Reflektionsdokument; Space Invaders

Objektorienterad analys

Vi kom igång med projektet genom att ladda ner, kika genom och försöka förstå sig på allt i det exempelprojekt som fanns tillgängligt för oss. Vi valde därpå att nyttja så mycket som möjligt av denna. Vår objektorienterade analys grundade sig främst i att vi diskuterade spelets funktioner oss två emellan och från det vilka klasser vi kunde komma att behöva för att uppfylla dessa, utöver det som redan fanns i exempelkoden för projektet. Således använde vi oss inte av vare sig Use-cases eller CRC-kort rent praktiskt, men däremot så diskuterade vi noga genom de funktioner, med tillhörande klasser, som vi skulle kunna tänkas vilja ha i det slutgiltiga spelet i ett tidigt stadie. Detta arbetssättet fungerade i huvudsak bra, men bristen av exempelvis CRC-kort gjorde att vi frekvent hoppade mellan klasser, exempelvis för att skapa nya metoder, bland annat getters och setters, men även för att lägga till parametrar och variabler m.m.

Systemets design

Det som man främst kan ta upp vad avser designen av vårt system är att allt kretsar kring klassen GameState. Från GameState så extendas det till 6 stycken subklasser som alla ärver av GameState. I dessa klasser så finns det gemensamma variabler i form av en model och en bild. Det finns även gemensamma metoder som sköter allt från inmatningen på tangentbord och/eller mus till uppdateringssekvensen och utritning. Från exempelkoden behöll vi även klassen för konstanter som innehåller hur många monster vi displayar per rad och totalt, samt skärmens storlek i längd och bredd. För att skapa själva spelandet så kallar PlayState antingen på en längre uppdateringssekvens och utritning av en array med bilder som antingen kommer från bana 1 eller bana 2, vilket är uppdelat i 2 olika klasser. Om det är bana 1 eller 2 som kallas på avgörs i menyerna där man får välja “Level” som settar en boolean i modellen. Vid det scenario att vi skulle gjort flera stycken nivåer i spelet hade detta system sannolikt behövt bytas ut, exempelvis mot en array av booleans eller en HashMap. För det objektorienterade skapades klassen OwnImage som lagrar en bild, dess position i x och y-led, samt dess bredd och höjd. Detta användes för att kunna rita ut alla images i nivåernas utritningmetod utan att behöva göra fullständig uppräkning, vilket även det hade blivit svårt då man inte hade samma bilder i arrayen varje gång spelet uppdaterade sig.

Svårigheter

De största svårigheterna vi hade till en början var att dela upp arbetet på ett bra sätt. Målet var ju givetvis att kunna sitta och programmera på olika delar av spelets kod, men detta kunde kännas svårt till en början. Efter ett tag fick vi till detta mycket mer effektivt och arbetade sällan i samma klasser. En annan svårighet som vi upplevde

