Projektbeskrivning

Dot Desktop Version

2019-03-12

Projektmedlemmar:
Theodor Lindberg theli316@student.liu.se

Handledare:
Nigel Cole nigco547@student.liu.se

Table of Contents

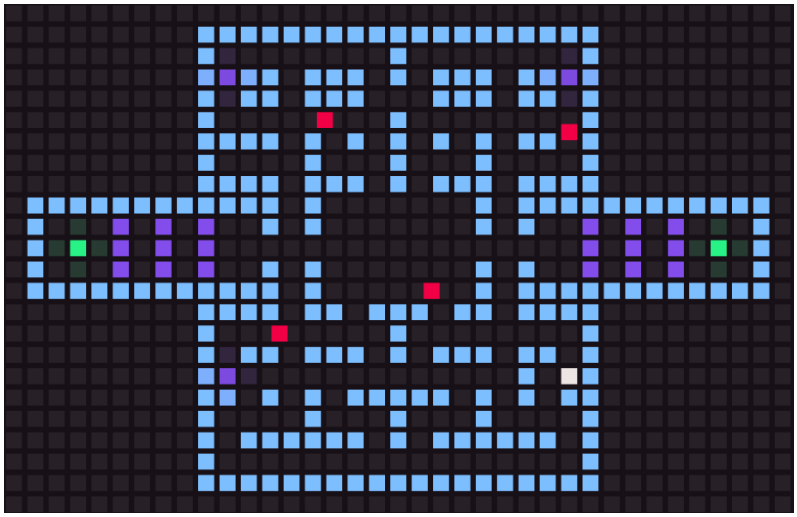
Planering

1. Introduktion till projektet

Dot Desktop Version är inspirerat av spelet Dot. Dot är ett 2D spel där spelaren styr en kvadrat och försöker ta den till målet, men för att ta sig dit måste spelaren lösa pussel och undvika fiender. Dot Desktop Version kommer vara likadant som Dot fast med några andra funktioner samt att det kommer att köras lokalt på datorn och inte i webbläsaren.

2. Ytterligare bakgrundsinformation

Bilden nedanför visar hur en nivå kan se ut. Hur alla olika block fungerar beskrivs längre ner.



Beskrivning av samtliga block (färgkodningen är den samma som i originalversionen för att underlätta för läsaren):

Spelaren:

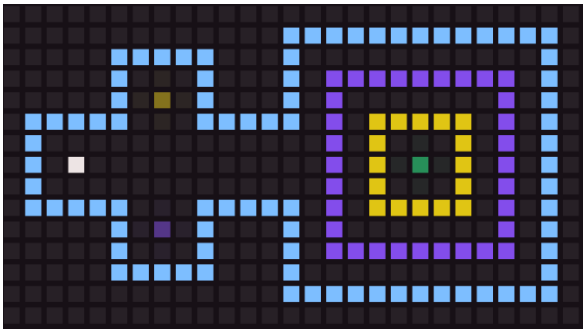
Spelaren är en vit ruta. När kvadraten rör sig, bildas en animation efter kvadraten – en ”svans” som sedan tonas bort.

Oförstörbar vägg:

Varje nivå omgivs av en oförstörbar vägg som hindrar spelaren och fiender från att åka utanför. På bilden är den ljusblå.

Nyckelblock:

Nyckelblock är antingen gula eller lila och har en pulserande animation. Nyckelblock används för att ta bort/låsa upp andra block och aktiveras när spelaren åker på dem. Nyckelblocket försvinner sedan. Nyckelblock kan bara ta bort block av samma färg, samt alla grannar av samma färg, men bara om det finns en väg dit. På bilden nedan ser man ett exempel där spelaren skulle vara tvungen att ta det lila nyckelblocket först då det gula nyckelblocket inte skulle komma åt den gula väggen.



Vägg:

En vanlig vägg är antingen gul eller lila och kan tas bort utav nyckelblock.

Fiender:

Fiender representeras som röda block. Fiender kan bara röra sig rakt framåt på banan, men kan ha 3 olika beteenden när de kolliderar med en vägg: svänga vänster, svänga höger eller vända. Om det finns tid över kan ett mer avancerat ai implementeras.

Mål:

Slutblock är målet för spelaren att ta sig till. Slutblocket är grönt och har en pulserande animation. Notera att det kan finnas flera mål på en nivå.

Beroende på hur mycket tid jag har, kommer jag antingen skapa en ”level-editor” för att göra nya nivåer samt att spara dem. Annars tänker jag läsa in bitmap filer som nivåer. Med bitmap bilder skulle man kunna skapa nivåer i t.ex. Paint där varje pixel representerar en typ av block.

