Projektbeskrivning

Mirrorinth

2018-02-19

Projektmedlemmar:
Daniel Norström danno126@student.liu.se
Simon Johansson simjo009@student.liu.se

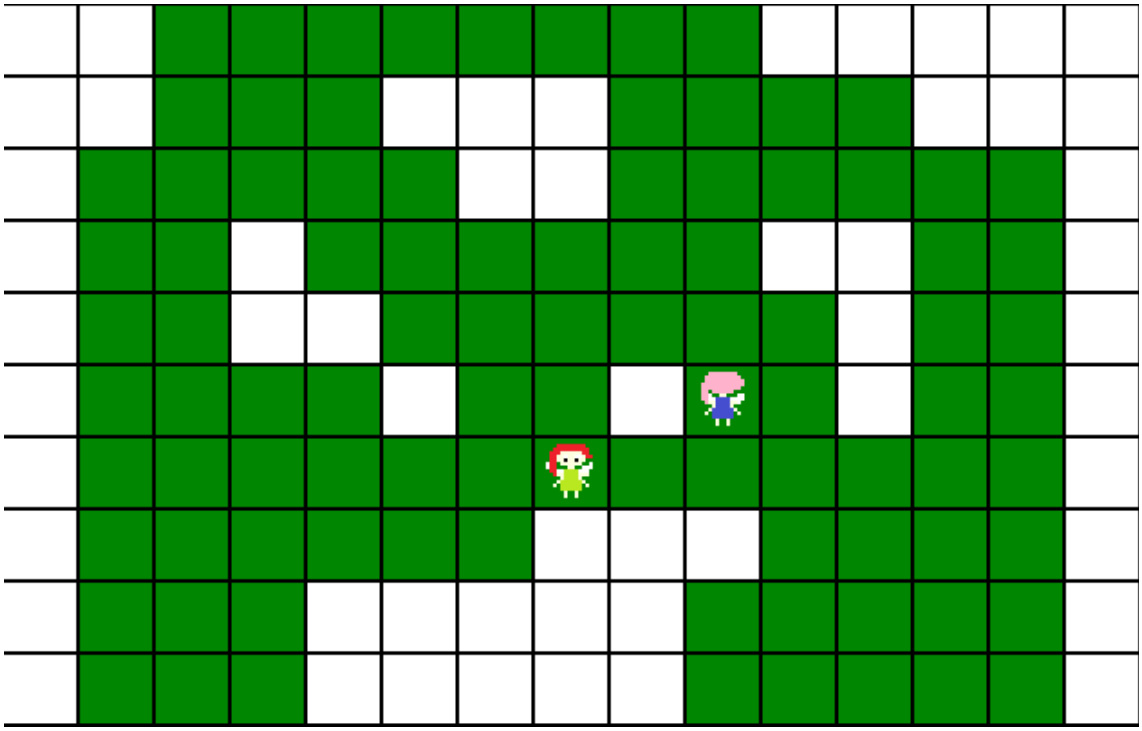
Handledare:
Piotr Rudol piotr.rudol@liu.se

Table of Contents

Planering

1. Introduktion till projektet

Ett "top-down, 2d-grid" spel där målet är att navigera två karaktärer genom en labyrint. De två karaktärerna börjar på motsatta sidor av labyrinten, och spelaren vinner när hon lyckas föra samman karaktärerna. Haken är att spelaren styr båda karaktärerna samtidigt, med samma tangenter, och den enas kontroller är inverterade; detta innebär att om spelaren trycker ned "upp"-tangenten, så kommer den ena karaktären att röra sig uppåt, medan den andra kommer röra sig nedåt.



2. Ytterligare bakgrundsinformation

Utöver det som beskrivs ovanför är det värt att nämna att spelaren egentligen kontrollerar en karaktär, nämligen den som faktiskt följer spelarens instruktioner, dvs. karaktären som rör sig nedåt när spelaren trycker ned ”ned”-tangenten. Den andra karaktären bara imiterar, fast spegelvänt. Vad detta innebär för spelaren är att ifall den första karaktären står mot en vägg på höger sida, och försöker gå höger, så kommer karaktären inte röra sig, och den andra karaktären kommer inte heller röra sig, då det inte fanns någon rörelse att imitera. Ifall den andra karaktären stod mot en vägg på höger sida däremot, och spelaren tryckte ned ”vänster”-tangenten, så kommer den första karaktären gå åt vänster, men den andra kommer inte röra sig alls, då den försöker gå åt höger. Detta kommer spelaren behöva använda sig av för att klara av många av nivåerna i spelet, då det låter henne omplacera sina karaktärer (spegelvänt) relativt varandra.

Enligt det som beskrivs som utökningar under nedanstående rubrik kommer spelet eventuellt innehålla allehanda faror för karaktärerna att undvika, såväl som förmågor som kan hjälpa dem att förenas med varandra.

