Projektbeskrivning

<OpenHeroes>

2018-03-22

Projektmedlemmar:
Jens Lindgren <jenli669@student.liu.se>

Handledare:
Teodor Riddarhaage <teori199@student.liu.se>

Table of Contents

Planering

1. Introduktion till projektet

OpenHeroes är en imitation av Heroes-spelen som utvecklades av New World Computing och publicerades av The 3DO Company. I spelet styr man ett kungarike, främst deras hjältar och soldater men man utvecklar och erövrar även städer och gruvor med mera på spelkartan.

Målet med spelet är att besegra alla motståndare vilket görs genom att erövra deras städer. Man kan beskriva spelet som ett avancerat brädspel där varje spelare turas om att spela. Under en tur i spelet kan man till exempel uppgradera sina stadsbyggnader eller skicka sina hjältar med deras tillhörande armèer för att slåss med neutrala armèer, andra hjältar, ta över gruvor samt erövra städer.

Olika hjältar har naturliga talanger som ger dem speciella förmågor, och dessutom får de erfarenhet under sina strider och erövringar som gör att de kan utveckla nya färdigheter och talanger. Sist men inte minst kan de hitta magiska artefakter som kan ge unika fördelar i strid.

2. Ytterligare bakgrundsinformation

Spelet styrs med mus och tangentbord.

I spelvärlden finns det flera olika sorters kungariken, t. ex. människoriket, alvriket, trollkarlsriket m.m. Varje rike har en egen uppsättning av sex olika sorters soldatenheter som har olika förmågor. Trots skillnaderna skall varje ”nivå” av soldat ungefär motsvara samma övergripande styrka som de andra soldaterna på samma nivå i de övriga rikestyperna.

Spelarens stad kan uppgraderas med guld samt resurser som hämtas från gruvor på spelplanen. Genom att bygga nya byggnader eller uppgradera existerande byggnader får spelaren möjlighet att rekrytera respektive uppgradera sina soldater.

Huvudspelplanen representeras av ett rutnät. Varje spelare börjar med en stad, en hjälte och en liten grupp svagare soldater. Omkring städerna finns det olika resurser och särskilda platser med unik funktionalitet som t.ex. stall där man kan få extra gångavstånd per tur.

När hjältar möts på spelplanen går de i strid med varandra på ett hexagonalt spelfält. Striden utspelas i en egen stridsvy där man kontrollerar de enskilda soldaterna. Hjältarna går aldrig själva ut på slagfältet, men kan påverka soldaterna med sina färdigheter samt använda magi för att skada fientliga soldater eller påverka slagfältet. En strid är över när en hjälte flyr eller förlorat alla sina soldater.

Inspirationen till OpenHeroes är främst Heroes 3 vars manual hittas här:

https://heroes3wog.net/download/Heroes%203%20Restoration%20of%20Erathia%20Manual.pdf

3. Milstolpar

Projektet skall följa nedanstående milstolpar i så god utsträckning som möjligt.

#	Beskrivning
1	Spelarkitekturen är modellerad som server/klient. Spelservern kör i en egen tråd och har auktoritär kontroll över speltillståndet. Kommunikation mellan servertråden och klienttråden(/ar) sker genom TCP.
2	Ett fönster med den primära spelkartan ritas upp. Det går att urskilja vilka spelrutor som innehåller objekt, t. ex. städer, gruvor, hjältar eller terräng, även om dessa inte konkret existerar som fullt utvecklade spelobjekt ännu.
3	En spelarstyrd hjälte finns på kartan som kan gå omkring på kartan genom att man klickar två gånger på den ruta man vill att hjälten skall gå till. Hjältens navigation använder en vägsökningsalgoritm för att hitta vägar runt upptagna rutor. Kartan kan zoomas in och ut, och man kan ”scrolla” omkring på kartan.
4	Spelturer implementeras. Hjälten kan bara gå ett visst antal steg per tur. En förhandsvisning för hjältens rutt ritas ut när man klickar på en ruta första gången. En tur avslutas genom att man trycker på en knapp. Flera lag kan existera och ta sina turer i en ordning som slumpas vid spelets start.
5	Varje lag tilldelas sin egen spelvy. Spelvyerna håller reda på vilka rutor varje lag har utforskat. Innehållet av outforskade spelrutor är gömda för det laget. Detta tillåter ”hot seat” flerspelarläge.
6	Flera hjältar skall kunna kontrolleras av samma lag. Hjälten man vill kontrollera väljs genom att klicka på den.
7	Alla objekt som hjältar kan interagera med skall ha ett standardiserat system för detta. Om en hjälte från ett lag går in i en hjälte från ett annat lag så slåss de genom att slumpa varsitt tal från 1-100 där högst vinner. Om hjältar från samma lag går in i varandra så skapas en popup med texten ”Hej!”
8	Varje spelarlag skall kontrollera en stad vid spelets start. När en hjälte interagerar med en fientlig stad skall den byta ägare till hjältens lag. När en spelare förlorat alla sina städer har den förlorat. Det skall även finnas neutrala städer och gruvor som tas över på samma sätt.
9	Spelarlag kan tjäna resurser från sina städer och gruvor. Vid början av varje spelrunda tilldelar städer och gruvor resurser till spelaren som kontrollerar dom.
10	Ett rudimentärt spelgränssnitt existerar. Man kan se en lista av ikoner till höger för varje hjälte och stad man kontrollerar. Bredvid varje hjältes ikon finns en mätare som visar återstående steg för hjälten denna spelomgång. Det går även att se hur mycket av varje resurs man har samt sin inkomst för varje resurs per spelrunda. En liten pixelerad övergripande karta kan ses uppe till höger.
11	Ett grundläggande ljudsystem finns för att spela upp ljudeffekter och musik.
12	Stridsfältet ritas upp i grundläggande form när två hjältar möts i strid. Varje hjälte kontrollerar en enhet av soldater som kan gå omkring på slagfältet, vägsöka sig omkring hinder, och attackera motståndarens enheter om det är inom räckvidd. Slagets vinnare bedömer nu vilken hjälte som förlorar på huvudspelplanen, istället för det tidigare tärningskastet.