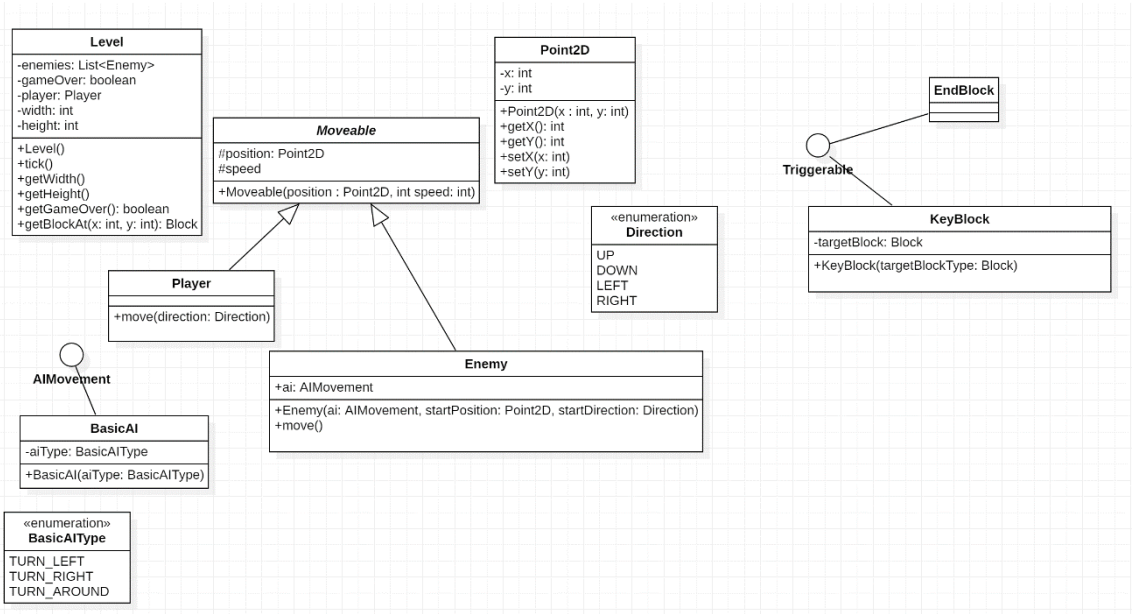
3. Milstolpar

#	Beskrivning
1	Det finns en data typ för nivå och block.
2	Det finns ett grafiskt fönster att visa nivån i, går även att kryssa ner fönstret.
3	Spelaren kan styra sin kvadrat och hantering av kollision är implementerad.
4	Vanliga väggar och nyckelblock är implementerade.
5	Fiender med standard ai finns och kan döda spelaren.
6	Det finns game over och man kan starta om nivån.
7	Mål är implementerat och man klara av nivån.
8	Det går att starta om nivån när man vill.
9	Det finns animation för spelaren.
10	Det finns animation för nyckelblock och slutblock.
11	Det finns animation för när block tas bort av nyckelblock.
12	Det visas en bild när man startar programmet.

13	Det finns ett menyalternativ för att byta till ”level-editorn”.
14	Samtliga block som ska gå att placera ut finns som alternativ på sidan.
15	Ett tomt rutnät ska ritas ut.
16	Man ska kunna välja vilket typ av block man ska placera ut.
17	Det går att placera ut det valda blocket i rutnätet.
18	Det går att ta bort block från rutnätet.
19	Det går att spara nivån man har skapat till en fil.
20	Det går att läsa in nivån och spela den.
21	Det finns menyalternativ för att välja vilken nivå man ska spela.
22	Det går att ladda in en existerande nivå i editorn så man inte behöver skapa en helt ny varje gång.
23	En ny typ av nyckelblock finns där de åker runt med samma ai som fienderna.
…	

4. Övriga implementationsförberedelser

Jag har börjat skissa ett UML diagram, det är långt från klart men visar några av de grundläggande förhållandena och ansvaren mellan klasserna.



5. Utveckling och samarbete

Då jag är ensam i gruppen så kommer jag kunna arbeta på håltimmar, kvällar och helger.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

#	Beskrivning
1	Det finns en data typ för nivå och block. [HELT]
2	Det finns ett grafiskt fönster att visa nivån i, går även att kryssa ner fönstret. [HELT]
3	Spelaren kan styra sin kvadrat och hantering av kollision är implementerad. [HELT]
4	Vanliga väggar och nyckelblock är implementerade. [HELT]
5	Fiender med standard ai finns och kan döda spelaren. [HELT]
6	Det finns game over och man kan starta om nivån. [HELT]
7	Mål är implementerat och man klara av nivån. [HELT]
8	Det går att starta om nivån när man vill. [HELT]
9	Det finns animation för spelaren. [INTE ALLS]
10	Det finns animation för nyckelblock och slutblock. [INTE ALLS]
11	Det finns animation för när block tas bort av nyckelblock. [INTE ALLS]
12	Det visas en bild när man startar programmet. [INTE ALLS]
13	Det finns ett menyalternativ för att byta till ”level-editorn”. [HELT]
14	Samtliga block som ska gå att placera ut finns som alternativ på sidan. [HELT]
15	Ett tomt rutnät ska ritas ut. [INTE ALLS, behövs inte]
16	Man ska kunna välja vilket typ av block man ska placera ut. [HELT]
17	Det går att placera ut det valda blocket i rutnätet. [HELT]
18	Det går att ta bort block från rutnätet. [HELT]
19	Det går att spara nivån man har skapat till en fil. [HELT]
20	Det går att läsa in nivån och spela den. [HELT]
21	Det finns menyalternativ för att välja vilken nivå man ska spela. [HELT]