3. Milstolpar

#	Beskrivning
1	Det finns en grafiskt uppritad spelplan där en handgjord nivå bestående av väggrutor och markrutor kan visas upp. Informationen för en sådan nivå ska åtminstone för tillfället lagras som en slags lista på koordinater.
2	Det skall finnas två karaktärer på spelplanen som reagerar på att piltangenterna trycks ned. Detta kommer få karaktärerna att förflytta sig på spelplanen.
3	Collision detection skall implementeras, så att karaktärerna inte kan gå genom väggar, och så att när karaktärerna rör varandra kommer ett ”you win” meddelande visas, och spelet kommer sedan övergå till nästa nivå, förutsatt att en sådan finns.
4	Rutor som fungerar som knappar som kan tryckas ned för att byta ut en eller flera rutor av vägg till mark, och/eller mark till vägg. Denna knapp skall kunna tryckas ned flera gånger, och alltid ändra samma rutor mellan mark och vägg.
5	Rutor som fungerar som knappar, som kommer i par, där de två karaktärerna kan ställa sig på varsin knapp för att öppna upp en ny väg av mark. Ett eller flera nya markblock kommer skapas där det tidigare fanns vägg. Ifall enbart en karaktär står på en knapp så kommer ingenting hända. Knappar försvinner när de båda tryckts ned samtidigt.
6	Farliga lavarutor som ger spelaren ett "game over" när en karaktär ställer sig på dem. Detta skulle tvinga spelaren att starta om från början av nivån.
7	Rutor som fungerar som spindelnät, vilket är en ruta som karaktärerna "fastnar" i en kort tid, vilket innebär att det skulle krävas flera tangenttryckningar för att karaktären skulle lämna rutan. Därmed skulle den fria karaktären kunna gå flera steg medan den andra satt fast i spindelnätet, förutsatt att det inte var den ledande karaktären som fastnat. Efter 5 knapptryckningar skulle karaktären bryta sig fri och ta sig ut ur spindelnätet.
8	Teleporter-rutor som kommer I par och kan teleportera karaktärer mellan sig.
9	Ge karaktärerna möjlighet att plocka upp "items" som kan finnas i nivåerna, vilka skulle ge karaktären i fråga nya förmågor som kan aktiveras genom att mellanslagstangenten trycks ned. Skapa ett sådant item som skulle låta karaktären gå igenom en singulär vägg för att ta sig till en ruta mark på andra sidan, såsom ett spöke. Efter att ett item har använts en gång skall karaktären inte längre ha kvar det, men det bör dyka upp på nytt på sin ursprungsposition så att spelaren kan plocka upp det igen. Items skall inte kunna bäras med från en nivå till nästa.
10	Ett item som fungerar som en "eldkastare", som låter en karaktär tända eld på rutan framför karaktären, vilket skulle förstöra "spindelnät" som kan vara placerat där, och som skulle ge spelaren ett ”game over” om eldkastaren används på en annan spelarkaraktär.
11	Implementera monster I spelet som patrullerar runt I labyrinten enligt fasta mönster. Om ett sådant monster rör en karaktär får spelaren ett "game over", precis som lavarutorna. Dessa monster skulle röra sig ett steg varje gång spelaren

	flyttar en karaktär
12	Implementera liknande monster som även kan reagera på att en karaktär finns I deras synfält (rakt framför dem), och röra sig framåt mot karaktären. Om karaktären lämnar synfältet skulle monstret gå dit karaktären senast sågs, kolla runt efter karaktären (snurra ett varv), och fortsätta jaga om karaktären syns, annars kommer monstret återvända till sin originalposition. Dessa monster behöver inte kunna patrullera runt, utan kan istället stå still och övervaka ett område.
13	Gör så att eldkastaren som tidigare implementerades kan träffa ett monster för att applicera en slags "stun"-effekt, vilket skulle hindra monstret från att röra sig en kort stund, ungefär som om monstret rört ett "spindelnät".
14	Gör så att vissa nivåer är "mörka", vilket innebär att spelaren inte skulle kunna se mycket mer än det som finns runt karaktärerna i två små cirklar. Gärna skulle det även finnas en mjuk övergång mellan ljus och mörker. Gör så att eldkastaren tillfälligt skulle lysa upp hela rummet vid användning.
15	Få karaktärerna att mjukt "gå" från ruta till ruta istället för att plötsligt teleportera en ruta framåt när en tangent trycks ned. Gör samma sak för monster. Detta steg kan förskjutas om vi inte än har fått nödvändig grafik, då vi kommer be bekanta om hjälp med grafikarbete.
16	Få monstren att röra sig i realtid, så att de alltid rör sig i en konstant hastighet, istället för att spelaren styr när de rör på sig. Få de stillastående monstren att röra sig extra fort när de får syn på en karaktär.
17	Skapa en huvudmeny som startas när spelet öppnas, varifrån spelaren just nu enbart kan börja spela från första nivån, eller stänga ned spelet.
18	Låt spelaren spara sina framsteg i spelet, och sedan ladda samma nivå som hon avslutade på (I början av nivån). Det skall gå att ladda en nivå från huvudmenyn.
19	Skriv ett program som lätt låter en användare skapa nya nivåer genom ett grafiskt interface. Dessa nivåer skulle sedan sparas till en fil, och därifrån kunna användas av själva spelet. Gärna skall det även gå att testa sin nivå innan den sparas ned i filen. Denna "map maker" skall kunna öppnas från spelets huvudmeny, och nivåerna användaren själv skapar skall inte läggas till i samma lista som de befintliga nivåerna som spelet skeppas med, utan de skall vara tillgängliga från huvudmenyn under "custom maps". Detta steg kan ta mycket tid, och om vi är osäkra ifall vi hinner när vi kommer hit så kommer vi dela upp det i mindre steg, och göra det vi kan
20	Vi kanske kommer hitta på nya bra idéer under utvecklingens gång, men om vi når slutet av denna lista och har tid kvar, och känner att spelet inte kommer förbättras av nya implementationer, så kommer vi spara undan en kopia av det vi har, och sedan bygga på med en massa galenskaper som troligtvis inte kommer göra spelet bättre, men som låter oss lära oss nya saker som vi inte har utforskat under utvecklingen. Troligtvis kommer vi konsultera vår handledare.

4. Övriga implementationsförberedelser

Vi kommer använda en klass som hanterar själva spelbrädet och dess innehåll, men vi vill att andra klasser, exempelvis monsterklassen, själva ska kunna styra sitt eget rörelsebeteende, så brädet kommer inte ha regler för förflyttning av sitt innehåll, utan snarare enbart funktioner som tar ett objekt på planen och en riktning, och utför förflyttningen. Själva visningen av spelplanen kommer nog inte vara mycket olik den från Tetris, men vi kommer inte enbart använda kvadrater, utan kommer även ha bilder på exempelvis karaktärer i våra kvadrater.