13	Städer och hjältar kan innehålla enheter. Vid spelets början tilldelas hjältarna en delvis slumpad mängd soldater. När man kontrollerar en hjälte eller stad kan man se vilka enheter den har och hur många.
14	När hjältar möts i strid är det deras egna soldater som används i striden. Innan striden börjar kan båda hjältarna bestämma vart deras respektive enheter skall stå på sin sida av spelplanen.
15	När en stad attackeras sker det en strid om staden har en garnison av soldater. Staden erövras endast om stadens garnison besegras eller är obefintlig.
16	En rudimentär stadsvy kan öppnas för en stad genom att dubbelklicka på den.
17	Man kan bygga byggnader genom stadsvyn om man har de resurser som krävs. Det kan även finnas krav på att ha byggt vissa byggnader innan andra blir tillgängliga.
18	Enheter i en stads garnison är synliga i stadsvyn. Det finns även en tom rad under garnisonsraden för att visa besökande hjältars armèer senare.
19	Det finns barracker för olika sorters enheter tillgängliga att bygga i städerna. Om man klickar på en barrack så kan man rekrytera enheter för en viss resurskostnad. Dessa enheter läggs automatiskt till i stadens garnison.
20	Hjältar kan besöka en kontrollerad stad genom att gå till dess port. Detta öppnar automatiskt stadsvyn. När en hjälte besöker en stad är den och dess enheter synliga i stadsvyn.
21	När en hjälte besöker en stad kan man flytta enheter mellan stadens garnison och hjältens armè i stadsvyn. Det går att dela upp enheter i mindre grupper, eller kombinera två grupper av enheter om de är av samma typ.
22	Genom att dubbelklicka på en hjälte kan man öppna en infoskärm om hjälten. Hjältens armè är synlig på infoskärmen.
23	Om två hjältar från samma rike möts på spelplanen kan de byta enheter med varandra. Detta sker genom en egen skärm.
24	Städer kan bygga en byggnad som låter spelaren rekrytera nya hjältar. Hjältar kan befinna sig i en stads garnison, och syns då inte på kartan förrän de flyttas ut från garnisonen. Hjältar kan inte flyttas ut ur garnisonen om en hjälte redan befinner sig i staden.
25	Soldattyper blir mer detaljerade. De har egenskaper som skada, hälsa, moral, gångavstånd och unika förmågor, till exempel distansattacker, flygförmåga eller att de undviker vedergällningsslag. En enhets totala storlek är en faktor för dess totala skada.
26	Hjältar kan retirera från strid. De förlorar alla sina enheter och försvinner från spelplanen, men går omedelbart att rekryteras på nytt i städer som kan rekrytera hjältar.
27	Hjältar kan utrusta sig med katapulter med hjälp av en speciell stadsbyggnad. Hjältarnas infoskärm visar om hjälten har en katapult eller inte.