22	Det går att ladda in en existerande nivå i editorn så man inte behöver skapa en helt ny varje gång. [HELT]
23	En ny typ av nyckelblock finns där de åker runt med samma ai som fienderna. [INTE ALLS]
…	

6.2. Dokumentation för programstruktur, med UML-diagram

6.2.1 Övergripande programstruktur

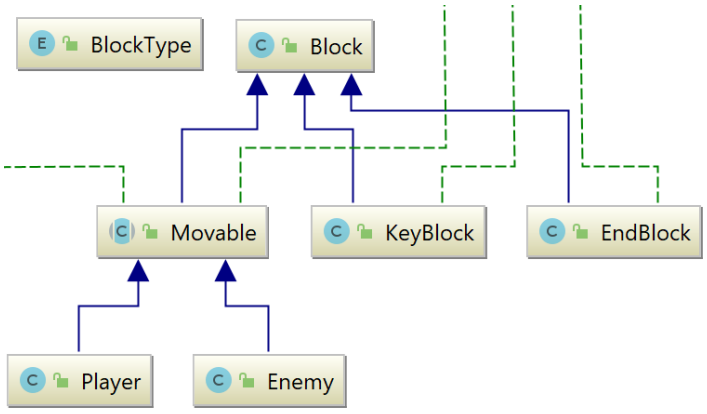
Programmerat startas från main-metoden i *LevelViewer* genom att skapa ett *Level*-objekt utifrån en nivå som följer med programmet. Sedan skapas en instans av *LevelViewer* med *Level*-objektet som parameter. *LevelViewer* kommer sen skapa ett Game-objekt från *Level*-objektet, sätta upp menyn via *initializeMenuBar*, sätta upp tangentbordsinmatning via *addKeyboardEvents* och starta spelet via *initializeGameLoop*. *Timern* som startas anropar *tick*-metoden i *Game* med 16ms mellanrum vilket motsvarar ca 60 FPS (om datorn klarar av det).

Game håller ihop hela spelet och det mesta logiken ligger hos de andra klasserna i *Game*-paketet, till exempel är det *Movable* själv som ansvarar för kollisioner och integrerar med andra block och nyckelblocken *KeyBlock* vet själva hur ska ta bort andra block.

När man ska ändra en nivå så tas *GameComponent* bort från *LevelViewer* och man skapar en *LevelComponent* istället, sedan läggs även en *LevelEditor* till. För att musklickningar ska upptäckas så läggs även *BlockPlacer* till som *MouseListener* och *MouseMotionListener* till *LevelComponent.* När man återgår till speltillståndet så tas *LevelComponent* bort och en *GameComponent* läggs till istället dessutom tas *LevelEditor* bort från *LevelViewer.*

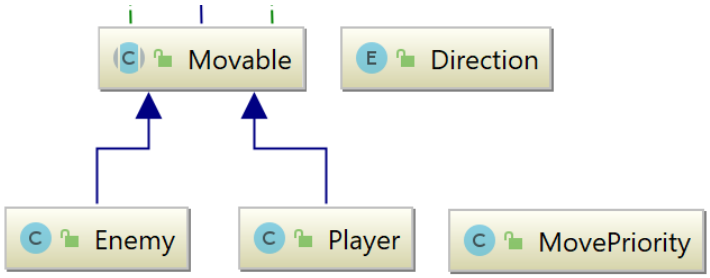
All loggning sker till standardkonsolen, det vill säga i IntelliJ så syns informationen i ”Run”-konsolen och när jar-filen körs så kommer texten till konsolen som startade filen.

6.2.2 Struktur för olika typer av block



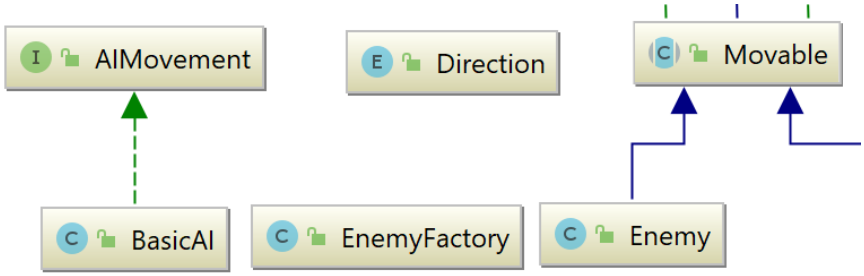
I detta diagram ser vi att alla spelobjekt ärver från klassen *Block,* det är för att alla spelobjekt är en typ av *Block* har alla varsin *BlockType*. *BlockType* är en *enum* av alla tillgängliga typer av block och beskriver även om det går att gå igenom dem. Enklare block som till exempel väggar har inte en egen dedikerad klass utan använder *Block*-klassen direkt.

