Projektbeskrivning

<Smrow> (Working title)

2015-02-12

Projektmedlemmar:
David Lindholm <davli921@student.liu.se>
Simon Karlsson <simka275@student.liu.se>

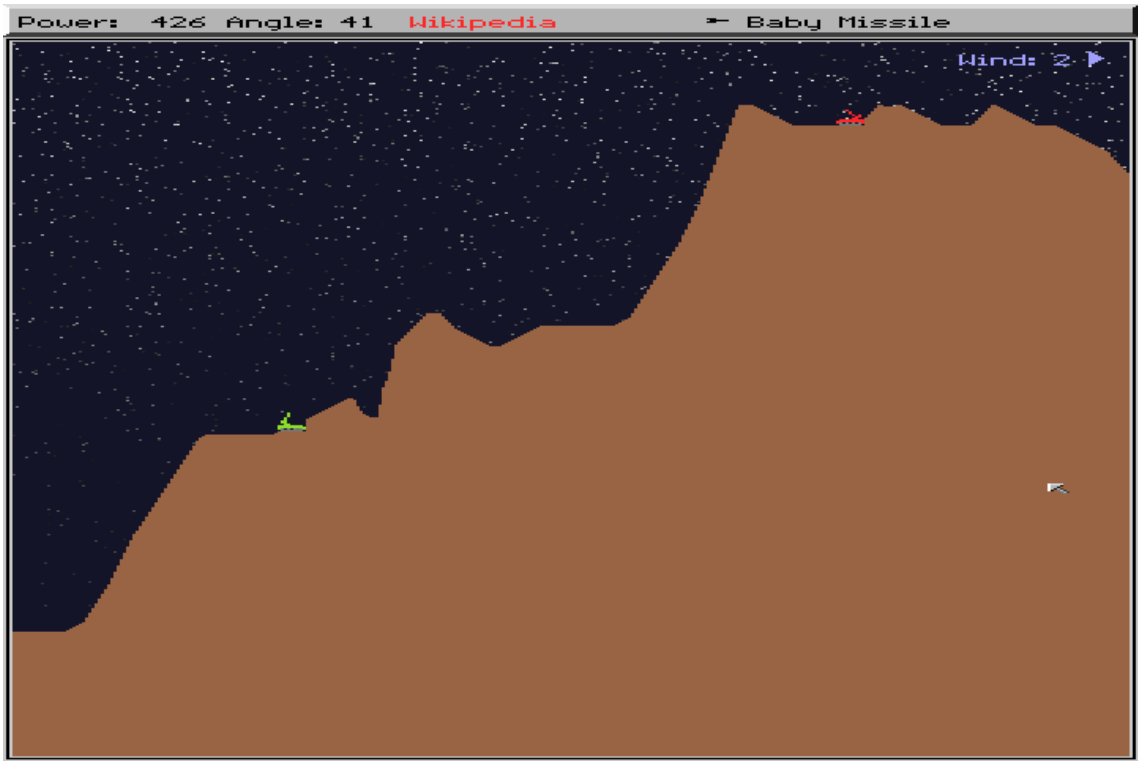
Handledare:
Mariusz <mariusz.wzorek@liu.se>

Table of Contents

Planering

1. Introduktion till projektet

Vi ska skapa ett ”turn-based artillery game” där två spelare möts på en bana med varierande landskap och slåss för att bli ensam kvar. Spelarna representeras av pansarvagnar på spelplanen och kan avfyra sina vapen för att ta kål på motståndaren. Vi har tagit inspiration från t.ex ”Worms” och ”Scorched Earth”



(Bild av Scorched Earth)

2. Ytterligare bakgrundsinformation

Spelarna har ett visst antal liv som reduceras när spelaren blivit träffad av en projektil.

När ens liv tar slut så har man förlorat spelet.

Under sin omgång så kan man röra sig och skjuta men man kan bara röra sig en bestämd distans och skjuta en gång.

