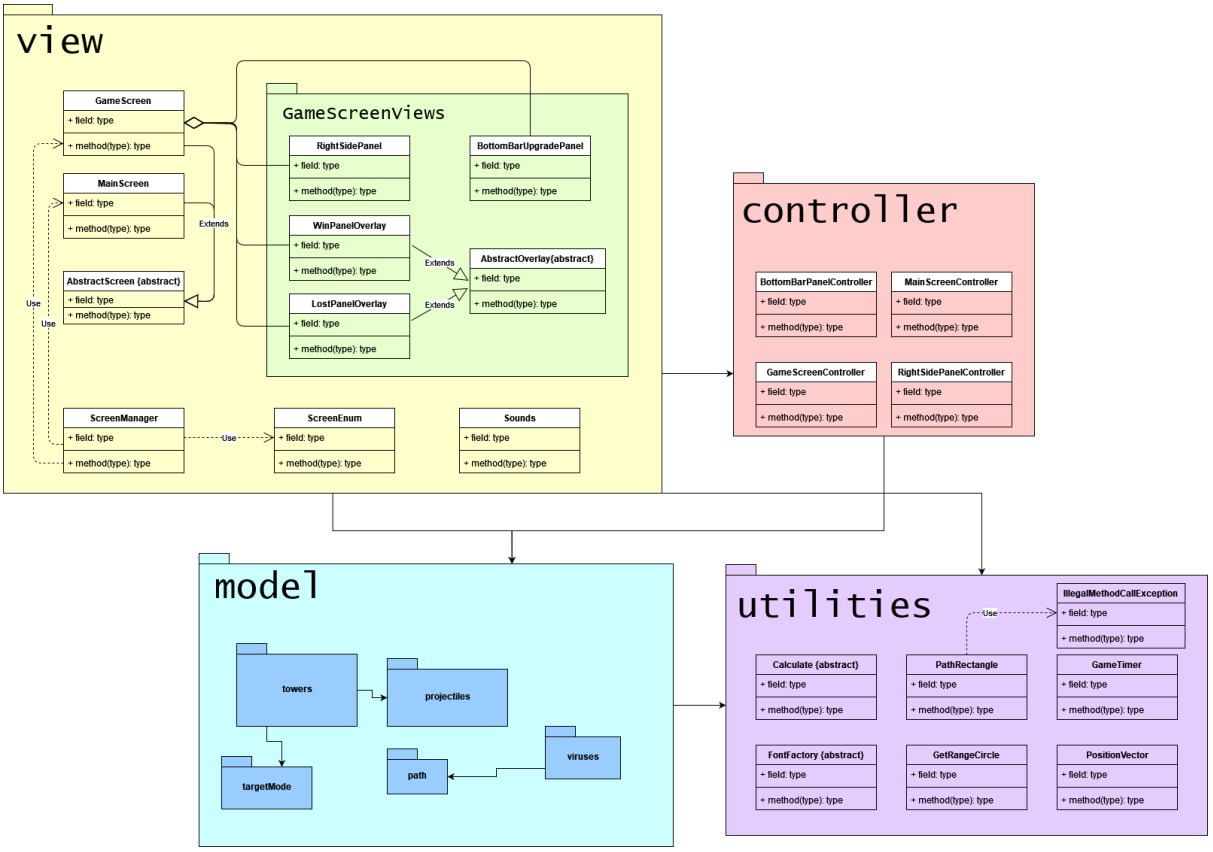
För renderingssystemet till applikationen används *libGDX* biblioteket. Klasser relaterade till detta ligger i View modulen och körs oberoende av Model-modulen. Detta innebär att renderingen av skärmelementen och uppdateringen av spelets inre funktionalitet görs separat vilket gör att antalet renderade bilder per sekund inte påverkar uppdatering av modellen tillsammans med en rad andra fördelar.

För att få detta system till att fungera har MVC:n satts upp genom att tillåta View fråga Model efter de objekt som ska renderas, därefter *Model* skickar den relevanta datan till View för att bli visad. Detta resulterar i att View under normala körförhållanden frågar Model efter objekt att rendera 60 gånger per sekund medan Model oberoende av nuvarande uppdateringshastighet skickar objekten direkt till View:n för rendering.

3 Systemdesign

Programmet *Chalmers Defense* följer designmönstret MVC för arkitekturen, vilket står för Model-View-Controller. Detta är inte den mest optimala systemarkitekturen för ett spel men det skapar en god modulär design för applikationen.