6.2.3 Struktur för objekt som rör sig



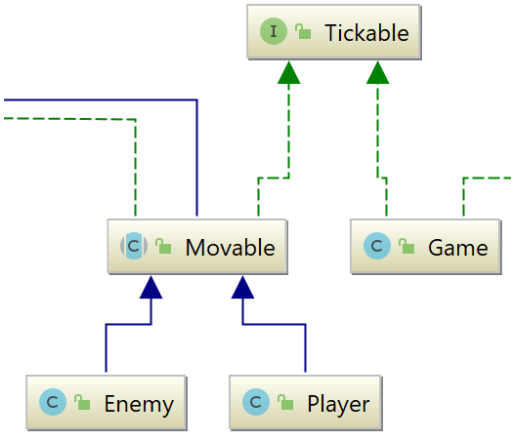
I det här diagrammet kan vi se att *Player* och *Enemy* ärver från samma abstrakta klass *Movable*, det är för att de rör sig mellan block och detekterar kollisioner på samma sätt men kan hantera kollisioner och hur de rör sig på olika sätt. *Movable* har en abstrakt metod som heter *handleCollision* som *Enemy* och *Player* sedan själva får implementera (*override*) och *Movable* kan fortfarande använda den i *move.*

6.2.4 Struktur för olika fiender skapas och är uppbyggda



Det här UML-diagrammet visar hur olika fiender skapas. Alla fiender behöver samma pekare till *Game* så tar *EnemyFactory Game* som en parameter som den sedan kan skicka vidare till alla fienders konstruktorer. För att skapa en fiende behöver man sedan skicka med en startposition *Point2D*, startriktning *Direction*, rörelsehastighet *Speed* och även vilken artificiellintelligens den ska ha *EnemyAI*. Det kan finnas många olika typer och konfigurationer av *AIMovement* så har *EnemyFactory* en *enum EnemyAI* som ska motsvara en viss implementation av *AIMovement*, varje implementation kan även ha olika konfigurationer. Att göra en fiendefabrik kräver lite hårdkodning med *enumen EnemyAI* men gjorde arbetet mycket smidigare för *LevelEditor* att genera en lista med tillgängliga typer av artificiellintelligens, dessutom är all logik samlad i en fabrik vilket gör att underhållet inte blir så stort.

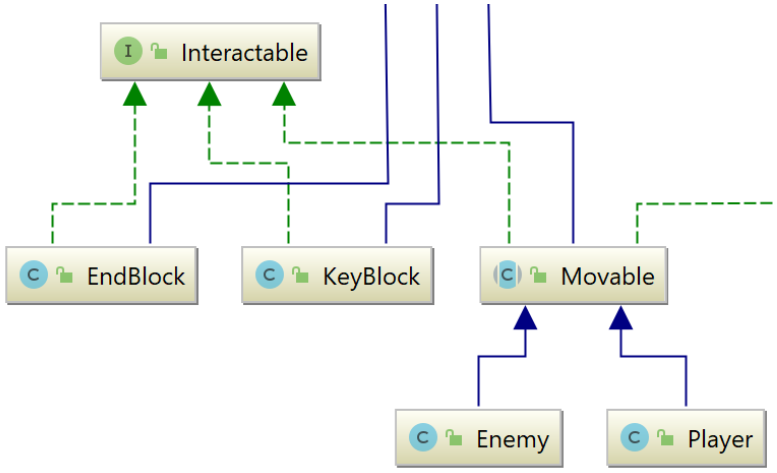
6.2.5 Struktur för vad som uppdateras varje ”tick”



I det här UML-diagrammet ser vi hur grunden för alla objekt som behöver uppdateras är uppbyggd. Samtliga klasser behöver uppdateras implementerar *interfacet Tickable* som lovar att objekten har metoden *tick*. *Game* har en lista med alla objekt som implementerar *Tickable* så när *LevelViewer* kallar på *Game’s tick*-metod så itererar *Game* över alla objekt som behöver

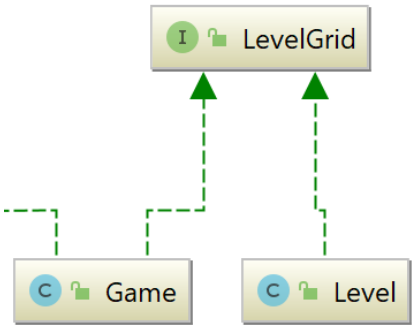
uppdateras och anropar deras *tick*-metoder. Det här är ett bra exempel på hur *interfaces* kan vara användbara genom att ge ett kontrakt kring vilka metoder som måste finnas medan varje implementation av metoderna kan vara helt olika. *Game* kan sedan notifiera sina lyssnare (*GameComponent*) att spelet har uppdaterats.