3. Milstolpar

#	Beskrivning
1	Implementera en board-klass som ska representera landskapet. **fin**
2	Skapa en graphics-komponent som ritar ut board. (MVC) **fin**
3	Implementera spelar-klassen. Spelar objekten ska finnas i board. **fin**
4	Uppdatera graphics-komponenten så att den även ritar ut spelar-objekten. **fin**
5	Lägg till en vapen-klass. Vapen objekt tillhör spelaren. Vapen sätter hastigheten till en projektil. **fin**
6	Lägg till skjut funktion i vapen-klassen. **fin**
7	Implementera en projektil-klass. Projektil objekt ska finnas i board men skapas av skjut-funktionen. **fin**
8	Lägg till timer så graphics-kompenenten uppdateras. **fin**
9	Uppdatera board så den uppdaterar projektilens kordinater. **fin**
10	Uppdatera graphics-komponenten så att den även målar ut projektil objekt **fin**
11	Lägg till kollisions hantering i board så att en spelare kan kollidera med en projektil. **fin**
12	Gör så att spelarens liv reduceras vid kollision med projektil. **fin**
13	Lägg till vinst/förlust. **fin**
14	Implementera rörelse för spelar-objekten. **fin**
15	Ändra i vapen så att den kan skjuta i olika vinklar. **fin**
16	Lägg till kraft till vapnet så att projektilen får olika hastigheter. **fin**
17	Ändrar hur projektilen rör sig på board. (Sneda kast, fysik) **fin**
18	Lägg till turordning, så att två spelare kan spela mot varandra. **fin**
	(Nu finns ett fungerande spel)
19	Lägg till en startmeny (State pattern) **fin**
20	Lägg till en vinstmeny, ”scoreboard”. (State pattern) **fin**
21	Lägg till ljudeffekter, bakgrundsmusik?
22	Snygga till all kod

	(Börja implementera svårare saker)
23	Varierbart landskap med hinder.
24	Implementera nya projektiler.
25	Förstörbar miljö.
26	Spelaren utsätts för en rekyl när vapnet avfyras.
27	Nytt skadesystem som ger olika skada beroende på om spelaren blir träffad direkt eller om den blir träffad av kraftvågen från projektilen.
28	Spelaren kan hoppa framåt/bakåt.
29	Kraft på spelaren vid träff, dvs spelaren flyttas när den blir träffad eller står nära en projektil som träffar miljön.

3. Övriga implementationsförberedelser

Klasser:

GameBoard: Spelplans-klassen. Innehåller Player/Projectile -objekt

Player: Spelar-klassen, innehåller move(), vapen-objekt m.m.

Weapon *Abstract Class***:** Ska innehålla Shoot()

Olika **vapen-klasser** (t.ex MissileLauncher): Ärver av ”Weapon” . Innehåller Shoot() som skapar en projektil i board, vapen-klassen bestämmer hastigheten till projektilen.

Projectile: Projektil -objekt som skapas av ”Shoot()” i ett vapenobjekt

(Implementationsärvning, projektil är en superklass, det kommer senare finnas olika typer av projektiler som ärver fält/metoder men kan lägga till t.ex. explode() för en kluster projektil).

GameFrame: Innehåller Meny, BoardGraphics-komponenten

GameComponent: Graphics Component som målar ut ett board-objekt.

Sound: Klass som skapar ljud

Vi har tänkt att använda **MVC** (model-view-controller) där player och board skulle kunna vara Model, BoardGraphics är View och Controllers

Vi har också tänkt använda State pattern för olika states i spelet, t.ex. skulle ett state kunna vara meny, ett annat spelläge och highscore läge.

Slutinlämning

5. Implementationsbeskrivning

5.1. Milstolpar

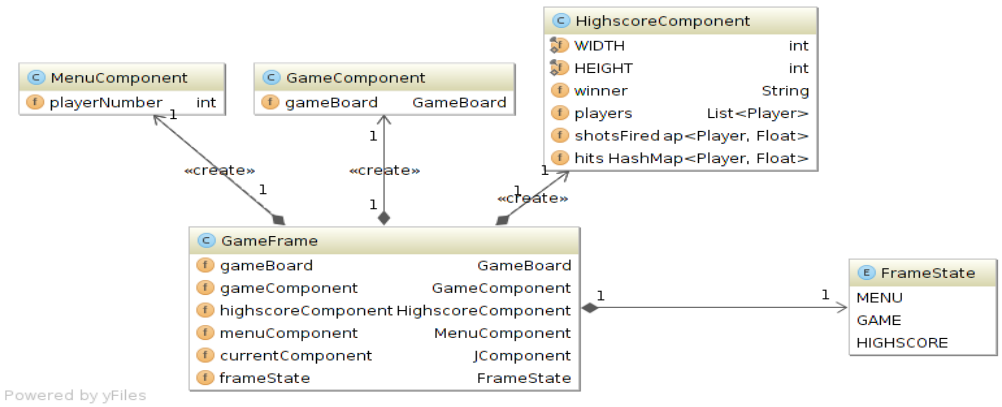
Se ovan, utförda milstolpar är markerade med **fin.**

5.2. Dokumentation för programkod, inklusive UML-diagram

Man startar spelet genom att köra main-klassen ”Smrow”. ”Smrow” skapar en timer som uppdaterar ”GameBoard” om ”GameFrame”:s state är i ”GAME”-state.