3.1 MVC



Figur 3.1: Mapp-diagram över programmet

Arkitekturen var skapad med designprincipen "*separation of Concern*" i åtanke. Detta genom att uppdelningen i de tre modulerna baseras på vad som hanteras och vad för typ av innehåll det är.

3.1.1 Relationer mellan modulerna

Relationerna mellan modulerna sker främst via gränssnitt. Både View och Controller använder ett eget gränssnitt för att komma i kontakt med Model, vilka Model implementerar. Detta gör att man gömmer funktionalitet som ej används bakom gränssnitten så att de ej kan nås. Gränssnitten gör det också lättare att kunna byta ut konkreta lösningar i framtiden om så skulle viljas utan att behöva ändra övrig implementation inom applikationen.

3.2 Model

Model är en oberoende modul. Det innebär att model kan klara sig på egen hand och dess View och Controller är utbytbara. Modellen står för spelets logik och hur de olika inre delarna samspelar med varandra för att utgöra spelet.

3.2.1 Intern arkitektur

Den interna arkitekturen i Model är i sig indelad i mappar, samt har några självstående klasser och gränssnitt. Detta är för att fortsätta på en modulär design, även i lägre nivåer av programkoden. Detta gör att koden följer (INSERT DESIGNPRINCIP).

3.2.2 Skapandet av objekt

- Beskriv vilka objekt som finns
    - Inom modellen finns objekt som står för programmets funktionalitet. Huvudingången sker genom klassen *model* som delegerar arbetet vidare till andra klasser genom designmönstret "Composite". De övergripande klasserna inom modellen är *Player, Rounds, Map och SpawnViruses. Player* tar hand om spelarens liv och pengar genom metoder som kan modifiera detta. *Rounds* håller koll på vilken runda spelet befinner sig på samt vilken runda som man vinner vid. *Map* tar hand om spelplanens objekt, vilket består av projektiler, torn och virus. *SpawnViruses* används för att spawna korrekta sekvenser av virus som sedan åker ut på banan.
- Beskriv hur dem skapas
    - I Model modulen skapas objekten främst vid uppstart av programmet då de relevanta klassernas objekt skapas i samband med att *model* klassen skapas.

      Undantag av detta utgörs av Torn, Projektiler och Virus objekt som alla skapas efter uppstarten av programmet. Torn objekt skapas genom att spelaren drar ut torn från högermenyn i spelvyn medan Projektiler objekt skapas av Torn klassen när en runda är aktiv. Virus objekten skapas av *SpawnViruses* klassen när en runda har startats.
- Path

3.2.3 Objekt och positioner på spelkartan

Objekt som ska ha en position på spelplanen implementerar alla gränssnittet "*IMapObject*". Detta gränssnitt består av metoder som hämtar relevant information såsom position, höjd och bredd samt rotation från objektet för att senare användas av View modulen för rendering. Viktigt i gränssnittet är också en metod som hämtar vilken nyckel som ska användas till att hitta korrekt Sprite som representerar objektet i View:n.