6.2.6 Struktur för vad som går att integrera med



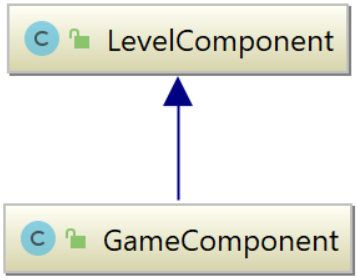
I metoden *move* i klassen *Moveable* så kollar man om blocket som man går på går att integrera med och om det går så kallar med dess *interact*-metod och skickar med sig själv som parameter. Även här ser vi att man inte behöver veta vad det är för objekt så länge de har en *interact*-funktion.

6.2.6 Struktur för sambandet mellan Game och Level



Både *Game* och *Level* implementerar *interfacet LevelGrid*, detta gjordes för att *LevelComponent* ska kunna rita ut vilken som helst då de båda har funktionerna som krävs.

6.2.7 Struktur för hur spelet och nivån ritas ut



LevelComponent tar *LevelGrid* som parameter och ritar sedan ut alla block i *paintComponent*. *GameComponent* kommer på samma sätt rita ut alla block men behöver även rita ut alla objekt som rör på sig så ärver *GameComponent* från *LevelComponent* och tar *Game* som en parameter som den sedan kan skicka vidare till *LevelComponent*. Sen när *GameComponent* behöver rita ut så kan den först kalla på *super*-klassen för att sen rita ut objekten som rör sig.

6.3. Användning av fritt material

Jag har använt mig av Gson där jag även behövde ladda ner en enskild fil från *Gson-repot*, *RuntimeTypeFactoryAdapter,* då den inte var inkluderad i jar-filen för att hantera arv i JSON-formatet. Jag har även använt en klass från StackOverflow, *InterfaceTypeFactoryAdapter*, för att hantera interface som medlemsvariabel. Sökalgoritmen som används i *KeyBlock* är lånad (står ”borrowcode” i metodnamnen). Koden för lista alla resursfiler är även lånad i *FileHandler* (står ”borrowcode” i metodnamnen).

6.4. Motiverade designbeslut med alternativ

1. Jag ville att nivåer ska kunna läsas från och sparas till filer på ett flexibelt sätt. För att uppnå detta valde jag att använda JSON-filer med hjälp av Gson-biblioteket. För att spara en nivå kallas metoden *saveLevel* i *Level* med ett filnamn som parameter som sedan kallar på *saveLevel* i *FileHandler* och skickar med en två dimensionell *array* av *Block*-objekt som representerar nivån. Gson gör sedan en representation av *arrayen* till en sträng som sen kan sparas till en fil. Några andra alternativ som övervägdes var att spara som XML eller binärfiler med hjälp av *serialization.* Binärfiler uteslöts då det inte är lika lätt om andra program skulle behöva läsa filerna, minnesutrymmet för filerna var inget bekymmer och textbaserade filformat är lättare att felsöka under utveckling. XML användes inte eftersom JSON är smidigare om man i framtiden jobbar mot Javascript i webapplikationer och dessutom var jag redan bekant med Gson-bibioteket.
2. Jag ville ha ett sätt att rita ut objekten på. Jag löste det genom att använda mig av MVC-mönstret så objekten själva vet inte hur de ser ut utan det gör endast *LevelComponent* och *GameComponent*. En annan lösning vore att spelobjekten själva vet hur de ser ut eller har en egen ritklass men jag bedömde min lösning som okej i detta fall då grafiken var så pass simpel och allt ritas ut nästan likadant utan animationer.
3. Jag ville kunna ändra och skapa banor utan att skapa ett helt *Game*-objekt och jag ville inte heller att någon annan ska kunna ändra nivån i *Game* när spelet körs. Jag skapade därför en separat *Level*-klass som har metoder för att kunna spara eller ändra. Detta är inte endast ett decentraliseringsbeslut utan även ett säkerhetsbeslut. Klasser från andra paket kan ändra block i ett *Level*-objekt hur som helst (till exempel *LevelEditor*) men när *Game* skapas så skapas nya *Block*-objekt med hjälp av kopieringskonstruktorer så att man inte har kvar samma referenser till objekten.
4. Jag ville att användaren på ett enkelt sätt kunna spela några nivåer utan att behöva skapa nya nivåer för att kunna lära sig/testa spelet. Jag bestämde mig för att i LevelViewer utöka *Load Game* menyn (JMenu) och lägga en undermeny för inkluderade nivåer som sedan skulle lista dem och ett JMenuItem för att fortfarande kunna välja en nivå från datorns filsystem. För att uppnå detta skapade jag först en metod i *FileHandler* som hette *getLevelsFromResources* som returnerar en lista av namn på alla nivåer från resursmappen. Nu skapar *LevelViewer* i metoden *initializeMenuBar* flera JMenuItems utifrån namnlistan och lägger till dem under *Included Levels*, detta sker alltså helt dynamiskt. Sedan när *LevelViewer* kallar på *readLevel* i *FileHandler* så kollar metoden om filnamnet finns bland resurserna, om den gör det så används metoden *readLevelFromResources* och annars *readLevelFromFile*. En annan lösning skulle vara att man skapar en mapp i user-mappen och kopierar alla nivåer från jar-filen dit. Eller att man kopierar dem till en temporär mapp och sedan raderar dem när programmet avslutas. Dessa lösningar användes inte då jag inte ville få