5. Utveckling och samarbete

- Hur tänker ni jobba?

  Till stor del enskilt, men med flera möten varje vecka, där vi diskuterar kod och arbetar tillsammans.

- Ska någon skriva vissa specifika delar av koden, och hur ser ni i så fall till att **alla gruppmedlemmar** fortfarande har **full förståelse** för all kod i projektet (vilket är ett krav)?

  Just nu finns ingen plan för vem som skriver vad, men det kommer säkert finnas tillfällen då en av oss säger ”jag gör gärna detta”, och då får vi lägga extra fokus på kommunikation och kommentering, och vi kommer även förespråka att den andra personen går igenom koden och försöker göra förbättringar, exempelvis tydliga kontrakt på olika funktioner mm.

- Jobbar ni kvällar? Helger? Eller hur får ni plats i **schemat** för alla de timmar som projektet kräver utöver det schemalagda?

  Daniel vaknar tidigt, och kommer jobba mycket innan han går till campus. Simon kommer jobba någon gång mellan 5-9 på kvällen. Under schemalagd tid samt passande håltimmar kommer vi båda sitta tillsammans på campus och jobba, samt gå igenom kod vi har skrivit separat.

- Vad har ni för **betygsambitioner**? Har ni samma ambitionsnivå med projektet?

  Daniel siktar på betyg 5, Simon satsar på 5, men är ok med 4.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

Milstolpe 1 och 2 är implementerade till fullo.

Milstolpe 3 är delvis implementerad, då inget ”you win meddelande visas.”

Milstolpe 4 – 11 är fullständigt implementerade.

Milstolpe 12 är delvis implementerad, då skelett inte snurrar runt ett varv innan de vänder sig om för att gå hem igen.

Milstolpe 13 är fullständigt implementerad.

Milstolpe 14 är delvis implementerad, då det inte finns någon mjuk övergång mellan ljus och mörker, och eldkastaren inte påverkar mörkret.

Milstolpe 15 är helt implementerad.

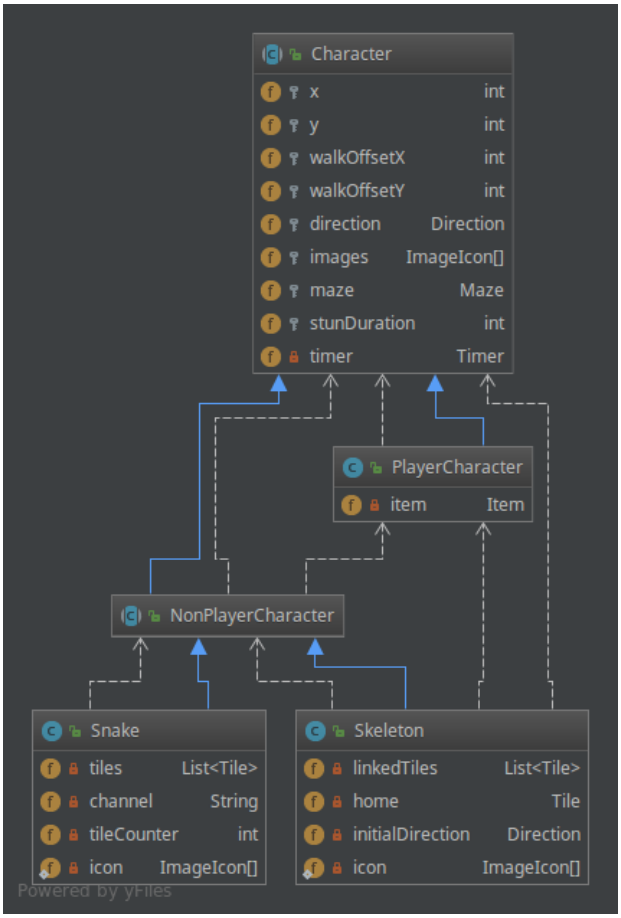
Milstolpe 16 är delvis implementerad, då skelett inte går fortare än någon annan karaktär.

