Projektbeskrivning

Pax-con

2018-11-04

Projektmedlemmar:

Nils Broman nilbr692@student.liu.se

Handledare:

Mariusz Wzorek mariusz.wzorek@liu.se

Innehåll

1 Introduktion till projektet

Det spelet jag ska utveckla är inspirerat av spelet Pax-con (https://www.y8.com/games/pac_xon). Det är ett single-player spel som har PacMan som huvudkaraktär. Målet med spelet är att fylla de tomma kvadraterna i spelet utan att kollidera med fienderna. När PacMan går in i det tomma området lämnar han efter sig ett spår. Det är sedan med detta spår som PacMan omringar kvadrater som sedan markeras klara. Klarar man att fylla en viss procentsats av spelplanen så går man vidare till nästa nivå där fienderna är svårare.

Bild 1. Målbild av spelet Pac-xon.

2 Ytterligare Bakgrundsinformation

För att få en bild av spelet är det simplast att helt enkelt spela det, men de viktigaste reglerna presenteras här. Kollision med fiende gör att PacMan förlorar ett liv. PacMan startar med 3 liv och spelet avslutas när dessa är förbrukade. För att komma vidare till nästa nivå måste 80% av spelplanen täckas av blåa kvadrater, vilket de görs genom att PacMan omringar dem.

3 Milstolpar

#	Beskrivning
1	Spelplanen implementerad bestående av kvadrater av olika typer. Funktionalitet för att skapa en tom spelplan med en ”ram” runt.
2	Klass för att representera spelaren skapad och keyboard-input för att styra implementerat.

3	Ritandet av spåret efter spelaren implementerat.
4	Fiender slumpas ut på spelplanen med starthastigheter.
5	Fienderna studsar på väggarna.
6	GUI för start-sida implementerat.
7	Kollisionshantering för fienderna och huvudkaraktären implementerad.
8	Kvadraterna efter spelarens spår markeras som klara när spelaren går tillbaka till en kant.
9	Kollisionshantering mellan fienderna och de ”färdiga” kvadraterna klar.
10	Mus-klick implementerat för start-skärmen
11	Fiender som kolliderar med spelarens spår markerar spåret rött och de röda jagar mot spelaren.
12	Av spelaren omringade områden ska fyllas med ”färdiga” kvadrater.
13	Poäng implementeras.
14	Spelaren förlorar ett liv om den åker in i sitt eget spår.
15	End-screen och alternativ att spela igen implementeras.
16	Flera nivåer.
17	Ytterligare en typ av fiende som kan förstöra de ”färdiga” kvadraterna.
18	Keyboard-input för att starta spelet
19	Highscore funktioner.
20	Ljud till olika händelser i spelet.
21	Power ups implementeras.
22	Instängda fiender tas bort och läggs tillbaka i spelet på ”öppen” yta.
23	Kollisionshantering av power ups.
24	Level-screen för att se vilka nivåer som är upplåsta.
25	Settings-skärm implementeras.

26	Ytterligare typer av fiender samt power ups implementeras.
26	Möjlighet till att pausa spelet samt klicka på nivå för att starta vald nivå.
27	Möjlighet till att höja/sänka ljudnivån i settings-skärmen.
28	Möjlighet till att ändra kontroller i settings-skärmen
29	Möjlighet till att välja två spelar läge med valda kontroller.

4. Övriga implementationsförberedelser

Jag har tänkt att skapa klassen Board för att representera spelplanen. Klasser för vart och ett av alla spelobjekt samt en superklass för spelobjektens gemensamma egenskaper och beteenden. Dessa kanske kommer delas i mindre subgrupper som delar en superklass, till exempel en superklass ”PowerUp” med subklasser som är olika typer av power ups. Utöver det kommer jag ha en klass för att rita ut det grafiska i spelet, klasser för hanteringen av keyboardinput och musklick samt enum klasser för att bland annat kunna skilja fienderna åt.

5. Utveckling och samarbete

Eftersom jag jobbar själv så tänker jag inte dela upp någon del av koden. Tanken är att jobba på förmiddagar för att jobba så effektivt som möjligt. Jag kommer även jobba med projektet på helger för att inte ta alltför mycket tid från andra kurser. Jag har som betygsambition att få betyg 5.