filer som ligger kvar och ”skräpar” eller riskera att de ligger kvar. Om man vore ett större företag skulle man kanske också ha en hemsida där man själv kan ladda ner nivåer ifrån eller skapa riktiga installationsfiler som på ett strukturerat sätt kan sätta upp i filsystemet och sedan städa vid avinstallation.

5. Jag ville skapa ett användargränssnitt för att kunna skapa nya nivåer och ändra befintliga. Jag löste det genom att skapa en egen *JComponent* som heter *LevelEditor* och en *LevelViewer* samt *GameComponent.* Så när man ska ändra en nivå så läggs en *LevelEditor* in i samma *JFrame* som *LevelViewer* har och *GameComponent* byts ut mot en *LevelComponent.* Detta sker i *showLevelEditor* när användaren klickar *Options-Editor* i menyn. En annan lösning är att öppna upp ett till fönster, detta vore lättare att ta upp och stänga ner än att behöva ändra i fönstret man spelar i men däremot blir det svårare kommunicera mellan fönsterna samt förvirrande för användaren. En ytterligare lösning skulle vara att inte göra en *LevelEditor*-komponent och använda lägga till enskilda komponenter direkt i fönstret vilket skulle ge samma resultat för användaren men istället bryta mot decentraliserings principer även om det blir lite lättare när allt blir i samma klass.
6. För att få nyckelblocken att hitta vilka block som går att ta bort behövde jag en använda någon slags algoritm. Jag såg att algoritmen skulle behöva bete sig som ”flood fill”-algoritmer för sökningen. Nästan alla exempel jag hittade av rekursiva lösningen eftersom som den typen av algoritmer är rekursiv i naturen men jag hittade en lösning som använder sig av en lista. Jag använde mig av den icke-rekursiva lösningen eftersom jag ville spara på stacken. En alternativ lösning var såklart att använda en rekursiv implementation, det skulle säkert även gå i det här spelet men jag ville ha framförhållning för större nivåer. En annan lösning kan vara att man går igenom alla block som nyckelblocket vill ta bort och kollar sedan om det finns en väg emellan dem, men denna lösning skulle kräva att man särbehandlade dessa block på något sätt så att man enkelt kan iterera över dem.
7. För att driva spelet framåt behövde jag någon typ av *gameloop*, jag ville att denna skulle vara enkel i början och senare i projektet kanske byta till en bättre fast mer avancerad. Jag löste det genom att använda *java.util.Timer* för att kalla på *gameTick*-metoden i *LevelViewer*, detta initieras i *initializeGameLoop*. En annan enkel lösning är att kalla på *gameTick* i en *while*-loop och använda *Thread.sleep* för att sakta ner spelet. Men *Thread.sleep* har nackdelen att man inte utnyttjar ”väntetiden” till att göra någonting annat som till exempel hantera alla inputs och events inputs som har pumpats. Men båda lider av problemen som uppstår om man har en förlångsam dator, förflyttningshastigheter hos objekt är direkt beroende av uppdateringshastigheten och att de inte är så exakta till att börja med. Detta skulle kunna lösas med att
8. Jag ville skapa ett flexibelt sätt att ge fiender olika typer av rörelsemönster men i grunden vara lika och göra det enkelt i framtiden om fiender ska behöva kunna byta rörelsemönster. Jag löste det genom att skapa ett *interface AIMovement* som hade metoderna *move, handleCollision* och *copy*. *Enemy* har sedan ett fält *AIMovement* som den kan delegera all logik till. En annan lösning är att ha flera villkorssatser som grenar beroende på vilket tillstånd man är i, men detta skulle kräva att man direkt ändrar i *Enemy*-klassen varje gång och då måste man ha tillgång till källkoden, dessutom skulle det bryta mot decentraliserings principer om det blir för mycket. En ytterligare lösning som decentraliserar logiken är att *Enemy* vore en abstrakt klass och man gjorde en ny klass för varje variation, till exempel *EnemyPlayerFollower*, men detta skulle betyda att

man blir tvungen att skapa en nytt objekt om fienden skulle ändra logiken under spelets gång vilket inte är önskvärt.