28	När en stad attackeras används ett unikt slagfält. Slagfältet har murar som kan forceras med hjälp av en katapult. Om en hjälte har en katapult genereras den som en enhet på hjältens sida av slagfältet, med en stor mängd hälsa. Den agerar som en distansattackerande enhet som inte kan gå, och bara kan sikta på murar.
29	Under en stadsstrid kan försvarande enheter gå ut ur murarna genom en port. På stadsmurarna finns det två torn som skjuter på fiendens enheter. Dessa kan attackeras av en katapult, men inte vanliga enheter.
30	Resurser kan placeras direkt på kartan, t.ex. guldkistor, som kan hämtas av hjältar och omedelbart blir tillgängliga för hjältens lag.
31	Varje typ av soldat kan existera som neutrala fiender på kartan. De måste besegras för att passera dem.
32	Samtliga sex enhetstyper för en stad är implementerade.
33	Hjältar har speciella förmågor som ger dem fördelar på slagfältet eller på spelkartan.
34	Hjältar kan äga speciella artefakter som ger dem fördelar eller nackdelar på slagfältet eller på spelkartan. Dessa är synliga på infoskärmen och kan tas av och på där. Hjältar kan också transportera artefakter. Endast burna artefakter påverkar hjälten.
35	Hjältar kan köpa magiböcker. Magier lärs ut i städer som har rätt byggnader om hjälten möter kraven för att lära sig de magier som finns tillgängliga.
36	Det går att spara hela speltillståndet, stänga ner spelet och sedan ladda det sparade speltillståndet korrekt.
37	Det finns en startmeny. Där kan man välja att starta ett nytt spel, ladda ett sparat spel eller avsluta.
38	Det går att välja mellan olika kartor.
39	När man startar ett nytt spel kan man välja att starta ett spel där andra klienter kan ansluta.
40	Det går att ladda ett spel där flera klienter varit anslutna. När man väljer att ladda spelet väntar servern på att alla spelarlag är styrda av en klient innan spelet börjar.
41	Det går att låta datorn styra ett lag. Datormotståndaren spelar hjälpligt.
42	Datormotståndaren försöker utforska kartan, skaffa fler resurser samt soldater, och försöker ta över motståndarlag.
43	Det går att låta en dator ta över lag som kontrollerats av människor när man laddar ett spel där flera människospelare tidigare styrt lag.
44	Mer typer av riken, hjältar, enheter, förmågor, artifakter, magier samt terrängobjekt.

4. Övriga implementationsförberedelser

Jag kommer behöva lägga relativt god tid i början av projektet på att strukturera min klient- och serverarkitektur. Det kommer även innebära lite jobb med att implementera och standardisera kommunikation via TCP mellan servern och klienten. Det tror jag dock att det kommer vara värt gentemot att först implementera spelet som ett strikt enhetligt program och sedan försöka dela upp det för att tillåta flera spelare. Jag känner mig ganska bekväm med de övriga koncepten i kursen så det känns givande för mig att försöka prova dessa nya koncept i det här projektet.

Att kommunicera med den egna datorn på det här sättet för singleplayer bör inte utgöra något problem då sådan kommunikation sker väldigt snabbt. Spelet kräver inte blixtsnabba reaktioner för att spela så det är också acceptabelt med en relativt hög latens när man spelar på en annan dator än den servertråden kör på.

Jag skall försöka implementera filhantering som ljud och grafik i en singleton för att minimera eventuell overhead med filhantering. Det kan kanske också vara lämpligt att tänka på hur och var de olika trådarna i programmet lagras, det verkar rimligt att även dessa kan tillhöra en singleton.

Projektet innehåller ganska många olika typer av objekt som är snarlika, och har snarlika egenskaper inuti sig, t.ex. städer, hjältar och enheter. Jag tror det finns stor potential och vinning med att se dessa som objekt i sig, och i att applicera polymorfism och ärvning vid implementation av dessa delar av spelet.

Spelkartan känns lämplig att implementera som en array. Jag tror också det kan vara lämpligt att låta kartobjekt såsom städer och gruvor ha interna representationer av sitt kartutseende som arrayer. Då kan en övergripande kartklass hämta objektens utseende på spelplanen och placera ut dem genom att bara ange deras ankarpunkter (övre vänstra hörnet av objektet t. ex.).

Stridsfältet kommer funktionellt att ha en hexagonal layout istället för ett rutnät. Jag tror även här att det är lämpligt att implementera detta med hjälp av en 2D-array, även om jag måste standardisera lite beteenden kring vilka ”rutor” som anses vara bredvid varandra.

Jag är osäker på hur tydligt server-/klientbeteendet kommer att definiera lagstyrningen, jag tänker mig att det kommer vara ganska tydligt från grunden att flera klienter (även om de är lokala) är involverade, men jag har satt milstolpar för att försäkra att den visionen blir verklighet.

Jag ser dock det som en ganska stor utmaning att bygga en AI som kan spela spelet på ett naturtroget/tillfredställande sätt, därför flyttar jag fram implementation av det till mycket senare, så att jag hinner fokusera på mer kursnära problem innan jag grottar ner mig i det.