6 Implementationsbeskrivning

6.1 Milstolpar

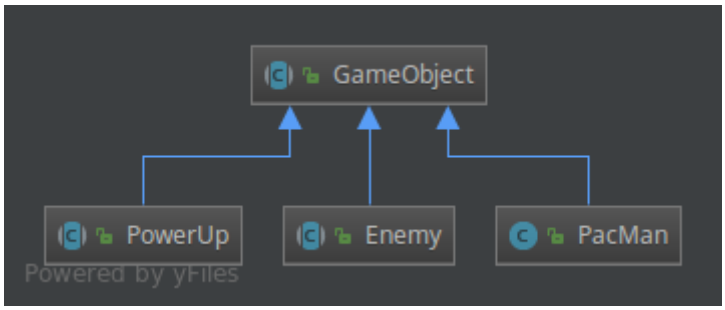
Alla milstolpar upp till och med milstolpe 24 är implementerade.

6.2 Dokumentation för programkod, inklusive UML-diagram

Övergripande programstruktur

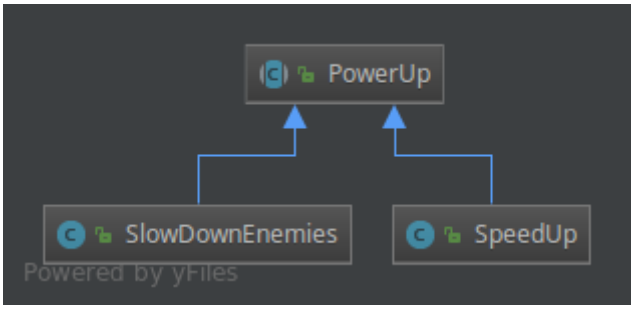
Då spelet startas skapas ett spelbräde i klassen Board. Därefter skapas ett fönster för spelet i klassen GameFrame. Sedan anropas en metod som startar spel-timern i klassen Board. Spelet styrs av just den timern som tickar kontinuerligt, och kallar på metoden tick i Board som hanterar spelet i just den stunden och gör eventuella förflyttningar och ändringar av objektens koordinater och hastighet. Varje spelobjekt har också en tick-metod som kallas på varje gång timern tickar. Till exempel flyttas fienderna i deras tick-metod. Därefter kontrolleras kollisioner för varje spelobjekt i deras respektive kollisionshanteringsklass. Sedan fortsätter timern att ticka så att spelet fortgår. Skärmen uppdateras genom att Board-klassen kallar på metoden boardChanged, som alla klasser som implementerar interfacet BoardListener måste ha, och därefter ritar klassen GameComponent ut spelet i nästa uppdaterade stund.

Spelobjekt



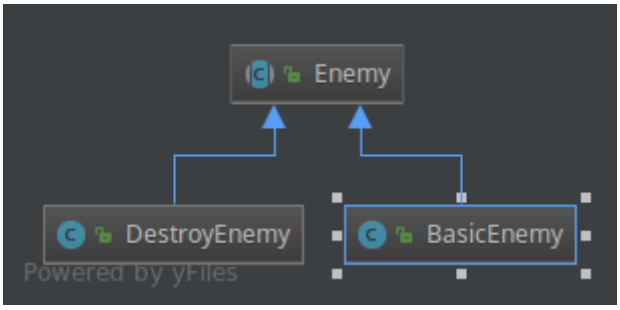
UML-Diagram 1. Beskrivning av vilka klasser som står i direkt relation till GameObject.

En viktig klass i spelet är den abstrakta superklassen GameObject som ärvs av klasserna PacMan, PowerUp samt Enemy som beskrivet i UML-Diagram 1. GameObject-klassen innehåller fält och metoder, både abstrakta och icke abstrakta, som samtliga spelobjekt använder sig av. Till exempel innehåller GameObject fälten x- och y-koordinater. De tre subklasserna PowerUp, Enemy och PacMan ärver dessa fält från GameObject eftersom de har dessa egenskaper gemensamt, medan varje individuell subklass implementerar olika hantering av fälten. Förutom att göra det tydligt och undvika redundans i koden ger det även fördelen att man kan kalla på metoder i GameObject utan att veta vilken specifik subklass ett objekt är av, eftersom alla spelobjekt i spelet ärver dessa metoder. Detta tycker jag är en särskilt bra lösning just eftersom den gör det otroligt tydligt var all funktionalitet ska placeras och bidrar till läsbarhet.