9. Jag ville ha ett sätt att alltid uppdatera grafiken efter att spelet har förändrats så de aldrig kommer i otakt. Min lösning var att använda *Observer* mönstret, implementationen är en abstrakt *Observable*-klass och ett *Observer*-interface. Övervakare (*Observer*) kan läggas som övervakare hos *Observable* (*addObserver)* som har en lista med övervakare. Klassen som ärver från *Observable* kan sen kalla på *notifyObservers* för uppmärksamma att en förändring har skett. En annan lösning som jag har sett ofta är att man uppdaterar varje tick i spelet, nackdelen med detta är att man kommer rita om även om ingenting har förändrats. Andra mönster som används är även *Publisher/Subscriber* mönster som konceptuellt rätt så likt *Observer* mönstret men här behöver *Publisher* och *Subscriber* inte nödvändigtvis veta om varandra direkt och går ofta via en mellanhand med meddelandeköer. Detta *Observer* mönster är även användbart andra saker som kan behövas i framtiden, speciellt i MVC-modellen där en komponent kan bindas till en viss data.

7. Användarmanual

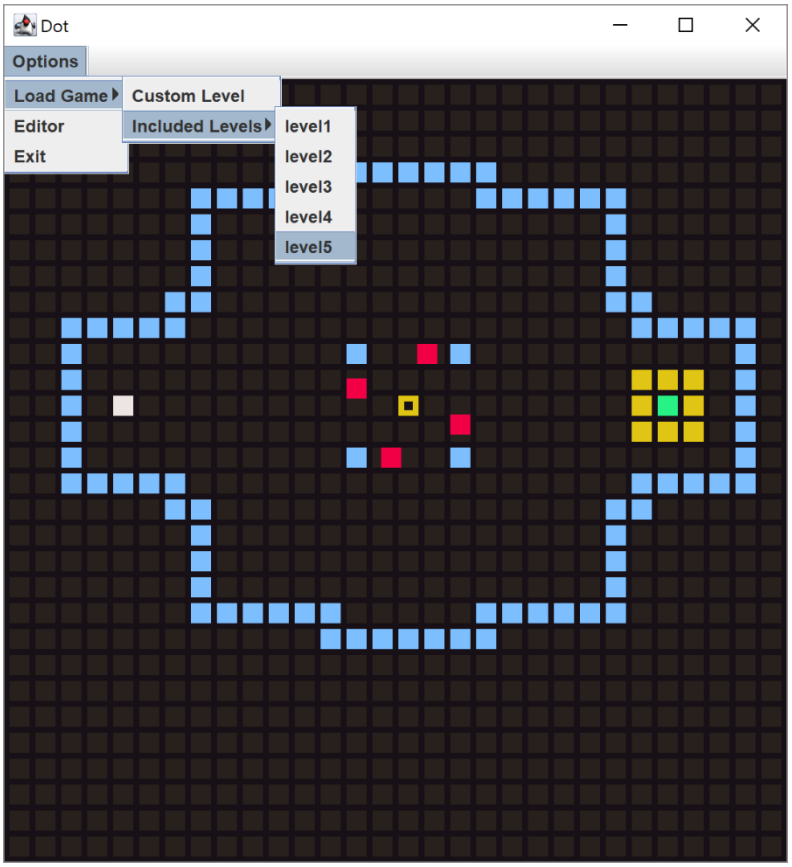
Programmet kan antingen startas från IntelliJ genom att köra main-metoden i *LevelViewer* eller köra jar-filen som byggs från kommandotolken via ”java -jar DotDesktop.jar” om man står i *out/artifacts/DotDesktop_jar*. Spelet startar upp direkt med en nivå från resurs-mappen.

7.1 Kontroller i spelet

För att styra spelaren används piltangenterna. För att starta om spelet används ’r’-tangenten och för att stoppa/starta spelet används ’p’-tangenten.

7.2 Ladda in en annan nivå

För att ladda in en annan nivå väljer man i menyn *Options-Load Game*, sen kan man välja på att antingen ladda in en egen nivå som man har sparad på datorn och då väljer man *Custom Level*, men för att välja en nivå som ingår väljer man istället något från listan under *Included level*