6.2. Dokumentation för programkod, inklusive UML-diagram

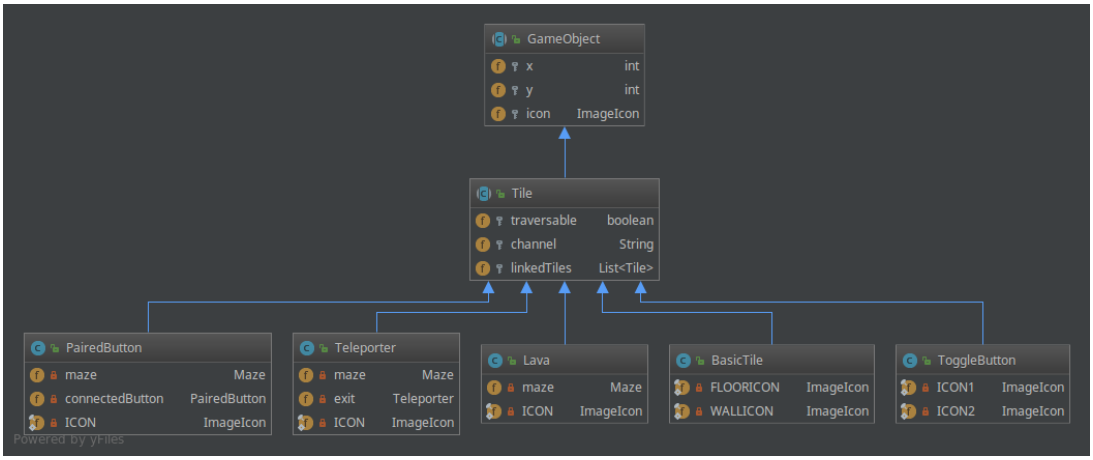
- **Övergripande programstruktur:**
  Vi börjar med att skapa ett Maze objekt jämsides ett MazeFrame-objekt genom Launcher-klassens 'main' metod. MazeFrame-objektet skapar ett MazeComponent-objekt som sköter sig självt därifrån, då det håller sin egen Swing Timer för att updatera sig själv med information som hämtas från Maze-objektet. Maze-objektet kommer I sin tur ladda in information om den första nivån genom att skapa ett Level-objekt, vilket läser data ifrån en fil. Efter detta börjar själva spelet, och spelaren kan kontrollera karaktärerna genom lyssnare I MazeComponent-objektet. Spelarkaraktärerna rör sig vid tangentbordsinput, medan andra karaktärer (av klass Snake eller skeleton) rör sig regelbundet med en Swing Timer mobTimer I Maze. Karaktärer kollar själva efter kollisioner efter att de rört sig ett steg, och de kan kollidera men olika rutor (Tile), föremål, (Entity), eller andra karaktärer (Character).
- **Angående spindelnät:**
  Om ni tar en titt på vår Cobweb-klass, kanske ni finner det lustigt att den implementerar gränssnittet Entity istället för att den ärver ifrån Tile. Cobweb är nämnligen den enda klassen som implementerar Entity förutom superklassen Item, så att ifall spindelnät istället behandlades som en Tile skulle gränssnittet Entity inte behöva existera över huvud taget, då vi skulle kunna använda Item istället. Detta kanske verkar smidigare, då Cobweb lätt skulle kunna vara en Tile funktionalitetsmässigt, men det finns ett problem, nämligen hur den ritas ut på skärmen. Skillnaden mellan en Entity och en Tile vad gäller utritning är att en Tile ritas ut först av allt, medan en Entity ritas ut senare, och däremellan ritas Character-objekt ut. Vad detta medför är att en Tile ligger alltid ”under” en Character, medan en Entity ligger ”ovanpå” en Character. Detta resulterar I att ett Cobweb-objekt, som är

en Entity, ritas ut ”ovanpå” Character-objekt, så att det syns tydligt att karaktären är fast I spindelnätet. Det passade att även låta Item-superklassen implementera Entity, då de är tillräckligt lika för att behandlas på samma sätt.

- **Översikter över relaterade klasser** och hur de hänger ihop.



Character är en abstrakt superklass som håller den huvudsakliga datan och funktionaliteten en karaktär behöver. Den har en position (x, y koordinater) och en ImageIcon för att ritas på skärmen. Den har funktionalitet för att försöka gå åt en riktning, och den har koll på om den av någon anledning inte kan gå någonstans (exempelvis ifall den rört ett Cobweb-objekt), och den kan hantera kollisioner. PlayerCharacter ärver av den, och den har lite mer funktionalitet, huvudsakligen för att hantera Item-object, något som NonPlayerCharacter-objekt helt ignorerar. NonPlayercharacter har lite mer utökningar, eftersom objekt av den typen själva måste hålla koll på hur de ska agera, dvs. vart de ska röra sig. Den har därför metoden moveTo(), som tar in ett Tile-objekt, och rör sig mot den. Playercharacter behöver inte denna funktionalitet eftersom spelaren styr karaktärer av den typen personligen. Snake håller funktionalitet för att välja en Tile och gå mot den med hjälp av moveTo(), efter vilket den kommer välja en ny Tile, medan Skeleton är lite mer anvancerad, och kollar efter PlayerCharacter-objekt. När den ser ett sådant kommer den lågga Tile-objektet den stod på på minnet, och rör sig dit. Ser den inte ett PlayerCharacter-objekt efter det kommer den återvända ”hem” och vända sig rätt igen.



GameObjekt är en abstrakt superklass som enbart håller I koordinater och en ImageIcon, vars enda syfte är att spara in på kod. Tile är mer avancerad, då den håller nödvändig data och metoder som dess underklasser kan behöva. Tile innehåller två viktiga abstrakta metoder onStep() och activate(). OnStep() kallas när en karaktär går på en Tile. BasicTile har en tom sådan metod, då den inte reagerar på spelaren, medan de andra underklasserna alla har olika funktionalitet I metoden. activate() är en metod som kallas av andra Tile-objekt. Exempelvis kommer ToggleButton I sin onStep()-metod kalla på activate()-metoden hos alla Tile-objekt I sin lista linkedTiles. Om ett BasicTile-objekt aktiveras på dessa sätt kommer det växla mellan vägg och mark.

6.3. Användning av fritt material

Vi har inte använt några klasser utanför de som finns inom Java 8.

Vi har direkt tagit en rad kod ifrån internet. Kodsnutten återfinns på rad 98 I Level.java, och är tagen ifrån StackOverflow. `https://stackoverflow.com/questions/10043209/convert-arraylist-into-2d-array-containing-varying-lengths-of-arrays`

6.4. Användning av objektorientering

Numrerade svar:

1. **Objekt/klasser:**
   Vi har självklart objekt överallt I vårt spel. Alla rutor en karaktär kan stå eller inte stå på är objekt, karaktärerna I sig är objekt, föremål de kan plocka upp är objekt, spindelnät de kan fastna I är objekt, spelbrädet allt ligger på (Maze) är ett objekt, och fönstret som visar upp spelet är ett objekt. Alla dessa har sin egen data och funktionalitet, så att exempelvis en karaktär själv vet vad och var den är, och hur den skall bete sig. Utan objekt skulle abstrakta pseudoobjekt kunna användas istället, och de skulle exempelvis kunna bestå av en slags lista som höll alla fält, tillsammans med exempelvis en sträng som berättade vilken typ av pseudoobjekt det var, men att få in funktionalitet I dessa vore osmidigt, så de skulle bara innehålla data, medan funktionaliteten skulle ligga öppet I olika filer, och behöva ta in ett pseudoobjekt som indata, samt kontrollera att det är samma objekt som hör till den metoden. Denna metod är på många sätt inte olik objektorientering, men är krångligare att implementera, och saknar samma stöd av de flesta arbetsmiljöer.
2. **Konstruktorer:**
   Vi har så klart använt konstruktorer för att initialisera alla ovannämnda objekt, då vi inte bara ville ha våra klasser definierade, utan vi ville faktiskt ha ett eller flera användbara objekt av varje klass, vars fält innehöll värden och vars metoder kunde