5. Utveckling och samarbete

Jag kommer att jobba själv om ingen annan student känner sig ensam och vill ha en projektpartner. Projektet som det är definierat i detta dokument är kanske lite på den ambitiösa sidan, men jag gillar att programmera så för mig kommer det vara ett rent nöje. Jag tror att jag kommer hinna implementera majoriteten av milstolparna, men att ljud- och grafisk design kan bli lidande. Eftersom dessa inte är intressanta för kursen ser jag inte det som ett problem, men om jag får tid över kommer jag att försöka polera dessa aspekter lite grann.

Slutinlämning

6. Implementationsbeskrivning

Struktur

Spelet är grovt strukurerat i sex paket: connection, entity, gamelogic, gamemodel, resources och view. Dessa paket ger bara en generell idé om deras klassers funktion.

Utmaningar

OpenHeroes byggdes från grunden utifrån en Server-/Klientarkitektur. Det innebär att servern bestämmer hur speltillståndet ska påverkas. Om en klients speltillstånd skulle stå i strid med serverns speltillstånd måste klienten helt enkelt rätta sig.

Att spelet har flerspelarläge innebar en del ganska stora utmaningar. Spelet måste kommunicera med en annan dator på ett stabilt och säkert sätt. Jag valde att använda Javas egna implementation av nätverksdataströmmar, så kallade ”Sockets”. Sockets använder det beprövade TCP-protokollet som gör mycket internt för att säkerställa att meddelanden verkligen kommer fram.

Ett problem med nätverksapplikationer är att vi inte kan vara säkra på exakt när en annan dator bestämmer sig för att kommunicera, med andra ord kan vi bli skickade meddelanden asynkront. När vi väntar på ett meddelanden kan vi inte göra någonting annat, eftersom de kan komma när som helst. Men vi kan inte sitta och vänta på meddelanden för evigt – då händer det ju ingenting i spelet!

Det innebär att OpenHeroes förutom nätverksprotokoll även måste implementera trådning. I princip byter applikationen väldigt snabbt mellan att utföra olika sekvenser av kod. Genom att tilldela en tråd ansvaret att ta emot meddelanden kan resten av vårt program fortsätta jobba med annat tills ett meddelande dyker upp.

Trådhantering

OpenHeroes centraliserar sina trådar i en Singleton, GameResourceManager (GRM). Eftersom alla trådar startas centralt från GRM är det enkelt att i ett senare skede avsluta applikationens alla olika trådar på ett enkelt sätt.

GRM var inte den första lösningen som implementerades för trådningen. Från början byggde och underhöll jag en ”pyramid” av trådar och undertrådar som sedan skulle avslutas sekventiellt. Detta visade sig extremt svårt att genomföra i praktiken, därför skapades GRM.

När OpenHeroes vill delegera en process anropas således GRM-instansen och den tilldelas tråden i fråga. GRM innehåller en så kallad ThreadPool dit trådar man vill köra parallellt allokeras, till exempel TCPClient, TCPServer, ClientSession och ServerSession. Men man kan även tilldela GRM trådar som skall köras sekventiellt; det är användbart när man accepterar nya anslutningar då dessa måste accepteras en i taget.

När applikationen är klar med sin spelinstans anropas GRM som avslutar alla trådar som instantierats under spelets gång. Den enda tråd som återstår efter detta är applikationens huvudtråd som driver användargränssnittet.

Nätverkskommunikation

Principen i OpenHeroes nätverksimplementation är att en server öppnar så många anslutningar som den förväntar sig skall ansluta, och väntar sedan på att de ansluter. När samtliga har anslutit slutar servern att acceptera nya anslutningar och startar spelet, se ”Illustration 1”.

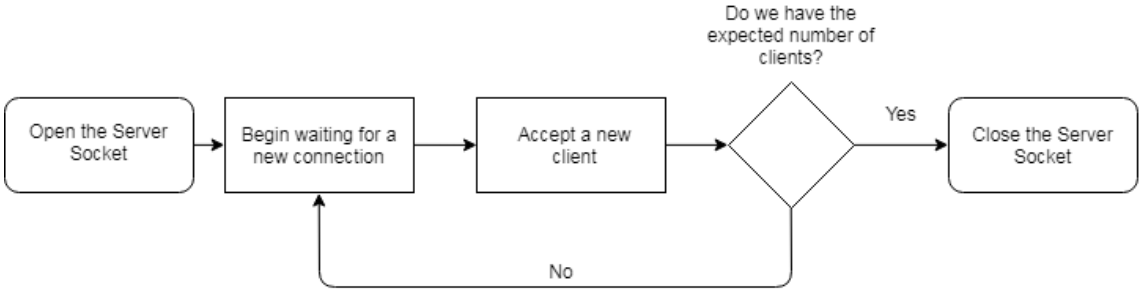

Illustration 1: Förenklad bild av anslutningsförloppet när ett spel startas.