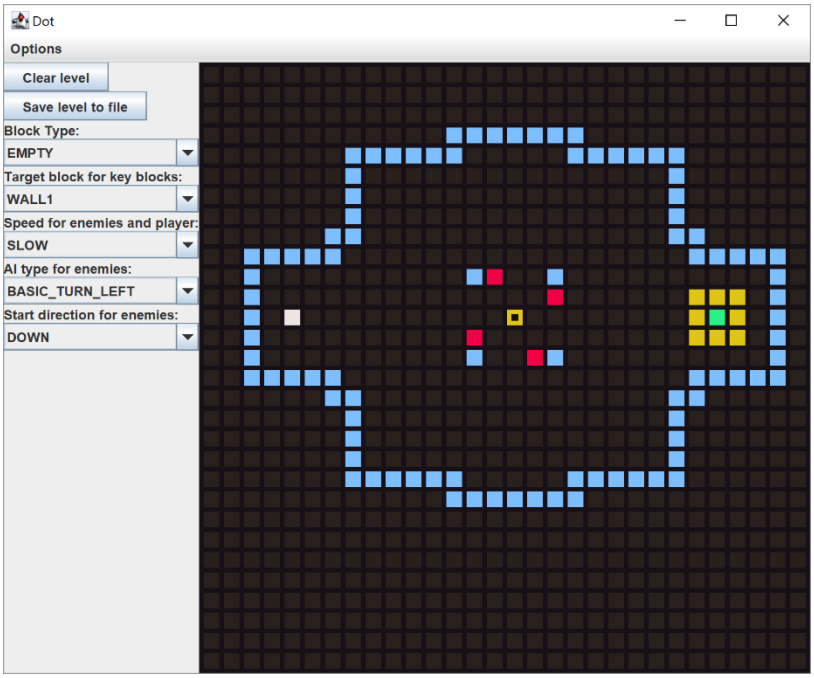


7.3 Skapa en egen nivå

För att ändra en existerande så måste man först ladda in nivån enligt ovan och sen väljer man i menyn *Options-Editor*, sen kan man klicka på *Clear level* för att göra en nivå från grunden.

7.4 Placera ut block i LevelEditor

Börja med att välja blocktypen i den första rutan. Du kan nu placera ut den typen av block genom att klicka med vänster musknapp. För nyckelblock kan du välja vilken typ av block som ska tas bort med hjälp av rutan nedanför. Hastigheten för fiender och spelaren välj under ”Speed for enemies and player”. Startriktning samt artificiellintelligens för fiender väljs under ”AI type for enmies” och ”Start direction for enemies”. Block kan placeras ut ovanpå ett annat men kommer då skriva över det som var där innan. Block för spelaren är unikt och kan därför endast placeras ut endast en gång. Det är dessutom inte nödvändigt att ha väggar runt hela nivån då spelaren ändå inte kan röra sig utanför nivån. Nivån är alltid 30x30 block stort och kan inte ändras av användaren, spelet kan så klart klara andra storlekar men detta har inte lagts till som val.



7.5 Ta bort block i LevelEditor

Man kan antingen välja att placera ut *EMPTY*-block med hjälp av listan det finns ett kortkommando för att göra samma sak genom att klicka på höger musknapp.

7.6 Spara en nivå

För att spara en nivå man har gjort så klickar man på *Save level to file*, man får sedan med hjälp av JFileChooser välja vart man ska spara nivån och vad den ska heta. Det går inte att spara ett speltillstånd, utan bara nivåer i ”Level-editorn”.

7.7 Avsluta spelet

För att avsluta spelet kan man antingen välja kryss-knappen i högra hörnet eller gå via menyn *Options-Exit*, du kommer sen få en fråga om du säker att du vill avsluta.

8. Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
  - Det som gick bra var att hitta tid och plats att arbeta på, men det var till stor hjälp att utveckla på sin laptop istället för att vara bunden till LIUs datorer.
- *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 tyckte att föreläsningarna gav väldigt bra information och tankesätt så jag gick på alla. Men jag hade velat haft en extra frivillig föreläsning kring GUI programmering och olika layouts (även om detta inte är fokus i kursen) då jag tror att många hade lite svårt för det.
  - Handledda labbar var väldigt bra för att diskutera designbeslut och kodarkitektur.
  - Internet var en viktig källa för att ta reda på vilka verktyg och vilka syntaxer som används.
- *Har ni lagt ned för mycket/lite tid?*
  - Jag tror att jag har lagt mer än den rekommenderade tiden 80 timmar.
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*

    - Arbetsfördelningen var jämn eftersom jag arbetade ensam.
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
    - Jag tyckte att det var bra att man behövde göra milstolpar innan och man kanske även ska kräva några UML-diagram så att man blir tvungen att fundera lite extra på koddesignen och inte bara funktionalitet.
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
    - Allt har fungerat som jag har 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.*
    - Under labbtillfällena var de flera som ville ha hjälp att felsöka buggar eller kraschar så det kunde ta mycket tid att hjälpa grupperna. Jag tycker att sådana problem ska man i allra högsta grad försöka lösa själv då labbassistenten måste sätta sig in i hela kodbasen för att kunna hjälpa, tillfällena borde vara till för designbeslut och kodarkitektur.
- *Vilka tips skulle ni vilja ge till studenter i nästa årskurs?*
    - Gör UML-diagram innan även om man inte måste, det hjälpte mycket.
    - Gör något som är lätt att göra grafiskt.
    - Kolla på betygskraven och lägg in dem i milstolparna så att man inte måste planera in det sen.
    - Arbeta på rapporten under projektets gång.
- *Har ni saknat något i kursen som hade underlättat projektet?*
    -
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
    - Flera har programmerat innan så det vore lättare att förstå (och intressant) att höra fler liknelser med andra programmeringsspråk.
    - Det skulle vara bra om det fanns länkar till youtube-videos som ni tycker är bra, kanske inte till allt men om man vill fördjupa sig i trådar, GUI eller något annat.