Spelet startas genom att ”StateList” skapar en ”GameFrame”, i ”GameFrame” skapas det en ”MenuComponent” en ”GameComponent” och en ”HighScoreComponent”.

Med hjälp av State pattern sätts nuvarande state till ”MENU” så att MenuComponent” visas. I menyn kan spelaren välja att starta spelet, det går att göra på två sätt, antingen startar man spelet genom att klicka på ”Start”-knappen eller så väljer man spelarnamn på spelarna genom att skriva in namnen i textrutan och trycka på ”Enter”-knappen i menyn.



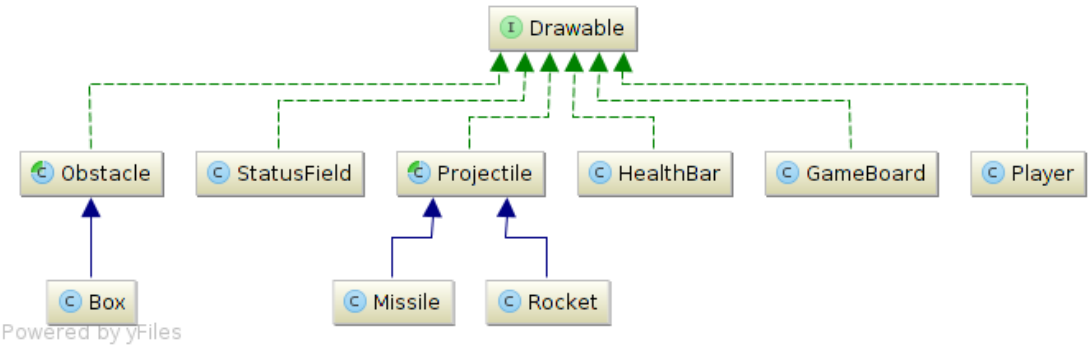
UML 1 GameFrame

När spelet har startats byter ”GameFrame” state till ”Game” och ”GameComponent” visas. ”GameFrame” skapade en ”GameBoard” som skickades till ”GameComponent”.

I ”GameBoard”, skapas spelarna ”Player” och vilka hinder som ska finnas på banan med ”addBox”-metoden. Det sätts även en del startvärden för spelet, t.ex. vilken spelare som börjar spela och det skapas ett ”StatusField” där information visas om spelarna.

I ”GameComponent” finns alla ”actions” som används i spelet för att styra spelarna, skjuta vapnen och styra vapnen. ”GameComponent” har också hand om utritningen till skärmen genom att metoden ”paintComponent” kallar på metoden ”draw” i ”GameBoard”.

Metoden ”draw” i ”GameBoard” målar ut spelplanen i ”GameBoard” och kallar andra objekts ”draw”-metoder. Dessa objekt inklusive ”GameBoard” implementerar gränssnittet ”Drawable” vilket garanterar att de kan måla ut sig själva. Se UML diagram nedan.



UML 2 Drawable

”GameBoard” representerar spelet och har all kollisionshantering, metoder för hur spelarna får röra sig, hur man kan skjuta och alla objekts positioner.

Spelarna rör sig genom att ”GameComponent” registerar ett ”Key event” som med hjälp av dess ”actions” kallar på metoden ”moveCurrentPlayer” i ”GameBoard”, ”moveCurrentPlayer” tar in en ”direction” som beror på vilket ”key event” som registrerades.

Det sker på liknande sätt när man ska skjuta, när man trycker ned mellanslag kallas ”setIsChargingWeapon” i ”GameBoard” som laddar upp vapnets kraft, när man släpper upp mellanslag avfyras projektilen, då kallas metoden ”shotsFired” i ”GameBoard”. ”shotsFired” skapar en projektil med vapnets ”shoot”-metod. Projektilens position uppdateras från ”tick” metoden i ”GameBoard” som kallar på ”moveProjectile”.