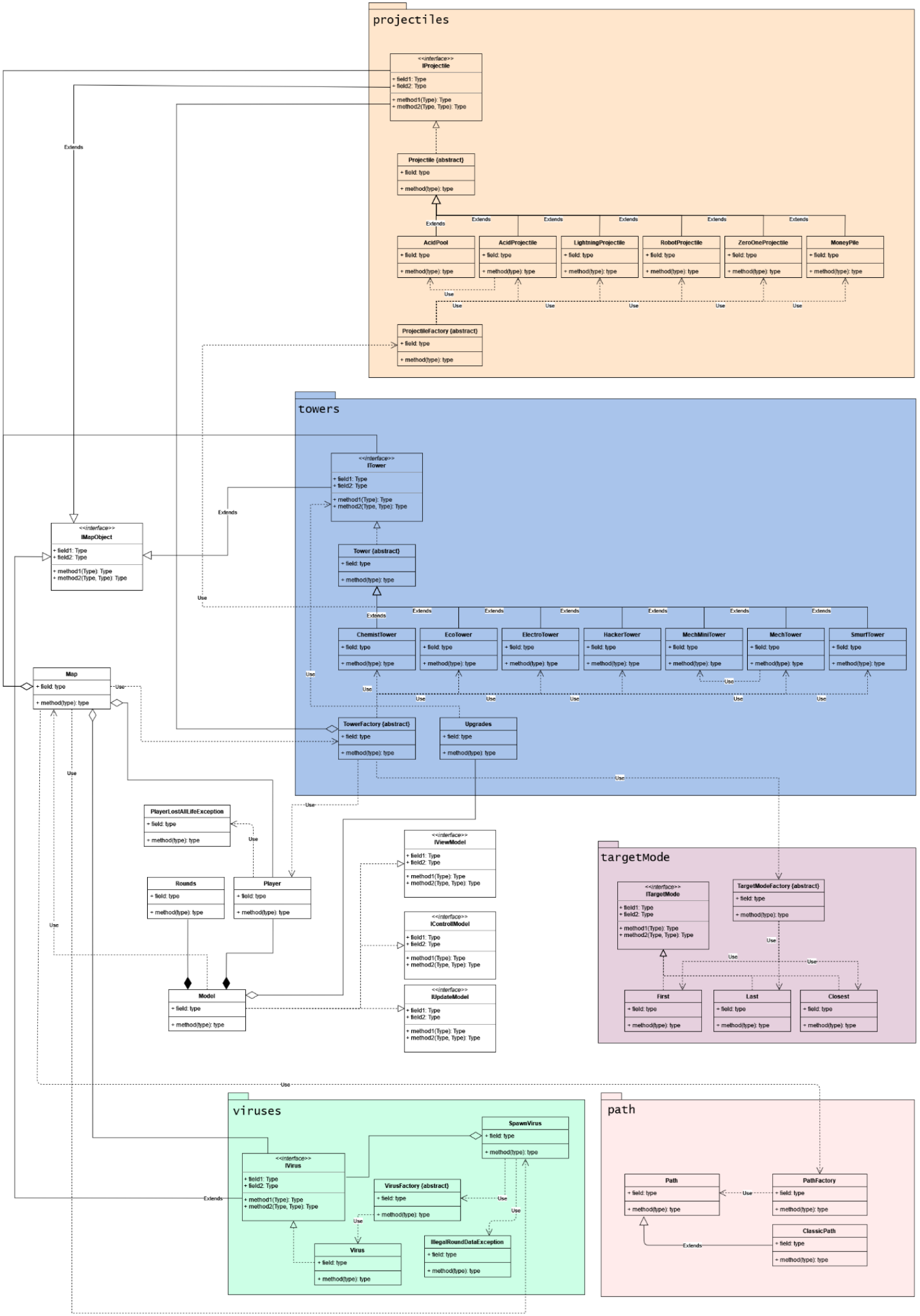
För att påverka objektens position används olika sub-gränssnitt till "*IMapObject*" som bidrar med metoder för att kunna uppdatera relevant information inom objektet där position ofta ingår. Objekten hanterar då uppdateringen genom att på sitt eget sätt uppdatera positionen på sig utefter eger sätt.

3.2.4 Utåtgående dependencies från Model

- JSON bibliotek
    - För att hantera och läsa JSON data för uppgraderingar behövs ett JSON bibliotek som hjälper till med detta. Därför är vi tvungna att använda ett extern bibliotek för att göra detta.
- Utilities
    - Utilities används för att underlätta för Model att göra vissa saker. Dessa verktyg är skapade av oss och är inte externa.

3.2.5 Relationen mellan designmodellen och domänmodell

Domänmodellen och designmodellen är starkt relaterade. Bortsett från *GUI*-delen i domämodellent, kan varje klass i domänmodellen återfinnas i designmodellen. Även relationerna mellan klasserna är densamma mellan de två modellerna. De skillnader som finns mellan de två modellerna är att designmodellen har en högre detaljnivå och därmed finns fler klasser, samt fler gränssnitt inkluderade.



Figur 3.2: Klassdiagram över Model-modulen. Även kallat designmodellen.

3.3 View

View är modulen för allt som skall "presenteras" för användaren. Detta innefattar allt användaren kan se och höra. *Chalmers Defense* View-mapp innehåller fem klasser och en mapp, *GameScreenViews*, som innehåller ytterligare fem klasser.

3.3.1 Klasser i Views mappen

ScreenManager är huvudklassen där alla vyer hanteras. *ScreenManager* är en *Singleton* och får in alla vyer den ska hantera när *initialize* metoden körs. För att byta mellan vyer används metoden *showScreen* som tar in en *ScreenEnum* och byter till respektive vy.