Praktiskt i programmet innebär detta att en hel del olika saker händer. Spelet startar en servertråd som i sin tur skapar undertrådar för varje anslutning. När en klient ansluter skickas speltillståndet till klienten. Dessutom tilldelas klienten ett ID som den skall identifiera sig med hädanefter. Klientsidan skapar också en del trådar för att lyssna på servern, så all nätverkskommunikation är frikopplad från spelets processer.

GameResourcesManager utöker sitt ansvarsområde i samband med att anslutningar öppnas. Javas Sockets stänger inte sig själva automatiskt, utan måste avslutas på ett kontrollerat sätt. När Sockets öppnas görs det genom GRM med samma princip som för trådningen. Det innebär egentligen att GRM axlar två skilda ansvar, vilket inte generellt är önskvärt inom objektorienterad programmering. I detta fall anser jag dock att det är rimligt eftersom anslutningarna redan har en tät koppling till trådningen. Dessutom löser denna implementation ett problem som jag tänker belysa senare under koddokumentationen.

Server- och klientkoden har väldigt mycket kod gemensamt på alla olika nivåer av abstraktion, och därför ärver de båda från gemensamma superklasser. De djupt nätverksrelaterade TCPServer- och TCPClientklasserna ärver funktionalitet från superklassen MessageProtocol. MessageProtocol innehåller metoder som skickar meddelanden och data, och det är faktiskt MessageProtocol som tillhandahåller strömmen som faktiskt lyssnar efter meddelanden.

Ett normalförlopp vore t.ex. att spelet startar en ClientSession i en ny tråd, som i sin tur startar en TCPClient, också i en egen, ny tråd. När TCPClient konstrueras så anropas dess superkonstruktor MessageProtocol, som bygger en TCPListener. Även den tilldelas en egen tråd. På så sätt har tre trådar skapats; En som underhåller speltillståndet, en som skickar meddelanden och en som lyssnar på meddelanden från servern.

TCPServer särskiljer sig från TCPClient på några enstaka punkter, generellt kan man säga att där TCPServers metoder skickar information tar TCPClients metoder emot den.

Information som skickas mellan TCPClient och TCPServer stannar bara inom dom klasserna om informationen är relaterad till nätverkstillståndet eller dataöverföring. I annat fall förmedlas informationen vidare till ClientSession respektive ServerSession.

Spellogik

De mer spelnära ClientSession och ServerSession underhåller det lokala speltillståndet, och mycket av den funktionaliteten finns i deras gemensamma superklass GameHandler. GameHandler är den klass som främst påverkar speltillståndet. ServerSession behöver också påverka speltillståndet utöver vad GameHandler genomför. Detta beror på att GameHandler bara kan hantera deterministiska aspekter av spellogiken. När något slumpartat sker måste det ske på servern eftersom den är auktoritär. Om två klienter tilläts slumpa inom sina respektive speltillstånd skulle de med all säkerhet tappa synkronicitet inom kort.

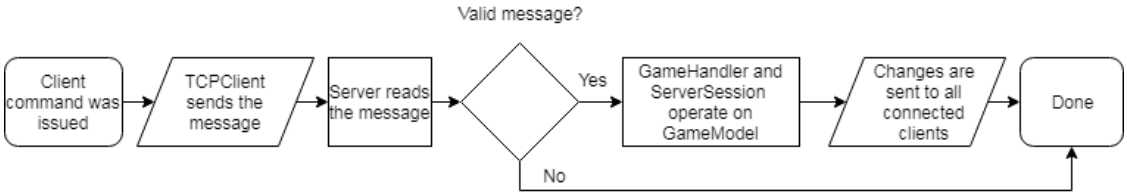

Illustration 2: Flödesdiagram som visar hur spellogiken drivs genom ServerSession.

Till exempel innebär detta att ServerSession ansvarar för att skapa nya Battlefield-objekt och propagera dom till sina klienter, se ”Illustration 2”. Hjältar har slumpvis valda namn så även de måste skickas från ServerSession till samtliga klienter, annars skulle speltillståndet anses vara desynkroniserat.

Spelmodell

Själva spelet ligger i klassen GameModel. Där samlas alla datastrukturer och metoder som beskriver och påverkar spelets tillstånd. Den främsta uppgiften som GameModel har är att hålla reda på vilka lag som finns i spelet, vems tur det är att agera samt spelobjekten som existerar på kartan just nu. GameModel använder även andra klasser sin implementation, t.ex. MainMap och Battlefield. Även om dessa klasser representerar speltillståndet har de naturliga ansvarsområden som med fördel kan separeras från GameModels mer centrala roll.