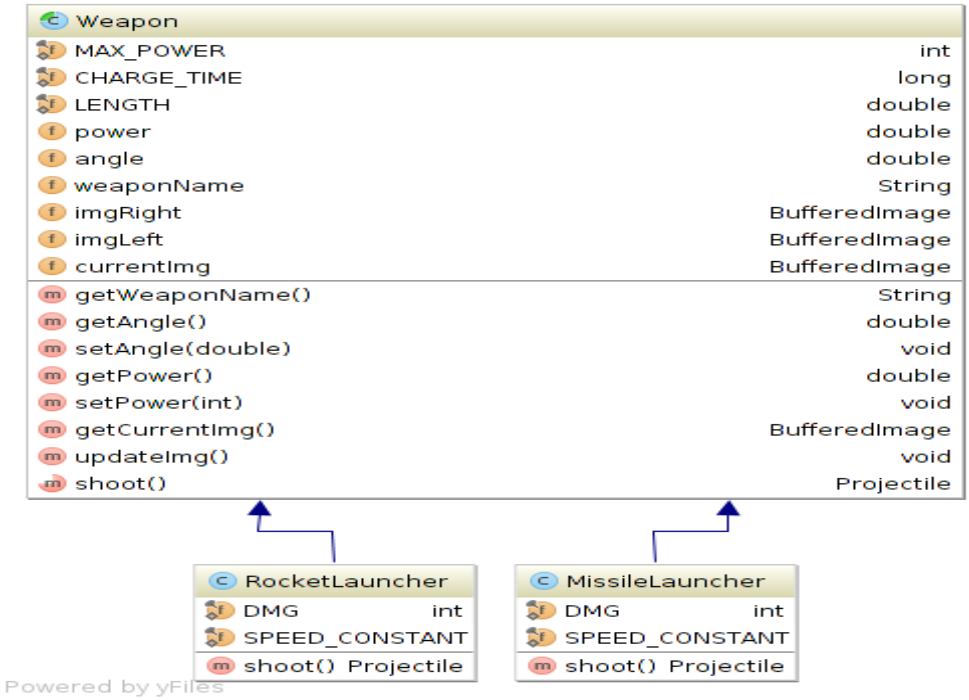
När något händer i ”GameBoard” så notifieras lyssnaren ”GameComponent” så att den grafiska representationen uppdateras.

I ”moveProjectile” kollas kollisioner med spelplanen och spelarna. När en kollision har skett tas projektilen bort med ”resetProjectile” och det är nu nästa spelares tur. Om det har skett en kollision med en spelare så förlorar spelaren ett antal liv beroende på vilken projektil som träffade spelaren. När en spelare har förlorat allt sitt liv så är spelet över och den spelare med mest liv kvar vinner spelet. Då kommer ”GameFrame”:s state bytas till ”HIGHSCORE” och information om spelomgången visas.

En spelare representeras med ett ”Player” objekt. Varje spelare har en unik position, samt ett bestämt antal liv som representeras grafiskt i ”healthBar” och ett antal vapen, som man kan byta mellan med ”changeWeapon”. I ”Player” finns metoder för att uppdatera spelarens position, spelarens liv, riktning (Direction) och vinkel (angle). Där finns också metoder för att måla ut spelaren och spelarens vapen.

De olika vapnen utökar den abstrakta klassen ”Weapon” som innehåller generella metoder och fält som varje vapen har. I underklasserna implementeras en unik skjut-metod ”shoot” som skapar en projektil. Alla vapen representeras grafiskt men implementerar inte

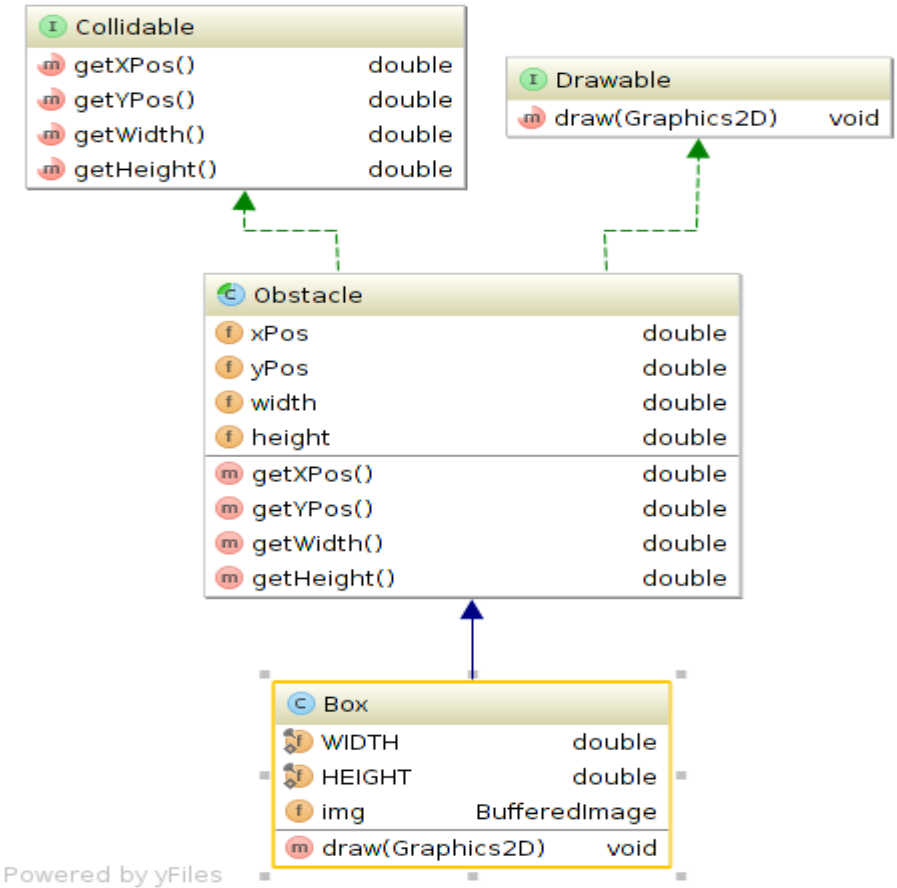
”Drawable” för att vi anser att de tillhör ”Player” och målas ut av ett ”Player” objekt, mer om detta under desisnbeslut.