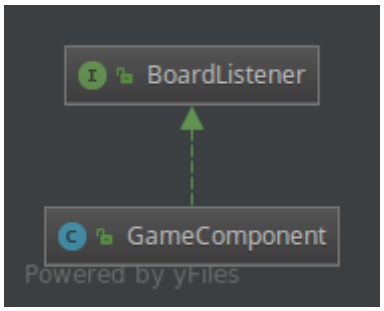
UML-Diagram 2. Beskrivning på vilka klasser som ärver av klassen PowerUp.

Utifrån UML-Diagram 2 så ser vi att de båda klasserna SlowDownEnemies och SpeedUp ärver från den abstrakta klassen PowerUp. Eftersom båda dessa klasser är just power ups har de delad funktionalitet, som till exempel att alla power ups ska försvinna då de kolliderar med en fiende. Genom att ha denna abstrakta klass för power ups:en blir det mycket lättare att utöka spelet och implementera fler power ups utöver de som redan finns. Precis som tidigare nämnt angående superklassen GameObject gör även denna implementation att redundans undviks, koden blir tydlig och polymorfism kan utnyttjas.



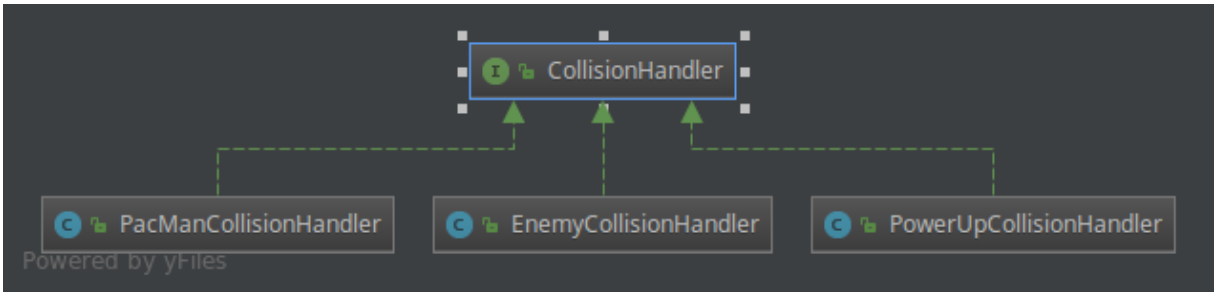
UML-Diagram 3. Beskrivning på vilka klasser som ärver av klassen Enemy.

På liknande sätt som tidigare diagram så visar Diagram 3 att subklasserna DestroyEnemy och BasicEnemy ärver från den abstrakta klassen Enemy. Subklasserna har till exempel rörelsebeteendet gemensamt, men reagerar olika på kollision med vägg. Denna struktur har samma fördelar som beskrivits ovan för power ups.



UML-Diagram 4. Beskrivning av grafikklassens implementering av BoardListener.

I klassen BoardListener finns metoden boardChanged som kallas varje gång spelplanen har uppdaterats. I klassen GameComponent ritas alla spelobjekt ut på spelplanen och därför är det viktigt att den klassen vet precis när spelbrädet har uppdaterats. Därför implementerar klassen BoardListener och också metoden boardChanged. Varje gång boardChanged kallas på så ritar GameComponent ut en ny, uppdaterad version av spelet på spelplanen.



UML-Diagram 5. Beskrivning av kollisionshanteringens relationer.

Kollisionshanteringen hanteras separat i olika klasser beroende på vilken typ av objekt som kolliderar. PacMans kollisionshanterare detekterar och hanterar kollision med fiender samt kollision med olika typer av kvadrater på spelplanen. Fiendernas kollisionshanterare detekterar och hanterar kollision med väggarna på spelplanen. Kollisionshanteraren för power ups detekterar och hanterar kollision med fiender samt PacMan.

Genom att samla dessa tre klasser med ett interface, CollisionHandler, och låta alla tre implementera den ärvda metoden checkCollisions, blir det smidigt att anropa metoden för varje spelobjekt. För varje tick som sker loopas listan av spelobjekt igenom och checkCollisions anropas. Till exempel använder sig båda fiendeklasserna (BasicEnemy, DestroyEnemy) av EnemyCollisionHandler medan power up-klasserna (SpeedUp, SlowDownEnemies) använder sig av PowerUpCollisionHandler. Specifika reaktioner på kollisionen implementeras i metoder i objektets klass som anropas ifrån kollisionshanteraren. Till exempel innehåller klassen DestroyEnemy själv funktionaliteten att förstöra en kvadrat på spelbrädet. Detta ger en tydlig och struktur i koden, och det blir smidigare att utöka spelet och implementera nya spelobjekt med andra typer av kollisioner.