- *AbstractScreen*
    - *AbstractScreen* är en abstrakt superklass som håller i alla gemensamma metoder för alla vyer. Klassen ärver från libGDX *Stage* och implementerar libGDX *Screen* gränssnittet. Det gör att klassen och alla subklasser kan renderas med hjälp av libGDX.
- *GameScreen*
    - *GameScreen* är en konkret vy som ärver från *AbstractScreen* och har som uppgift att visa all spelrelaterad information på skärmen. Detta inkluderar en karta över spelplanen, alla *IMapObjects* som ska visas på kartan och ett HUD där spelinformation visas.
- *MainScreen*
    - *MainScreen* är också en konkret vy som ärver från *AbstractScreen*. Denna klassen är programmets ingångspunkt och har flera navigationsknappar för att ta sig runt i programmet.

3.3.2 Klasser i GameScreenViews mappen

GameScreenViews innehåller flera stycken paneler som används av *GameScreen.*

- HUD
    - Sidopanelerna i programmet är uppdelade i två separata klasser. Den ena är torn hyllan där torn och startknapp visas. Den andra är uppgraderings panelen som visas när man har klickat på ett torn.
- Overlays
    - Overlays används för att visa en ruta mitt på skärmen med information till användaren. Detta används när en spelare vinner, förlorar eller pausar spelet. Varje overlay skiljer sig i design men har liknande funktionalitet. Därför finns det en abstrakt klass som alla dessa overlays ärver.

3.4 Controller

Controller-modulen behandlar all användarinput i programmet och skickar vidare till Model för hantering. Den innehåller fyra klasser:

- *BottomBarPanelController*
    - Hanterar all input som kommer från bottenspanelen. Tar inputs från exempelvis uppgraderingsknapparna och knappen för att sälja torn.
- *MainScreenController*
    - Hanterar all input som kommer från startvyn. Här tar den inputs från exempelvis startknappen som tar spelaren till spelvyn.
- *GameScreenController*
    - Hanterar all input som kommer från spelvyn (exkluderat paneler). Inputs från pausknappen lyssnas efter här som ett exempel.
- *RightSidePanelController*
    - Hanterar all input som kommer från sidopanelen. Den lyssnar efter inputs från exempelvis tornknapparna, som används för att lägga till torn.

3.5 Utilities

Utilities är en mapp innehållande "assisterande" klasser. Att dem assisterar menas med att de ofta genomför en uppgift, som sedan en annan klass använder för att utföra en annan uppgift.

De klasser Utilities-mappen innehåller är:

- *Calculate*
    - Hjälper till med diverse beräkningar.
- *FontFactory*
    - Underlättar att sätta LabelStyle för typsnittsanvändning. Denna klass används i View-modulen.
- *GameTimer*
    - Innehåller metoder som är relaterade till spelets timer. Den omsluter ett timerobjekt i klassen. Timern kallar sedan på en uppdateringsmetod i den givna modellen.
- *PositionVector*
    - Representerar en punkt / vektor med hjälp av koordinater. Objekten av klassen är efter skapandet icke-muterbara och kan endast ändras via "*mutate-by-copy*".

- *GetRangeCircle*
    - Används för att rita ut en grå cirkel runt tornen. Klassen innehåller metoder om var och hur detta görs.
- *PathRectangle*
    - Omsluter en javarektangel och gör den klassen icke-muterbar. Gör detta genom att ta in värdena i en konstruktor men ger sedan ingen möjlighet till att ändra dess värden.
- *IllegalMethodCallException*
    - Denna klass innehåller ett custom exception för *PathRectangle*.

3.6 Designmönster

Chalmers Defense har använt sig av objektorienterande designmönster där de har varit applicerbara. Implementerade designmönster beskrivs nedan.

3.6.1 Model-View-Controller (MVC)

MVC är ett arkitekturmönster som går ut på att dela upp programmet i tre moduler. Dessa är Model, View och Controller. Model hanterar datan, View hanterar det presenterade innehållet och Controller hanterar användar-inputs. Vi använder *MVC* som vår övergripande programstruktur.

3.6.2 Factory Method

Factory method är ett skapande mönster. Det är ett "interface" för att skapandet av ett objekt i en superklass. Subklasserna alternerar vilken typ som skall skapas och implementeras genom att göra en factory-klass med statiska metoder, som beroende på metod skapa olika typer av objekt. Den returnerade typen av objekten ska vara en abstraktion av objektet med nödvändig funktionalitet för klienten. I *Chalmers Defense* används factory method på flera ställen. Det används för skapandet av torn, projektiler, virus, nya banor och fonter.