UML 3 Weapon

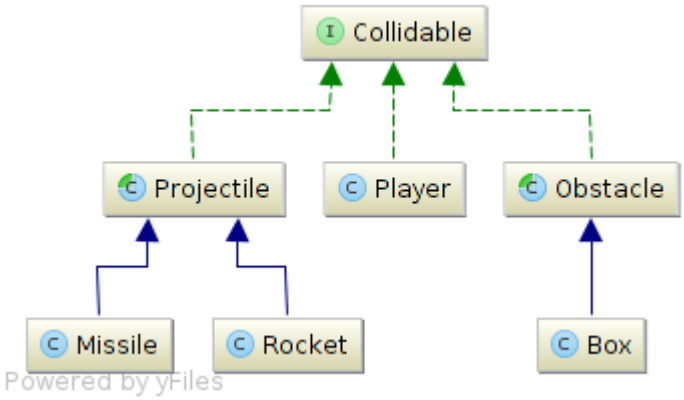
Strukturen på projektilerna ser liknande ut som för vapnen, det finns en abstrakt klass ”Projectile” som sedan utökas i underklasser.

Hinder på banan representeras av underklasser av den abstrakta klassen ”Obstacle” som utökas i ”Box”. ”Obstacle” uppfyller gränssnittet ”Collidable” och ”Drawable”.



UML 4 Obstacle

”Collidable” ser till så att klasserna som uppfyller gränssnittet har position och mått som behövs för att detektera kollisioner.



UML 5 Collidable

Vi använder oss av tre enumklasser, ”WeaponType”, ”Direction” och ”FrameState”. Mer om dessa under 5.6.

5.3. Användning av fritt material

Vi har använt ”MigLayout” till swing.

5.4. Användning av objektorientering

1. **Gränssnitten ”Drawable” och ”Collidable”:**

   ”Drawable” garanterar att alla ”Drawable” typer har en metod (”draw”) med vilken objektet kan rita ut sig själv. Detta implementeras till exempel av ”Player” som ska kunna representeras grafiskt på spelplanen.

   ”Collidable” är typer som har metoder som returnerar mått och position som gör det möjligt att räkna ut vare sig två ”Collidable”-objekt kommer eller har kolliderat. ”Collidable” implementeras till exempel av ”Projectile” och ”Obstacle”. ”Collidable” används av metoder som till exempel ”checkCollision” i ”GameBoard” som tar in två ”Collidable” objekt och kollar om det sker en kollision mellan objekten.

   Utan objektorientering hade lösningen sett liknande ut i de specifika klasserna men vi hade inte kunnat garantera att ett objekt uppfyller kraven som ställs av ”Collidable” eller ”Drawable”. Detta hade lett till att vi inte hade kunnat ställa lika specifika krav på inparametrarna i t.ex. ”checkCollision”, vilket hade lett till att man hade kunnat kolla kollision med något som inte ska kunna kollidera.

2. **Abstrakta klasser så som ”Weapon” och ”Projectile”:**

   Ett ”Weapon” ska dels ha och kunna ändra kraft (”power”) och riktning (”angle”), sedan ska också varje ”Weapon” ha en unik version av metoden ”shoot” som skapar en passande projektil. Den abstrakta klassen utökas i ”MissileLauncher” och ”RocketLauncher”.

   Den abstrakta klassen ”Projectile” ser till så att alla underklasser har ett antal fält och metoder, till exempel ”getDmg” och ”move”. Den utökas i ”Missile” och ”Rocket”.