MainMap representerar spelkartan och alla föremål på den, som städer, hjältar och övriga kartobjekt, t.ex. gruvor genom egna arrayer. MainMap ärver från MoverPathMap, som i sin tur implementerar PathMap, en abstrakt klass för vägsökning med hjälp av A*-algoritmen.

Battlefield fyller en liknande funktion som MainMap, men ska vara mer flyktig då ett slagfält ska genereras för varje strid som en hjälte påbörjar. Tyvärr hann jag inte implementera slagfält.

6.1. Milstolpar

Här listas de milstolpar som faktiskt genomfördes under projekttiden.

Milstolpe 1: Genomförd.

Milstolpe 2: Genomförd.

Milstolpe 3: Genomförd.

Milstolpe 4: Genomförd.

Milstolpe 5: Genomförd.

Milstolpe 6: Genomförd.

Milstolpe 7: Genomförd.

Milstolpe 8: Genomförd.

Milstolpe 9: Genomförd.

Milstolpe 10: Delvis genomförd.

Milstolpe 11-22: Ej genomförd.

Milstolpe 23 Delvis genomförd. (Endast skärmbytet.)

Milstolpe 24-37: Ej genomförd.

Milstolpe 38: Delvis genomförd. (En karta för 2-3 spelare, en karta för 4-8 spelare.)

Milstolpe 39: Delvis genomförd. (Det beteendet är standard, alltså inget val.)

6.2. Dokumentation för programkod, inklusive UML-diagram

Spelet styrs genom ett användargränssnitt byggt av JComponents. Olika delar av användargränssnittet implementerar olika LayoutManagers. Detta gjorde jag med flit för att få prova på olika sorters LayoutManagers. MainMenu använder MigLayout, HeroesFrame använder CardLayout, GameControlPanel använder GridBagLayout och den JFrame som innehåller MainMapInterface, GameControlPanel och GameInfoPanel använder BorderLayout.

Spelmotorn är fundamentalt kopplad till nätverkslösningen. Om man skulle vilja utveckla koden vidare till ett enspelarläge måste man fortfarande skapa en servertråd och klienttråd som kommunicerar med varandra över ett s.k. loopback-interface. Dock skulle det vara lätt att frikoppla GameModel och bygga en ny spelmotor omkring den, om man ville.

Ett spel startas genom att en GameModel skapas av GameModelFactory. GameModelFactory slumpar terrängen delvis när den skapar kartan. GameModelFactory är inte en del av projektets slutgiltiga form utan snarare en nödlösning för att kunna börja bygga spelet utan att lägga en stor mängd tid på att designa kartor och dylikt.

Varje klient som ansluter orsakar faktiskt en fullständig synkronisering av varje klients speltillstånd. Man skulle kunna vänta med att skicka speltillståndet tills efter alla klienter anslutit, och det vore också önskvärt senare då GameModel växer i storlek. Men eftersom det inte finns ett vänterum för spelare där man kan se vad som pågår på servern så föredrar jag för tillfället att ge varje anslutande spelare en visuell bekräftelse på att de faktiskt anslutit genom att skicka speltillståndet direkt. De kan inte spela innan alla andra spelare anslutit, men de ser att spelet faktiskt anslutit ordentligt.

En spelklient kan endast skicka en begäran till servern om att ändra speltillståndet, till exempel att flytta en hjälte. Om servern bekräftar meddelandet sker förändringen. Bekräftelsen sker genom att servern kontrollerar att det är klientens tur att agera samt att klienten har skickat en giltig hashkod för GameModel. Om man vill skydda sig mot fientliga klienter, d.v.s. klienter som modifierats för att fuska, vore det lämpligt att göra mer ingående kontroller på specifika meddelanden, till exempel om en hjälte kan gå som klienten påstår. Av tidsskäl har jag inte implementerat några ingående kontroller.

Efter att GameModel påverkats skickas events från GameModel och eventuellt MainMap som signalerar till lyssnande View-komponenter att en ändring har skett. På så vis kan man ansluta andra View-objekt än de som projektet använder just nu för att visa spelet.

När spelet kulminerat i att endast en spelare äger städer så deklareras vinnaren och spelet avslutas. Spelresurserna rensas bort när man lämnar GameOverScreen, och spelet återgår till ursprungsläget.

6.2.1 Resurshantering

Jag har implementerat en Singleton, GameResourceManager (GRM), för att hantera de olika trådar och Sockets som öppnas under spelets gång.