anropas.
För att replikera detta I vår pseudoobjektorienterade lösning ovan, skulle pseudokonstruktorer kunna skrivas utan problem. De skulle vara metoder som gjorde nästan samma sak som våra nuvarande konstruktorer, men som skapade och returnerade ett pseudoobjekt.
Just denna del av pseudoobjekthanteringen skulle inte skilja sig tokmycket från det vi har skrivit, då pseudokonstruktorn och konstruktorn skulle användas nästan identiskt, och de skulle inte se särskilt olika ut heller, bortsett ifrån att pseudokonstruktorn skulle behöva en eller ett par extra rader kod för att faktiskt skapa ett pseudoobjekt medan konstruktorn automatiskt skapar sitt objekt.

3. **Typhierarkier:**
Vi använder typhierarkier här och var. Vid utritning av spelet på skärmen via MazeComponent ritar vi ut en lista på objekt av typen Tile, en lista på objekt av typen Character, och en lista på objekt av typen Entity, trots att dessa tre ofta håller I många olika typer av objekt. Vi använder typhierarkier på många andra ställen också, då Maze har en lista på objekt av typen Tile ”tiles”, utan att veta exakt vad det är för olika objekt I listan. Detsamma gäller en lista objekt av typen Entity ”entities” I Maze. Tack vare denna användning av typhierarkier slipper vi ha en lista för varje typ av objekt som nu ingår I exempelvis Tile.
I vårt exempel ovan för pseudoobjektorientering skulle det vara möjligt att skriva liknande typhierarkier, men det vore inte lika smidigt som vår lösning I java. Man skulle kunna lösa det genom att skapa ett nytt pseudoobjekt som enbart höll I ett annat pseudoobjekt (Jag kallar dessa överpseudoobjekt och underpseudoobjekt för tydlighetens skull), och det underpseudoobjektet skulle enbart kunna vara av en passande typ. Alltså skulle man kunna skriva överpseudoobjektet PseudoTile som I sin pseudokonstruktor tog in ett underpseudoobjekt, och kontrollerade att det underpseudoobjektet var av en passande typ. Detta överpseudoobjekt skulle därmed kunna innehålla ett av flera möjliga underpseudoobjekt, vilket är en slags typhierarki. Denna pseudoobjektorienterade lösning skulle vara någorlunda lik vår lösning, men det finns en fundamental skillnad I att I den pseudoobjektorienterade lösningen finns det inte en pekare till ett pseudoobjekt som kan vara av typ A eller B, utan det finns en pekare till ett pseudoobjekt som kan *hålla I* ett objekt av typ A eller B. Detta är inte ett stort problem, men jag fann det värt att peka ut. Överlag är det krångligare att både implementera och använda, så jag finner vår objektorienterade lösning vassare.

4. **Subtypspolymorfism:**
Vi har använt subtypspolymorfism exempelvis vid kollisionsdetektering. När en karaktär flyttar på sig kommer karaktären kalla på onStep() metoden hos den Tile den nu står på. Alla subtyper till Tile har sin egen onStep() metod som gör olika saker. Detta resulterar I att när spelaren ställer sig på en Lava-ruta får hon ett ”game over”, och när spelaren rör en BasicTile (golv) så händer ingenting, trots att karaktären kallade på onStep() I Tile båda gångerna. På liknande sett har vi använt subtypspolymorfism på många andra ställen, såsom att Maze I sitt Timer objekt ”mobTimer” kallar på onTick() hos alla nonPlayerCharacter objekt I en lista med sådana, vilket genom subtypspolymorfism resulterar I att Snake objekt och skeleton objekt beter sig helt olika.
Om vi återgår till vårt pseudoobjektorienterade exempel så skulle vi exempelvis behöva skriva en metod anyTileOnStep() som tog in ett överpseudoobjekt som beskrevs ovan och som liknade Tile från vår kod, och anyTileOnStep() skulle då ta reda på vad det var för underpseudoobjekt som låg I överpseudoobjektet, och sedan kalla på lämplig metod tillhörande just det underpseudoobjektet.

Jämförelsevis tror jag inte att det finns någon större skillnad mellan de två lösningarna, då jag antar att även I Java måste rätt subtyp identifieras för att rätt metod ska anropas, skillnaden är att I Java sker detta per automatik, medan detta måste implementeras manuellt I den pseudoobjektorienterade lösningen, vilket medför att vår lösning I Java är smidigare att skriva.

6.5. Motiverade designbeslut med alternativ

Numrerade svar:

1. **Entity gränssnittet:**
   enligt vad jag tog upp tidigare ville vi rita ut Cobweb-objekt efter Character-objekt, vilket vi gjorde med hjälp av Entity-gränssnittet, då MazeComponent först ritar ut objekt av typen Tile, följt av objekt av typen Character, och sedan object av typen Entity. Anledning till att detta blev ett designbeslut var att Cobweb egentligen fungerar mycket som en Tile, så det hade kanske varit mer smidigt att låta Cobweb vara en Tile, så att vi inte behövde skapa ett helt gränssnitt Entity bara för Cobweb (Även Item implemeterar Entity, men utan Cobweb hade vi kunnat bytt ut pekare till Entity-objekt till pekare till Item-objekt). För att fortfarande få Cobweb att ligga ovanpå Character-object utritningsmässigt skulle vi kunna skriva lite utökningar av MazeComponent, men att göra detta snyggt och smidigt vore inte lätt, eftersom MazeComponent inte gärna vill ta reda på vilken subklass till Tile den behandlar, så vi skulle kanske behöva be karaktären rita ut ett spindelnät ovanpå sig själv när den står på ett Tile av typen Cobweb. Detta vore möjligt, men mindre smidigt än vår lösning, då Character-klassen måste hålla koll på ifall den står på ett spindelnät, och den måste hålla I en bild på spindelnätet, som ritas ut två gånger, en under karaktären, och en över karaktären. Meriten med vår lösning är att saker håller sig mer abstrakta. Vi kollar inte specifikt efter spindelnät, utan vi behandlar dem likadant som Items.
2. För att initialisera vår ImageIcon-objekt skrev vi en klass IconMaker, som innehåller statiska metoder för att skapa och returnera ImageIcon-objekt av rätt storlek. Vi gjorde detta för att få bort duplicerad kod, men det hade varit ett rimligt alternativ att bara kalla på lämpliga konstruktorer där de behövdes, istället för att gå igen denna extraklass. En merit med att inte använda vår lösning skulle vara att det vore lättare att bestämma storleken på en ImageIcon, då vår IconMaker-klass bara kan skapa ImageIcon-objekt av två storlekar, men de två råkar vara de enda vi använder, och därmed finner jag vår lösning fullt lämplig.
3. Vi har flera olika listor på karaktärer I Maze, nämligen en för Character-objekt, som håller alla karaktärer, en för PlayerCharacter-object, som håller alla spelarkaraktärer, och en som håller nonPlayerCharacter-object. Detta kan verka osmidigt, då vi har två pekare till varje karaktär, men vi behövde kunna kalla på vissa metoder som var specifika till just PlayerCharacter vid vissa tillfällen, såsom när en tangent tryckts ned, och vissa metoder som var specifika till NonPlayerCharacter vid andra tillfällen, såsom I Timer-objektet mobtimer, som kallade på onTick()-metoden, något som PlayerCharacter varken har eller behöver. Vi behövde även en lista på alla karaktärer så att exempelvis ett Cobweb-objekt skulle kunna hitta en karaktär som vidrört det, oavsett vilken karaktär det råkar vara. Vi skulle kunna ha enbart använt en lista med alla karaktärer, och skrivit mer avancerade getter-metoder för att hämta ut enbart exempelvis PlayerCharacter-objekt från den listan när de behövdes, eller så skulle vi kunna ha skrivit fler abstrakta metoder I Character-superklassen, så att exempelvis PlayerCharacter måste ha en onTick(), vilket skulle betyda att mobTimer skulle kunna ta en lista på alla karaktärer, och att implementationen av onTick()-metoden I

Playercharacter skulle lämnas tom, så att dessa objekt oavsett ignorerade anropet. Vi föredrar vår lösning framför båda dessa, då det är mindre krångligt att helt enkelt skapa nya pekare än att skriva om ett gäng metoder. Det blir fler fält I Maze-klassen, men det anser inte vi vara ett problem.

4. För att få våra Snake-objekt att patrullera enligt ett fast mönster gave vi den pekare till olika Tile-objekt som den skulle gå till, och gav den funktionalitet för att hitta dit, även om detta objekt var långt bort. Ett alternativ vi betänkte var att ge den pekare till många fler Tile-objekt, så att den alltid skulle ha en sådan Tile bredvid sig, och på så sätt gå ett steg åt taget. Meriten med vår lösning är att vi använder färre pekare till att ofta uppnå samma sak, samt att textfilerna vi skriver våra nivåer I blir mycket renare. Utöver det appliceras vår kod för att gå till en viss Tile (metoden moveTo() I NonPlayercharacter) även används av Skeleton-klassen, vilket sparar in på kod. En merit med den alternativa lösning skulle dock vara att vi hade kunnat bestämma I mycket högre grad exakt vart en Snake skulle patrullera, då vi just nu nästan är begränsade till att flytta den fram och tillbaks mellan två Tile-objekt. Vi var inte tokintresserade I detaljerade patrulleringsrutter, så vi föredrar stort vår nuvarande lösning.
5. Vi skrev ihop Tile-objekt av typen vägg och golv till ett objekt, BasicTile, eftersom de två egentligen inte skiljer sig vad gäller funktionalitet, utan de skiljer sig enbart vad gäller fält. För att få till denna lösning var vi dock tvungna att ge objektet två olika Imageicon-objekt, vilket krånglade till saker lite mer än andra Tile-klasser (detsamma gäller ToggleButton, men den initialiseras alltid med samma ImageIcon, så det blev inga svårigheter med att kalla på superklassens konstruktor). Ett alternativ skulle vara att helt enkelt skriva två olika objekt, vilket skulle vara lite smidigare att arbeta med, men det hade varit dålig praxis att skriva två för lika klasser.

7. Användarmanual

Mirrorinth Användarmanual

Spelkoncept:

Mirrorinth är ett spel där målet är att förena två karaktärer som är fast i en labyrint. Haken är att spelaren styr båda karaktärerna, samtidigt, med samma tangenter. Exempelvis då piltangent uppåt trycks ned, kommer den första karaktären att röra sig ett steg uppåt, medan den andra kommer försöka röra sig ett steg nedåt. Inom de olika nivåerna finns flera hjälpredor som kan vara till nytta för att förena karaktärerna, men det finns också flera faror som bör undvikas.

Att starta spelet:

För att starta spelet, kör Launcher-klassens Main-metod. Du kastas då rakt in I första nivån,

och kan börja spela.

Spelkontroller:

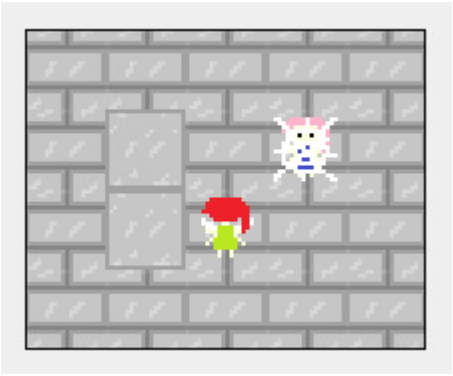
Använd de fyra piltangenterna för att förflytta de två spelarkaraktärerna.
Använd mellanslagstangenten för att använda eventuella ”items” du kan ha plockat upp.