   Vi valde att skapa abstrakta klasser för att enkelt kunna implementera flera vapen/projektiler. Genom att ha den abstrakta klassen behöver vi inte implementera alla metoder igen i den nya underklassen och det garanterar att alla underklasser implementerar samma gränssnitt som den abstrakta klassen gör.

   Utan objektorientering hade vi behövt skriva om alla fält och metoder för varje ny typ av vapen eller projektil.

3. **Konstruktorer:**

   Varje föremål i spelet instantieras med en konstruktor.

Det används till exempel i ”GameBoard” där spelar-objekten och hinder skapas i konstruktorn och startvärden för spelet sätts till det nya ”GameBoard” objektet.

Utan objektorientering hade vi blivit tvungna att skapa en stor fil med variabler för alla spelares information, som position och storlek o.s.v., istället för att ha enstaka objekt som representerar spelare och projektiler m.m.

4. **Sen/dynamisk bindning:**
   Detta används till exempel av metoden ”shotsFired” i ”GameBoard” som kallar på metoden ”shoot” i ett vapen. Beroende på vilken vapentyp som kallas på så utförs metoden annorlunda. Om ”MissileLaunchers” ”shoot” metod kallas på så skapas det en ”Missile” och vissa värden sätts till den, motsvarande sker för ”RocketLauncher” som skapar en ”Rocket”.

   Utan objektorientering hade vi varit tvungna att veta vilken funktion vi skulle använda.

5.5. Motiverade designbeslut med alternativ

1. Först skrev vi koden utan ”Drawable” och det resulterade i att ”GameComponent” blev en väldigt stor klass. Därför valde vi att förenkla den genom att göra vissa klasser ”Drawable”. Att en klass är ”Drawable” innebär att den kan rita ut sig själv. Vi tyckte också att det var en fördel att man kan se direkt i klassen hur den representeras grafiskt.
2. Även fast ”Weapon” ska representeras grafiskt så implementerar den inte ”Drawable”. Detta på grund av att vi anser att vapnet tillhör spelaren och därför har vi valt att rita ut vapnet därifrån. Om ”Weapon” skulle implementera ”Drawable” så skulle ”draw” metoden ha varit tvungen att ta in en ”Player” som inparameter för att kunna rita ut vapnet på rätt plats.

   Detta hade lett till att de andra klasserna som implementerar ”Drawable” hade behövt ha en överflödig parameter, vilket vi helst ville undvika.
3. Vi har valt att lagra spelar-objekten i en lista ”players” i ”GameBoard” och vi har definerat alla metoder utifrån det. Detta leder till att det enkelt går att lägga till fler spelare till spelet.

   Ett exempel på en metod som påverkats av beslutet är ”moveProjectile” i ”GameBoard” där den kollar efter kollision med hjälp av ”willCollide” med ”players”-listan.

   En annan lösning hade varit att ha ett bestämt antal spelare och definiera alla metoder utifrån det. Vi valde att implementera den förstnämnda lösningen efter förslag från labassistenten. Vi tyckte att det var bra att ha med möjligheten att lätt kunna utöka spelet.

4. Vi valde att implementera gränssnitttet ”Collidable” för att lätt få en översikt över vilka objekt som ska kunna hantera kollisioner och för att kollisionsmetoder i ”GameBoard” ska kunna hantera rätt typ av objekt, till exempel projektiler och spelare.

   Spelet hade kunnat fungera utan ”Collidable” men det hade gett sämre översikt och det hade kunnat bli fel i kollisionshanteringen.

5. Vi har valt att ha med en Singleton ”StateList” som gör att vi enkelt har tillgång till en frame. Den används när man ska byta ”frameState” som i sin tur byter komponent. Detta sker till exempel när spelet är slut och ”GameBoard” byter ”GameFrame”:s ”frameState” till ”HIGHSCORE” och ”HighScoreComponent” visas.

   En annan lösning hade varit att ”MenuComponent” och ”GameBoard” som byter ”GameState” hade haft ett ”GameFrame”-fält för att få tillgång till det aktuella ”GameFrame” objektet. Vi ville undvika det för att inte få loopar av beroende mellan klasser. Till exempel hade ”MenuComponent” behövt en ”GameFrame” samtidigt som ”GameFrame” har ”MenuComponent”.

6. Vi valde att använda enum-typer för till exempel riktningar (”Directions”). För att metoder som till exempel ”moveCurrentPlayer” som tar in en riktning bara kan få ett värde från enum-typen.