3.6.3 Template method

Template method är ett "Beteende"-mönster som används när man klasser har stor bitar kod gemensamt men med små saker som skiljer i dess sammanhang. Mönstret utnyttjar *implementation inheritance* då det flyttar gemensam kod till en abstrakt klass och det är abstrakta metoder som representerar variationerna. *Template method* används inom applikationen när ett torn ska skjuta en projektil. De flesta torn

använder samma huvudmetod för hanteringen men ska skjuta olika projektiler vilket implementeras i de konkreta torn klasserna individuellt.

3.6.4 Composite

Composite (översättning: sammansatt) är ett mönster som gör att man kan hantera en samling objekt som om det vore ett objekt av samma typ. Det är användbart när man modifierar och skapar ett flertal objekt på samma sätt med nästan identisk kod. Det gör att vi kan återanvända kod för alla omslutna objekt i *composite*-klassen och minskar komplexitet. Mönstret implementeras genom att skapa en klass som har en samling av underklasser som metodanrop genom huvudklassen delegerar till. I Chalmers Defense fall utgörs modulen Model löst hela Composite trädet med basklassen *Model* som rot. Metodanrop som görs till *Model* delegeras i det flesta fall vidare till underklasser som hanterar det specifika metodanropet vilket följer huvudidén med designmönstret.

3.6.5 Singleton

Singleton är ett mönster som används när man bara vill ha en möjlig instansvariabel(en singleton) för ett objekt/modul. Det implementerar detta genom att skapa en privat static variabel av samma typ som klassen, gör konstruktorn private, och sen en statisk metod som returnerar den statiska variabeln, och skapar om det inte finns någon. Singleton löser problemet när man måste skapa ett objekt flera gånger och är då användbart vid skapandet av databas-connections eller filhanterare.*Chalmers Defense* använder detta mönster för *ScreenManager*.

4 Persistent data management

- Bilder
    - Bilder lagras som png filer i mappen "\core\assets" och är sedan uppdelade i fler mappar så som *towers*, *viruses* och *projectiles.*
- Upgrade skins
    - Alla knappar i applikationen är skapade med verktyget *Skin Composer*. Verktygen genererar tre olika filer som lagras i en mapp med ett passande namn för knappen.
- Musik

    - Musiken lagras i ".waw" format.
- JSON
    - Uppgraderings data för varje torn sparas i en JSON fil. Filen är uppdelad efter torn namn och varje namn har sedan en JSON lista med uppgraderingar

5 Kvalité

• Describe how you test your application and where to find these tests. If applicable, give a link to your continuous integration.

Testen ligger i "test" mappen under huvudmappen "desktop"

• List all known issues.

Om man klickar på space innan en runda så kan man inte starta en runda efter.

• Run analytical tools on your software and show the results. Use for example:
– Dependencies: STAN or similar.
– Quality tool reports, like PMD.

NOTE: Each Java, XML, etc. file should have a header comment: Author, responsibility, used by ..., uses ..., etc.

6 Referenser

6.1 Verktyg

- Github
    - Webbaserad och centraliserad lagring av versionshistorik för programvaruutvecklingsprojekt.
- Slack
    - En kommunikativ plattform för att diskutera intern information gällande projektet.
- Figma
    - Webbaserad plattform för att designa gränssnitt.

- Zoom
    - Verktyg för digitala möten.
- Google Drive
    - Textredigeringsprogram, som tillåter delning och parallellt arbete med live-uppdateringar.
- Trello
    - Projektplaneringsverktyg
- Diagrams.net
    - Webbsida för att skapa och redigera UML-diagram
- JUnit
    - Java bibliotek samt verktyg för att testa skriven kod.
- Travis CI
    - Verktyg för att automatisera byggandet och testandet av JUnit tester.
- Javadoc
    - Ett system som autogenererar kommentarer från koden på ett standardiserat sätt till en sammanställning.
- Skin Composer
    - Ett verktyg för att skapa knappar med visuell respons.

6.2 Bibliotek

- JUnit
    - Är ett ramverk som programmerare i Java använder för enhetstestning.
- libGDX
    - Är ett gratis plattformsoberoende Java- spelutvecklingsramverk. Det är ett javabibliotek som ger stöd för bland annat grafik, inmatningar och ljud.