Spelmekanik:

Vid förflyttning kommer den första karaktären försöka gå åt den riktning som trycktes ned, och enbart ifall den första karaktären lyckades förflytta sig, kommer den andra karaktären att försöka förflytta sig åt motsatt riktning. Detta innebär att om den första karaktären är fast, så kan ingen karaktär röra sig, men om den andra karaktären är fast, så kan den första karaktären röra sig ensam.
Det finns flera faror I spelet som ger spelaren ett ”game over”; detta innebär att spelaren måste börja om från början av den nuvarande nivån.

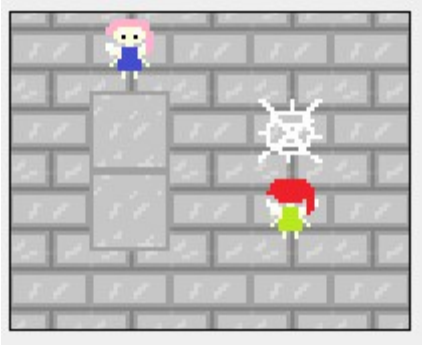
Olika rutor, ”tiles”:

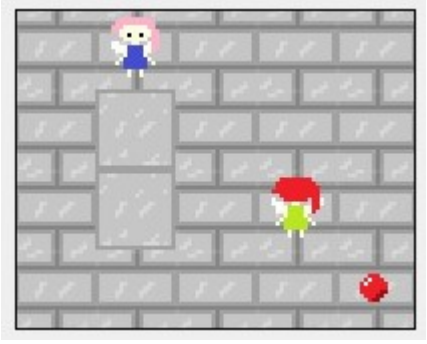
I spelet finns det golv, som karaktärer kan gå på som vanligt, väggar, som karaktärer inte kan gå på, lava, som är farligt och ger spelaren ett ”game over” ifall en karaktär går på det, spakar, som karaktärer kan vidröra för att ”dra I spaken”, vilket förändrar nivån lite grann, på det vis att vissa golvrutor blir väggrutor och/eller vice versa, detta kan ske flera gånger då spaken inte försvinner när den används, liknande knappar som kommer I par, som bara kan aktiveras en gång, och som bara aktiveras när två karaktärer står på varsin knapp, samt en teleportör-ruta som kommer I par och kan teleportera karaktärer emellan sig. Ovanpå en ruta kan det även ligga ett spindelnät, som, vid vidrörelse, kommer göra så att den vidrörande karaktären fastnar, och måste kämpa för att komma loss; det tar 6 rörelser för karaktären att bryta sig fri.



Föremål, ”items”:

I spelet finns det två ”items” som kan plockas upp av spelarkaraktärerna. Dessa hålls av karaktärerna tills de används (eller tills en ny nivå laddas in). Efter att ett ”item” har använts kommer karaktären inte längre hålla I det, utan det kommer återvända till den plats I nivån där det först fanns, så att det kan plockas upp på nytt.
Det första föremålet är eld, vilket, vid användning, påverkar rutan framför karaktären. Ifall det på denna ruta finns ett spindelnät, så kommer det försvinna, om det finns en annan spelarkaraktär, så kommer spelaren får ett ”game over”, och om det finns ett ”monster”, så kommer monstret drabbas av en ”stun”-effekt, och bli stillastående några sekunder, ungefär som om denna hade gått in I ett spindelnät.