6.3 Användning av fritt material

I detta projektet använder jag mig av de externa klasserna sun.audio.AudioPlayer samt sun.audio.AudioStream. Detta för att kunna spela upp ljudfiler i spelet.

6.4 Användning av objektorientering

1. **Egenskap**
Abstrakta klasser.
Användning
Jag har implementerat klassen Enemy som är en abstrakt klass och klasserna BasicEnemy och DestroyEnemy är subklasser till den. Detta gör att gemensamma metoder och variabler kan delas i klassen Enemy, medan skild funktionalitet finns i subklasserna. Även klasserna GameObject respektive PowerUp är abstrakta och innehåller delade variabler och metoder för samtliga spelobjekt och olika power ups. Alla dessa abstrakta klasser minimerar redundans i koden, gör strukturen tydligare, gör det enkelt att implementera nya klasser och ny funktionalitet samt ger möjligheten att utnyttja polymorfism.
Lösning utan objektorientering
En Lösning utan objektorientering hade varit att helt enkelt inte ha dessa abstrakta klasser, utan att bara implementera klasserna för sig.
Skillnader mellan lösningarna
Utan abstrakta klasser hade koden kunnat bli väldigt repetitiv, mer svårläst samt gjort det svårare att hantera de olika typerna i andra metoder och klasser. Det hade inte heller fungerat att ha den gemensamma listan objects i GameObjectHandler för alla spelobjekt, eftersom den deklareras som en lista som innehåller objekt av typen GameObject, och inte hade kunnat deklareras innehålla instanser av olika typer av klasser.

2. **Egenskap**
Interface
Användning
Jag använder mig av två interface, BoardListener och CollisionHandler. BoardListener deklarerar metoden boardChanged, vilken implementeras av klassen GameComponent. boardChanged anropas varje gång en förändring av spelbrädet sker, alltså meddelas alla instanser av BoardListener. I en utökning av spelet skulle fler klasser kunna behöva ”lyssna” på spelbrädet, och i så fall implementera BoardListener och därmed boardChanged. CollisionHandler deklarerar metoden checkCollisions, som implementeras av de separata kollisionshanterarna. På så sätt kan varje spelobjekts kollisionshanterares implementation av metoden anropas via interfacet, till exempel i GameObjectHandler där det inte är känt exakt vilken typ av kollisionhanterare som ska anropas eftersom det inte är känt exakt vilket typ av spelobjekt som hanteras.
Lösning utan objektorientering
Istället för att använda BoardListener när något i spelet uppdateras hade man direkt kunnat kalla på GameComponent.repaint. För att hantera kollision hade man kunnat kontrollera vilken typ objektet är av innan man kallar på dess kollisionshantering och på så sätt behöver man inte något interface.
Skillnader mellan lösningarna
Utan interface hade kollisionshanteringen inte kunnat skett lika smidigt. Objektorienterade lösningen gör även att klassen som ska kalla på kollisionshanteringen inte behöver veta vilket typ av objekt det är. På samma sätt som att Board inte behöver ha koll på vad metoden boardchanged gör, den behöver bara kalla på metoden så sköts resten i GameComponent. Interface gör koden mer läsbar.

3. **Egenskap**
Implementationsärvning,
Användning
Jag använder mig av implementationsärvning till exempel i klasserna BasicEnemy samt DestroyEnemy då båda dessa klasser ärver implementationen av metoden tick från den abstrakta klassen Enemy. Det gör så att implementationen av tickmetoden inte behöver finnas i både

BasicEnemy och DestroyEnemy utan den behöver bara finnas i Enemy.
Lösning utan objektorientering
Utan objektorientering hade jag kunnat implementera en tickmetod i båda Enemy-klasserna.
Skillnader mellan lösningarna
Fördelen med den objektorienterade lösningen är att det blir mer strukturerad kod och mindre upprepad kod.