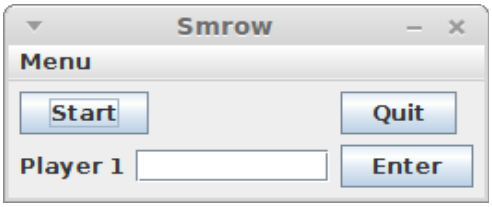
   Tidigare hade vi strängar som inparameter till liknande metoder men det ville vi undvika för att det är enklare att göra fel då.

5.6. Bredd inom Java och objektorientering

1. Enum-typer: ”FrameState”, ”Direction”, ”WeaponType”.
2. GUI-komponent: ”GameComponent”
3. Tangentbordsstyrning: Används i ”GameComponent”, t.ex. i ”actions”: ”changeWeapon” och ”shoot” som sedan används i metoderna ”shotsfired” i ”gameBoard” och ”changeWeapon” i player.
4. Grafiskt gränssnitt
5. Designmönster: Singleton.
6. Designmönster: State pattern, används i ”GameFrame”.

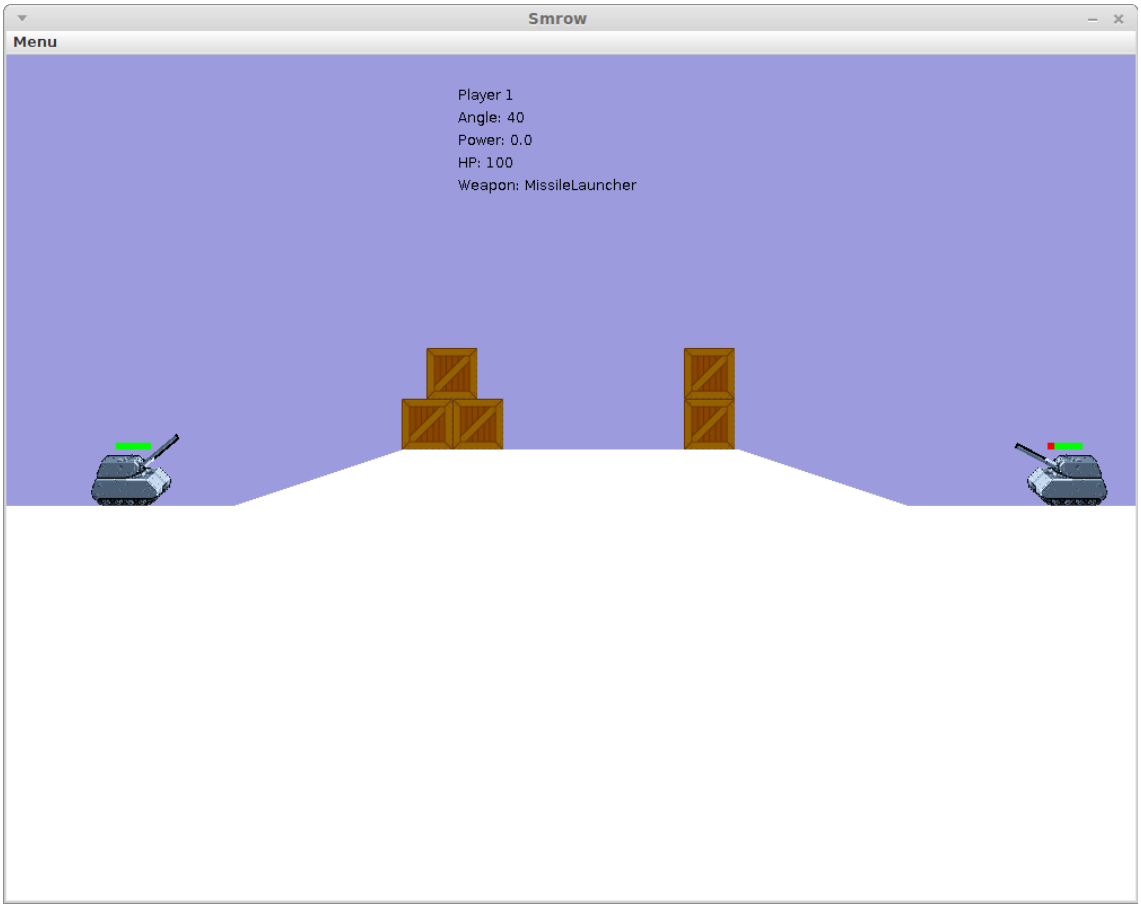
6. Användarmanual

Spelet startas genom att man kör ”Smrow” klassen. Då kommer man in i menyn, där kan man starta spelet genom att trycka på ”start” eller genom att skriva in spelarnamn och trycka på ”enter”. Se bild 1, Meny.