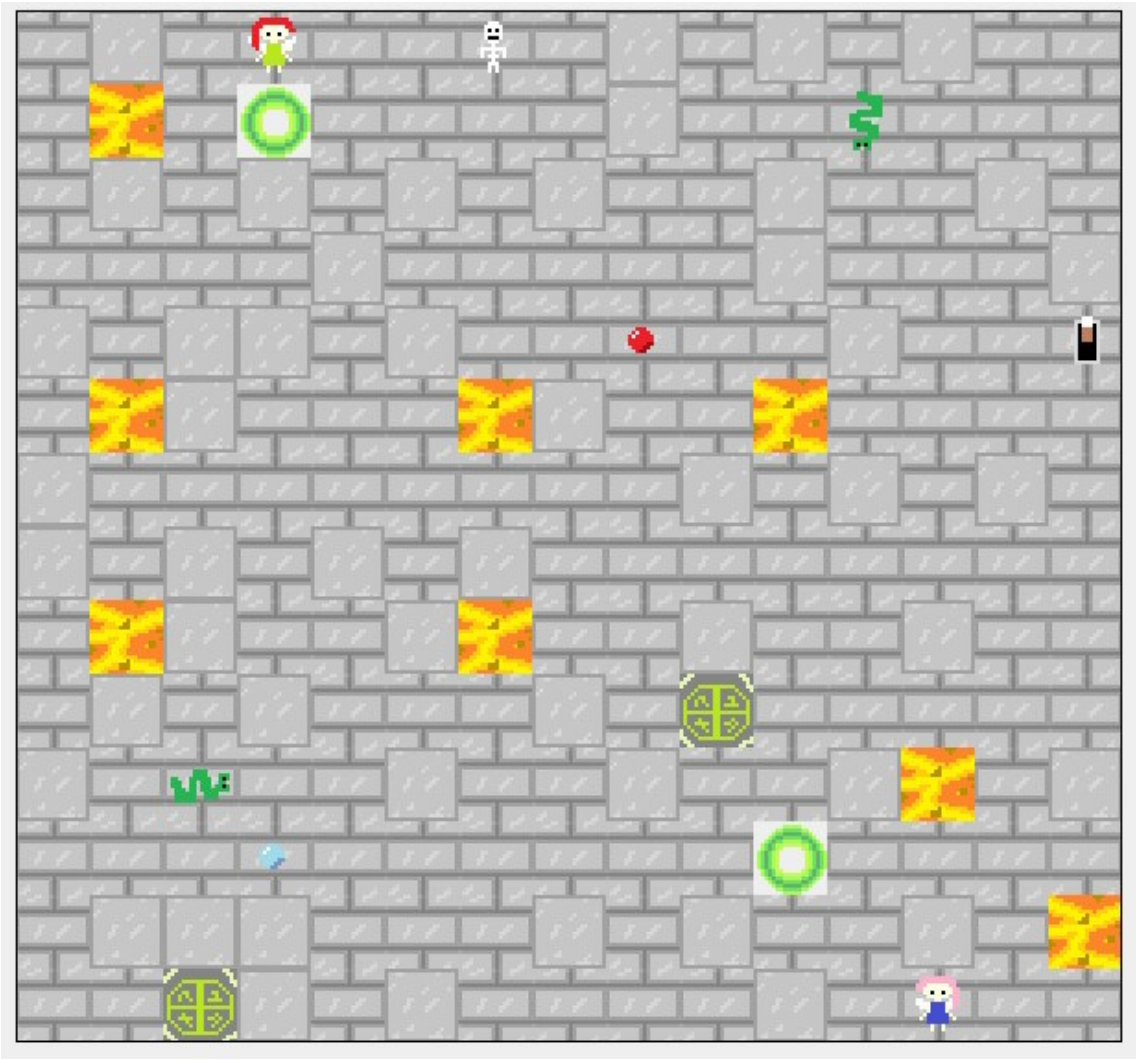




Det andra föremålet gör vid användning så att karaktären som håller det hoppar framåt två rutor, vilket innebär att karaktären exempelvis kan ta sig igenom en vägg, eller något farligt.

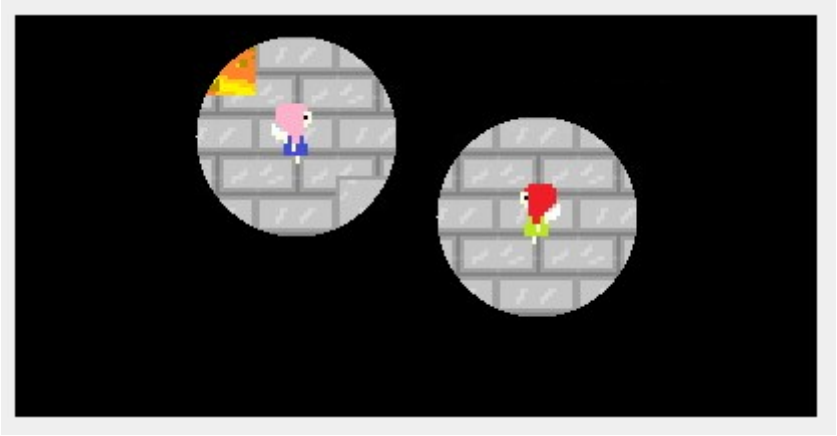
Fiender ”monster”:

I spelet finns det två typer av ”monster”, som vid vidrörelse av en spelarkaraktär, ger spelaren ett ”game over”.
Den ena fiendetypen är en orm, som patrullerar ett område enligt ett fast mönster.
Den andra fiendetypen är ett skelett, som står still tills den ”ser” en spelarkaraktär (karaktären står framför skelettet), då det kommer börja gå mot karaktären. Ifall karaktären försvinner ur skelettets synfält kommet det återvända till sin startposition.



Mörker ”fog of war”:

I vissa av spelets nivåer kommer spelarens synfält vara begränsat till två cirklar, centrerade runt de två spelarkaraktärerna; resten av nivån kommer vara täckt av mörker.



8. Slutgiltiga betygsambitioner

Vi hann inte med allt vi ville och därmed kan vi för tillfället inte sikta på något högre än en fyra. Vi är beredda att komplettera upp detta vid senare skede dock.

9. 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?*
    - Att arbeta tillsammans och utveckla ny funktionalitet gick bra, då det är mycket motiverande aspekter av programutveckling. Det som gick dåligt var dokumentation och städning, då vi sköt upp allt för mycket utav det till slutet.
- *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!*
    - *Vi har varit på föreläsningarna och de handledda labbarna. Vi har även kollat upp mycket på nätet, huvudsakligen på forum, officiella sidor, och kurshemsidans vanliga lösningar.*
- *Har ni lagt ned för mycket/lite tid?*
    - Några timmar för lite.
- *Var arbetsfördelningen jämn? Om inte: Vad hade ni kunnat göra för att förbättra den?*
    - Daniel arbetade fler timmar än Simon. Simon tappade mycket på sin dåliga dygnsrytm, då han var trött mycket.
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
    - Milstolpar var mycket användbart.
- *Har arbetet fungerat som ni tänkt er? Har ni följt "arbetsmetodiken"? Något som skiljer sig? Till det bättre? Till det sämre?*
    - Det har gått som planerat för det mesta, bortsett från att dokumentation inte hände.
- *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.*
    - Att det var så mycket jobb på så kort tid. Vi började visserligen lite sent, men att jobba I 80 timmar på mindre än tre veckor när vi även har andra kurser är mycket.
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*

    - Håll bra koll på vad som behövs för att få betyget ni vill ha tidigt I projektet så att ni kan planera er tid därefter. Läs igenom betygskraven ordentligt, och lägg hellre er feature freeze en dag tidigt än en dag sent.
- *Har ni saknat något i kursen som hade underlättat projektet?*
    - Mycket arbete skedde under tentaperioden, så det hade varit fint att ha träffat handledare under den tiden.
- *Har ni saknat något i kursen som hade underlättat er egen inlärning?*
    - ...