4. **Egenskap**
Subtypspolymorfism

Användning
Jag använder mig av subtypspolymorfism där jag anropar interface-metoden checkCollision i klassen GameObjectHandler. På raden där jag anropar tempObj.getCollidionHandler().checkCollision() utnyttjas just subtypspolymorfism. Först hämtas klassens kollisionshanterare och därefter anropas metoden checkCollision i den kollisionshanteraren. I det anropet vet vi inte vilken subklasstyp objektet vi hanterar är av och därmed inte vilken subklasstyp dess kollisionshanterare är av, men samtliga objekt har en kollisionshanterare och samtliga kollisionshanterare implementerar metoden checkCollision från den abstrakta klassen CollisionHandler. Därför kan jag kan anropa den metoden utan att veta vilket typ av objekt det är.
Lösning utan objektorientering
Detta kunde ha lösts utan objektorientering genom att till exempel antingen först jämföra vilken typ av objekt det är med hjälp av if-satser och sedan anropa rätt kollisionshantering.
Skillnader mellan lösningarna
Skillnaden hade då blivit att strukturen hade blivit mycket otydligare. Hade det funnits hundra olika typer av objekt hade if-satser gjort koden väldigt lång.

5. **Egenskap**
Konstruktorer
Användning
Jag har använt konstruktorer i samtliga klasser av vilka ett objekt ska kunna skapas. För att skapa ett nytt objekt anropas helt enkelt klassens konstruktor. Abstrakta klassers konstruktorer anropas endast inifrån konstruktorer i klasser som implementerar den abstrakta klassen.
Lösning utan objektorientering
Utan objektorientering hade inte koden byggt på att hantera objekt som skapas via konstruktorer, utan data för alla objekt hade behövts hanteras i andra datastrukturer. Till exempel hade ett objekt kunnat motsvaras av en lista med egenskaper, och konstruktorn hade kunnat motsvaras av en funktion som sätter värdena i listan.
Skillnader mellan lösningarna
Att representera objekt med en lista som beskrivits ovan skulle göra koden svårhanterlig och rörig. Objekt och konstruktorer gör att mycket kod går att återanvända och hanteras på ett smidigt sätt. Det blir också lätt att ändra egenskaperna hos ett objekt jämfört med en lista.

6.5 Motiverade designbeslut med alternativ

1. **Vad jag vill åstadkomma**
Underlätta indexeringen för platser på spelplanen och lätt bygga upp spelplanen med de olika rut-typerna.
Hur
Genom att först skapa detta rutnät vars koordinater anges som (i:te raden, j:te kolumnen) och sedan anpassa UI:t efter det rutnätet. Från Game-klassen skapas ett rutnät genom göra en instans av klassen Board, vilket är klassen som skapar rutnätet. I GameComponent där all grafik ritas upp skalas

fönstret utifrån rutnätet.

Alternativ lösning

Istället för denna lösning så hade jag kunnat skippa rutnätet och enbart hittat rutors positioner genom pixlar på spelplanen. Det hade gjort det svårare att rita upp spelplanen korrekt.

Bättre/sämre

Jag anser att det blev bättre med ett rutnät med koordinater som inte angavs i pixlar då det blev lättare att hantera i resten av koden. PacMan rör sig i linje med dessa koordinater. Det var nödvändigt för att PacMan skulle kunna omringa delar av planen, vilket spelet går ut på.

2. **Vad vill jag åstadkomma**

Spåret efter PacMan skall vara visuellt tillfredsställande men PacMan ska dö då denna kolliderar med spåret.

Hur

Jag implementerade två rutor i spåret som var ”säkra”, alltså som PacMan tilläts kollidera med och vara av en annan typ än resten av spåret. Det gjorde att spåret som nyss skapats inte kolliderade direkt med PacMan men eftersom resten av spåret var av en annan typ så kunde kollisioner med det övriga spåret ske, vilket var önskat.

Alternativ lösning

En alternativ lösning hade varit att inte implementera två säkra rutor utan istället rita upp spåret en aning efter PacMan-figuren. Det hade gjort så att man slapp implementera rutorna och ingen kollison hade uppstått.

Bättre/sämre

Jag valde den lösningen med två säkra rutor eftersom jag ansåg att det såg mycket bättre ut visuellt än om spåret hade kommit någon ruta efter PacMan, och då fick man inte samma känsla av att det var ett spår.

3. **Vad vill jag åstadkomma**

Fylla rutorna som PacMan har omringat.

Hur

Algoritmen som används för att fylla i ett segment av spelplanen som PacMan omringat är algoritmen Flood-fill. Först räknas antal tomma rutor på båda sidor om huvudkaraktären PacMans skapade spår. Därefter fylls det segmentet som innehåller minst antal tomma rutor med algoritmen Flood-fill.