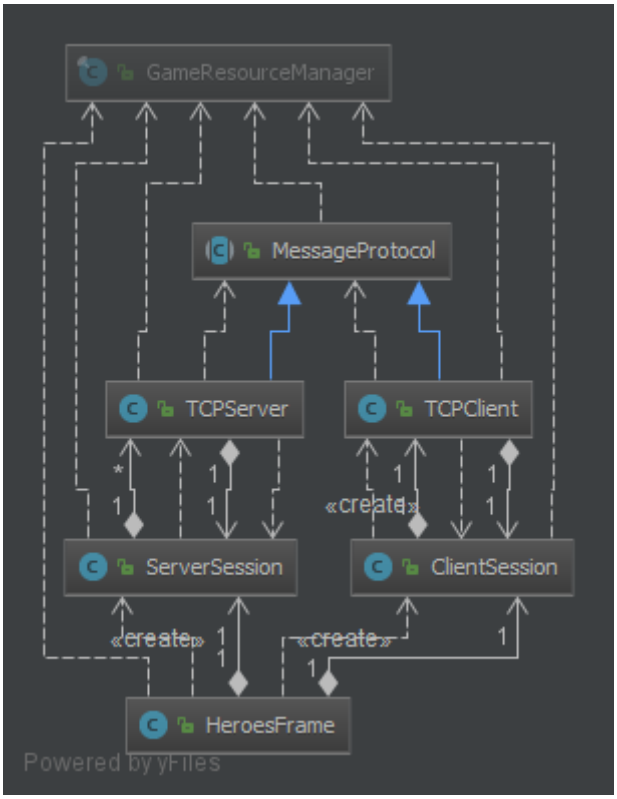

Illustration 3: UML-Diagram över GRM-beroenden

GRM används i alla klasser som öppnar Sockets eller startar nya trådar och äger allt ansvar över de trådar och Sockets som skapas, se ”Illustration 3”.

När man startar spelet så skapar HeroesFrame en ServerSession och en ClientSession om man valt att vara värd för en spelomgång, annars startas enbart en ClientSession. Dessa tilldelas egna trådar genom GRM.

ServerSession och ClientSession skapar i sin tur anslutningar som tilldelas Sockets och egna trådar från GRM.

TCPServer och TCPClient i sin tur behöver lyssna efter nya meddelanden asynkront, detta sker genom deras gemensamma superklass MessageProtocol.

GRM används också i inre klasser till dessa. Användningen av GRM för resurser som skickas över trådar gör att inga lösa trådar eller sockets lämnas när vi avslutar spelet.

Singletons är ett anti-mönster inom OO-programmering, men här kändes det ändå lämpligt, eftersom alla trådar ska använda samma Sockets, och programmets trådar är en övergripande egenskap hos hela programmet. GRM har ett tydligt och begränsat ansvar.

6.2.2 Kommunikationsstruktur

När servern och klienten kommunicerar måste det ske på ett strukturerat sätt. I projektet har jag skapat klassen Message för detta. Ett Message-objekt kan skickas eller tas emot av en klass som ärver från MessageProtocol.