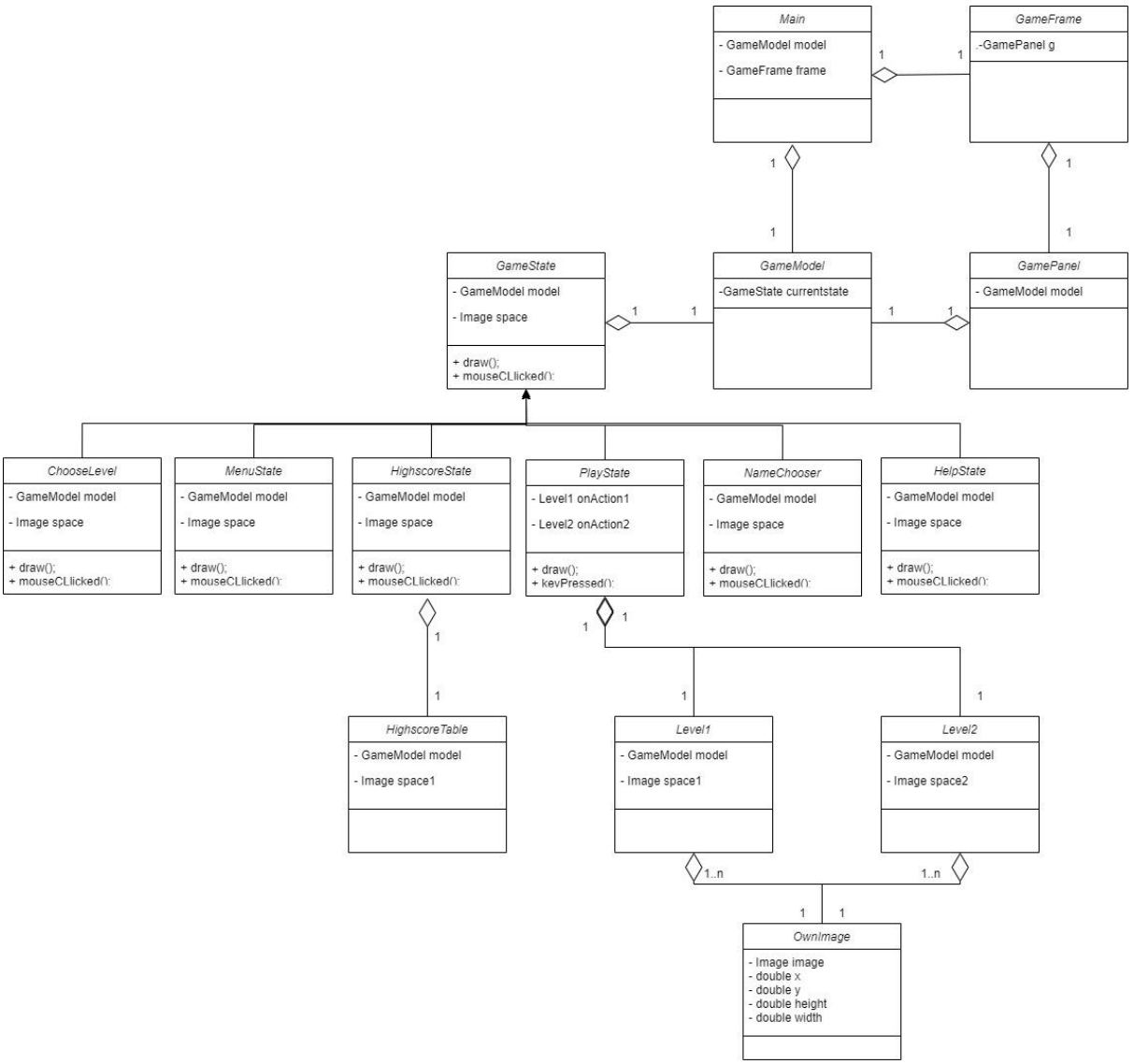
under projektarbetet var att man i stor utsträckning kunde sitta och koda utan att nå några konkreta framsteg i spelet funktion. Detta kunde bli väldigt frustrerande och demotiverande.

Hjälpmedel

Vi har under hela projektets gång använt oss av git för att synka vårt arbete. Att få git att fungera kändes först väldigt svårt att implementera i programmerandet och stökigare än att maila koden sinsemellan (vilket vi hade gjort dessförinnan). Efter hand så förstod vi hur man använder git och från det har vi upplevt det väldigt smidigt och effektivt. Allra främst då vi nästan alltid har kodat samtidigt, fast på 2 olika datorer, och att man snabbt kunnat skicka mindre uppdateringar till varandra. Givetvis stötte på vissa mergekonflikter, mest nämnvärt är highscorelista vilket vi hade som en .txt fil som givetvis alltid skilde sig när vi satt och kodade och spelade parallellt. I eclipse så nyttjade vi även funktionen coverage när vi runnade och felsökte koden, allra främst för att se om if-satsen uppfyllde den menade funktionen.

Lärdomar

Vi lärde oss många nyttiga saker under projektets gång. För det första så gav gitlab oss en mer effektiv filhantering och arbetet blev bara allmänt enklare när vi implementerat det. En annan lärdom vi fick när vi programmerade hade med optimering att göra och spelets “snabbhet”. Under designen av menyerna märktes vi att om man tryckte på en knapp tog det kanske 0.2 sekunder innan man bytte “stage” vilket inte kändes särskilt smooth, och vi kom på när vi skrev koden att varje gång vi ville byta stage så skapade vi en ny stage av den vi vill byta till istället för att redan ha skapat den när vi initierade den ursprungliga “stage:n”. Då efter att ha skrivit om koden lite skapar vi alla “stages” när vi startar spelet vilket gör att om man trycker på en knapp så får man en omedelbar respons vilket gör spelet mer flytande.


Main
- GameModel model
- GameFrame frame
GameFrame
-GamePanel g
GameState
- GameModel model
- Image space
+ draw();
+ mouseCLlicked();
GameModel
-GameState currentstate
GamePanel
- GameModel model
ChooseLevel
- GameModel model
- Image space
+ draw();
+ mouseCLlicked();
MenuState
- GameModel model
- Image space
+ draw();
+ mouseCLlicked();
HighscoreState
- GameModel model
- Image space
+ draw();
+ mouseCLlicked();
PlayState
- Level1 onAction1
- Level2 onAction2
+ draw();
+ keyPressed();
NameChooser
- GameModel model
- Image space
+ draw();
+ mouseCLlicked();
HelpState
- GameModel model
- Image space
+ draw();
+ mouseCLlicked();
HighscoreTable
- GameModel model
- Image space1
Level1
- GameModel model
- Image space1
Level2
- GameModel model
- Image space2
OwnImage
- Image image
- double x
- double y
- double height
- double width
1
1..n