Alternativ lösning

En annan lösning hade varit att fylla det segment som inte innehöll några fiender. Om båda segmenten innehåller fiender så fylls inget av segmenten. Detta var något jag tänkte länge på och försökte implementera. Problemen som uppstod vad dock att jag då hade behövt slå upp alla fiendernas position samt deras ställning relativt det av PacMan skapade spåret. Det hade jag kunnat lösa genom att markera alla de rutor inom segmentet som innehåller fiender med en speciell rut-typ, för att sedan låta bli att fylla de rutor som innehåller den rut-typen. Detta skulle ske varje gång PacMan slutförde spåret.

Bättre/sämre

För själva spelupplevelsen hade det blivit en ganska stor förändring av spelet med den alternativa lösningen. Det finns bra och dåliga sidor med båda implementationerna men jag valde varianten att fylla segmentet med minst antal rutor eftersom jag hade svårt att implementera den andra varianten. Jag personligen tyckte inte det gjorde så stor skillnad, förutom att spelet inte blev identiskt med originalet och eftersom olika typ av lösning i detta problem inte var av så stor vikt för projektet så valde jag den simplare.

4. **Vad vill jag åstadkomma**
Spara highscore i en lista
Hur
Jag har sparat highscore i en highscore-lista i klassen HighscoreList. Poängen jämförs i ScoreCompare och sedan visas de på skärmen som dyker upp efter man förlorat.
Alternativ lösning
En alternativ lösning hade varit att skriva HighscoreList till en fil för att spara resultaten även när spelet stängs.
Bättre/sämre
Den alternativa lösningen hade gett en bättre spelupplevelse eftersom det bästa resultatet hade sparats då man stänger spelet. Jag valde dock att implementera den sämre lösningen då jag valde att använda mig av ljud i spelet istället för filhantering.

5. **Vad vill jag åstadkomma**
Hantera kollision på spelplanen. Kollisioner mellan fiender och PacMan, kollisioner mellan fiender och väggarna, och så vidare.
Hur
Detta implementerades genom att skapa ett interface CollisionHandler och sedan skapa separata kollisionshanteringsklasser för de olika spelobjekten vilka implementerar CollisionHandler.
Alternativ lösning
Jag skulle kunna ha implementerat all kollisionshantering i en och samma klass så att de olika objekten bara hade kallat på olika metoder i den klassen.
Bättre/sämre
Enligt mig blir min nuvarande lösning bättre då den blir tydligare och mer lätthanterlig. Den använder sig även av typiska objektorienterade egenskaper vilket är önskvärt.

6. **Vad vill jag åstadkomma**
Hantera fienderna på spelplanen
Hur
Jag särskiljer fienderna genom att representera dem med olika klasser, men eftersom de även delar många egenskaper implementerade jag den abstrakta klassen Enemy som innehåller variabler och metoder som alla fiender delar.
Alternativ lösning
Jag hade kunnat enbart implementerat klasser för de specifika fienderna.
Bättre/sämre
Den alternativa lösningen gör det mycket svårare att utöka spelet med nya fiende-klasser medan den nuvarande lösningen undviker redundans, gör det enkelt att implementera nya fiender samt utnyttjar fördelaktiga objektorienterade egenskaper.

7. **Vad vill jag åstadkomma**
Implementera en spel-loop på ett bra sätt.
Hur
Jag implementerade själva spel-loopen i klassen Board som en metod kallad tick.
Alternativ lösning
Jag hade kunnat bryta ut själva spel-loopen och implementerat den i en separat klass.
Bättre/sämre
Fördelen med att ha spel-loopen i en separat klass är att det gör det tydligt vad som hanterar vad. Jag har dock inte implementerat det så eftersom alla funktioner för att förändra spelplanen redan finns i

Board och det känns överflödigt att göra ytterligare en klass för att enbart hantera spel-loopen. Board håller koll på det som sker i spelet, och därför är det rimligt att den hanterar loopen.