1. Meny

Efter detta steg så kommer man in i själva spelet, se bild 2, Game.



2. Game

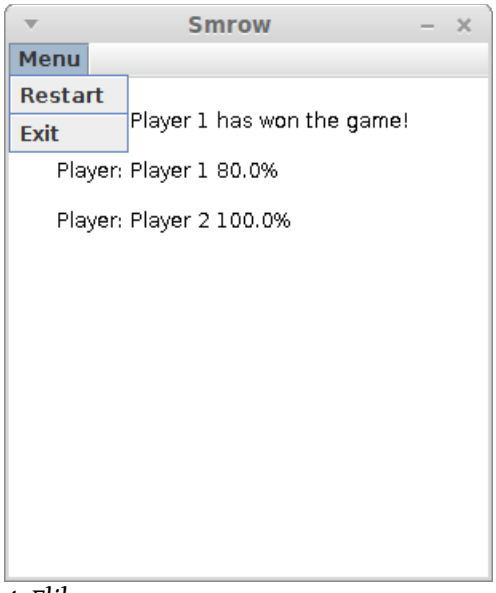
Spelare 1 börjar spela, man styr sin spelare med höger- och vänster-piltangent och siktar vapnet med upp- och ned-piltangent. För att ladda vapnet håller man inne mellanslag och för att skjuta släpper man upp mellanslag, när vapnet laddas kan man se vilken kraft man har under ”power”-fältet i informationsfältet i mitten av skärmen. Där finns också information

om vems tur det är, vilken vinkel man har på vapnet, hur mycket liv man har kvar och vilket vapen som används.
För att byta vapen trycker man på ”1” eller ”2”. När man har avfyrat sitt vapen och projektilen har kolliderat med något objekt eller hamnat utanför spelplanen så avslutas turen och det är nästa spelares tur.

När ens spelares liv når 0 så är spelet slut, man kommer då in i highscore och kan se information om spelomgången. Se bild 3, highscore.

3. Highscore

För att starta om spelet kan man när som helst under spelets gång gå in i menyn uppe till vänster och välja ”Restart”, där kan man också välja ”Exit” för att avsluta spelet. Se bild 4, flikmeny.



4. Flikmeny

7. Slutgiltiga betygsambitioner

Vi siktar på betyg 4. Vi har inte uppfyllt de kvantitativa kraven för 4 på ”breda praktiska kunskaper inom programmering”, men på ”förestående för objektorientering” och ”UML-diagram” har vi uppfyllt kraven för betyg 5.

8. Utvärdering och erfarenheter

Detta avsnitt är en väldigt viktig del av projektspecifikationen. Här ska ni tänka tillbaka och utvärdera projektet (något som man alltid bör göra efter ett projekt). Som en hjälp på vägen kan ni utgå från följande frågeställningar (som ni gärna får lämna kvar i texten så att vi lättare kan sammanställa information om specifika frågor):

- *Vad gick bra? Mindre bra?*
- *Lade ni ned för mycket/lite tid?*
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
- *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.*
- *Vad har ni lärt er så här långt som kan vara bra att ta med till era egna kommande kurser/projekt?*
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
- *Har ni saknat något i kursen som hade underlättat projektet?*

Vi använder detta för att utveckla och förbättra kursen till nästa år. Vissa delar som är användbara som tips till andra studenter kan komma att citeras (givetvis helt anonymt!) under föreläsningar eller på websidor.

(Glöm inte att exportera till PDF-format innan ni skickar in!)

Vi har följt arbetsmetodiken under projektets gång, det vi skulle kunna förbättra tills nästa gång är att dokumentera mer under projektets gång, något som vi till viss del har gjort men det skulle ha kunnat gjorts i större utsträckning.

Vi tog till oss tipset att felsöka ofta under projektets gång vilket vi tycker att vi har haft nytta av, vi har kört igenom kodinspektionen och testat efter buggar när vi har lagt till något nytt till projektet. Detta har lett till att vi i slutfasen inte har haft några större problem i spelet.

Vi tror också att det har hjälpt oss att komma vidare i projektet och vi känner inte att vi har suttit fast vid någon speciell punkt särskilt mycket.

När vi började projektet hade vi inte tillräckligt bra koll på de kvantitativa kraven så vi fick inte med allt som var tänkt. Vi kollade på kraven och försökte sätta upp milstolpar därefter men det är svårt att följa planeringen exakt.

Vi tycker att vi har lagt ned tillräckligt med tid, vi har jobbat mycket utanför schemalagd tid.