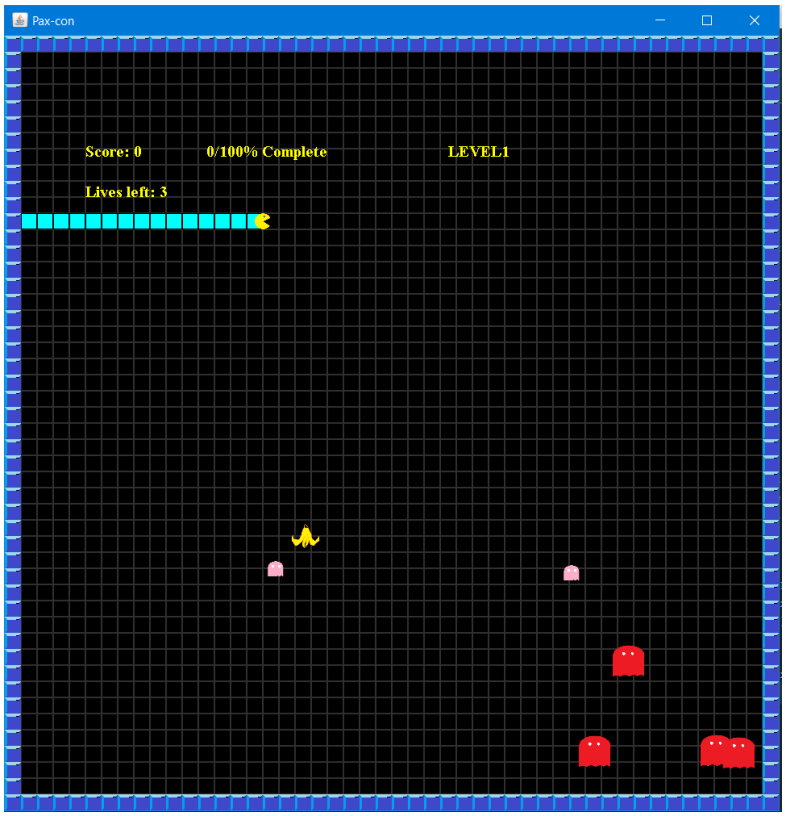
8. **Vad vill jag åstadkomma**
Implementera funktionalitet för power ups.
Hur
Implementationen av power ups ligger separat i de konkreta klasserna SpeedUp samt SlowDownEnemies. De ärver viss funktionalitet från den abstrakta klassen PowerUp.
Alternativ lösning
Jag hade kunnat implementera power ups funktionaliteten i klassen PacMan, eftersom till exempel en power up ändrade farten på PacMan.
Bättre/sämre
Den nuvarande lösningen är helt klart bättre eftersom funktionaliteten ligger tillsammans med det spelobjekt som den tillhör. Hade den alternativa lösningen implementerats hade det kanske i nuläget blivit aningen färre rader kod men nackdelarna hade varit att det hade varit svårare att hitta i koden och svårare att implementera nya power ups.

9. **Vad jag vill åstadkomma**
Hålla PacMan innanför spelplanen.
Hur
Jag implementerade en ram runt spelplanen som består at SquareType:n OUTSIDE. Koordinaterna innanför ramen börjar då alltså på (1,1) eftersom ramen börjar på koordinat (0,0).
Alternativ lösning
Jag hade kunnat skapa spelplanen utan ramen och jämfört om spelobjektens koordianter hamnade utanför spelplanen, alltså var negativa eller större än planens bredd.
Bättre/sämre
Hade den alternativa lösningen valts så hade koden för att kolla om PacMan försökte gå utanför planen blivit längre och mindre läsbar. If-satsen hade kollat koordinaterna (-1, y), (boardWidth, y) för alla y på planen, och motsvarande koordinater i x-led för alla x på planen. Men den valda lösningen behöver if-satsen endast kolla om en ruta är av typen OUTSIDE.

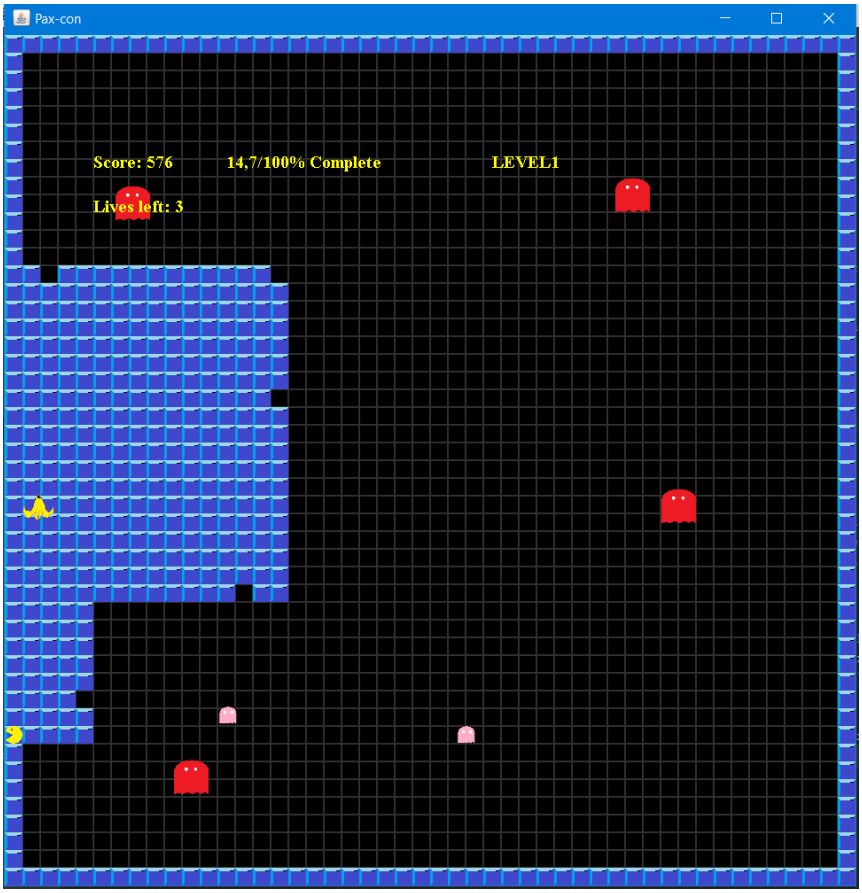
7 Användarmanual

För att starta spelet från start-bilden kan man antingen använda musen, genom att klicka på ”PLAY”, eller så klickar man helt enkelt på enter. När man väl är inne i spelet använder man piltangenterna för att flytta på huvudkaraktären PacMan.

Syftet med spelet är att fylla så mycket som möjligt av spelplanen med blåa rutor genom att med PacMan omringa ett område på skärmen, samtidigt som man undviker fienderna. Fyller man tillräckligt stor del av skärmen med dessa blåa rutor så har man klarat nivån och går vidare till nästa nivå.



Vill man från spel-skärmen se hur många nivåer man låst upp eller pausa spelet kan man klicka på spacebar. För att komma tillbaka till spelet klickar man på spacebar igen.



När man har förbrukat sina tre startliv får man möjlighet till att skriva in ett namn till highscore-listan. Därefter hamnar man på en skärm där man kan välja att spela igen, och även här gör man detta genom ett knapptryck eller ett klick på enter.

8 Slutgiltiga betygsambitioner

Jag siktar på betyg 5 då jag anser mig ha uppfyllt samtliga av de krav som ställts för det betyget. Jag anser att jag utnyttjat och använt mig av egenskaper hos objektorientering och att koden är genomtänkt och håller en god kvalité.

9 Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
  - Det gick bra att hitta tid för att arbeta på projektet, och jag har följt arbetsplanen. Jag har lärt mig massor om objektorientering genom att applicera det på det här projektet. Det tog lite för lång tid att implementera vissa algoritmer.
- *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 vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!*
  - Jag har mest läst på om objektorientering och hur man ska strukturera upp objektorienterade projekt på nätet. Jag lärde mig en del från att utföra kursens labbar.
- *Har ni lagt ned för mycket/lite tid?*
  - Jag har lagt ner ungefär så mycket tid som kursmomentet ska ta, så jag anser att jag lagt ner lagom mycket tid.
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
  - Eftersom jag utfört projektet själv är inte frågan relevant.
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
  - Det har varit bra att ha en plan att utgå ifrån, och jag tycker att milstolparna har varit mest användbara. Jag har kunnat pricka av dom efterhand, och det har varit bra för motivationen och för att känna att projektet gått framåt.
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
  - Jag har följt arbetsmetodiken och det har fungerat som jag tänkt mig.
- *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.*
  - Det har gått väldigt bra, så jag har inget att skriva på denna punkt.
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
  - Att verkligen ta tid till att bestämma projektets struktur innan man börjar implementera det. Till exempel att bestämma i förväg vilka interfaces, abstrakta klasser, konkreta superklasser och superklasser som behövs. Även om det såklart kan komma att ändras efterhand så är det bra att ha en plan från början.
- *Har ni saknat något i kursen som hade underlättat projektet?*
  - Nej, jag tycker att allt har fungerat väldigt bra.
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
  - Nej.