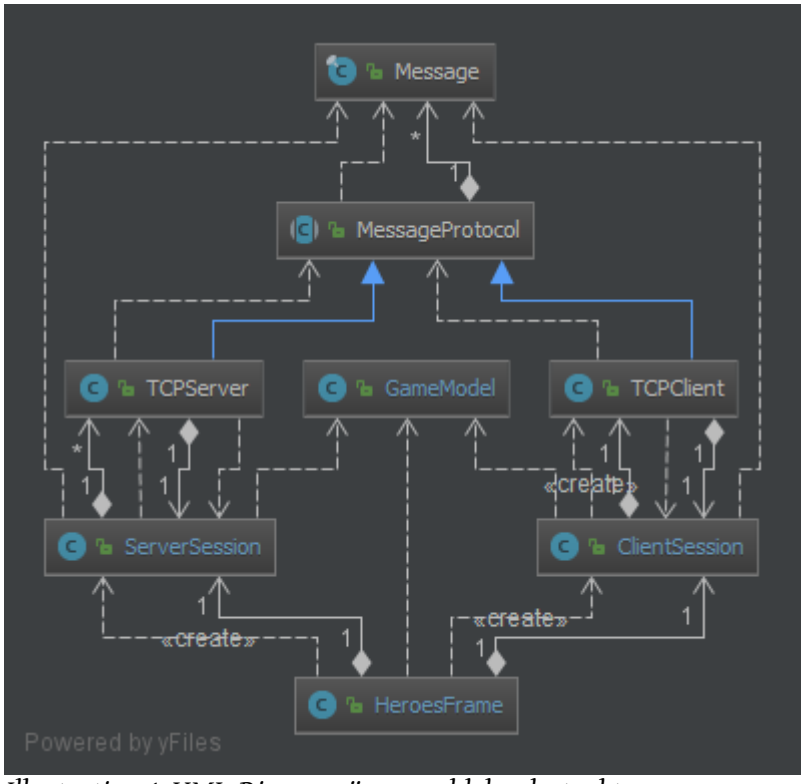

Illustration 4: UML-Diagram över meddelandestrukturen

Meddelanden tas emot via MessageProtocol i TCPClient eller TCPServer. ServerSession och ClientSession konstruerar meddelanden som skickas via deras respektive nätverksanslutning. Varken GameModel eller HeroesFrame är medvetna om Message-klassen, se ”Illustration 4”. Detta är ett exempel på inkapsling.

Message-objekt hanteras på olika nivåer beroende på deras typ. Nätverksrelaterade Message-instanser hanteras i TCPClient respektive TCPServer, övriga meddelanden passas vidare. När ett meddelande når Client- eller ServerSession tolkas meddelandet där och speltillståndet påverkas.

När en spelare vill påverka speltillståndet är det HeroesFrame och dess inre klasser som tolkar dennes kommandon. När klienten skickar en begäran så anropas en metod i ClientSession. Först där konstrueras en Message-instans för den relevanta handlingen och skickas vidare genom TCPClient.

Skulle kommunikationen avbrytas på ett oväntat sätt upptäcks det på lite olika sätt. Om servern har slutat svara blir klienten medveten om det så fort den försöker skicka ett meddelande; servern kommer inte svara som den skall. Klienten stänger då ner det pågående spelet, återgår till menyn och informerar spelaren om vad som hänt.

Om anslutningen till en klient avslutas plötsligt så kommer servern att upptäcka det omedelbart, besegra den spelarens lag och informera alla andra klienter om att spelarens anslutning avbröts.

GRM löser även ett problem som uppstår på grund av Javas implementation av BufferedReader. När en BufferedReader väntar på att läsa från en ström är tråden låst och kan inte avbrytas. För att stoppa tråden måste den underliggande Socket som används av BufferedReader stängas. GRM gör detta enkelt.

6.2.3 Vägsökning

När en hjälte ska röra sig på kartan är det viktigt att den gör det på det billigaste möjliga sättet. Det vore tråkigt att behöva styra hjälten manuellt steg för steg, så jag har implementerat en vägsökningsalgoritm. Omkring detta har jag byggt en ärvningshierarki så att jag kan återanvända koden när Battlefield implementeras.

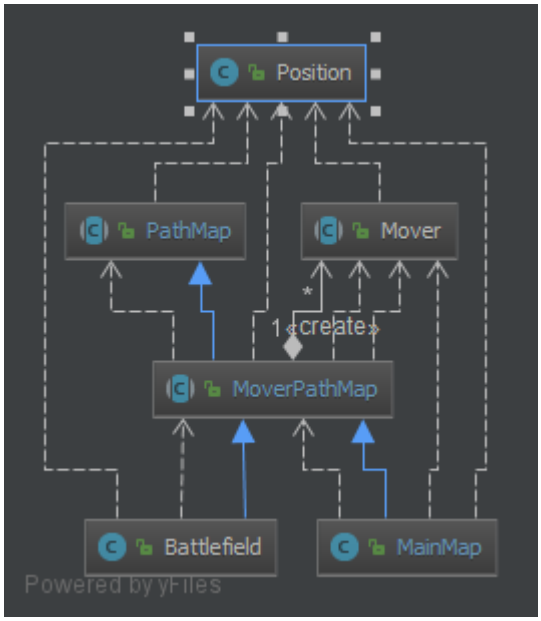

Illustration 5: UML-Diagram över vägsökningsklasserna

PathMap är en abstrakt klass som implementerar vägsökning. Genom att ärva från den kan jag definiera måttet för en kort väg för min karta och därefter enkelt vägsöka, se projektets implementation enligt ”Illustration 5”.

Notera att PathMap är frikopplad från idén om ”Mover”, och vet alltså ingenting om förflyttningskostnader eller dylikt. Det enda som PathMap jobbar med är Positions, allt annat är helt öppet för de ärvande klasserna att bestämma. Detta är ett exempel på generalisering.

PathMap använder en inre klass ”Node” för att hålla reda på enskilda grafnoder. Den välkända A*-algoritmen används sedan för att hitta den billigaste vägen till målet. När målet har hittats rekurserar PathFinder tillbaka över alla noder som lett till målet och returnerar den listan. PathMap litar enbart på de funktioner som den specifierar själv samt min egen Position-klass för att genomföra detta.

Klickar man på en nod som man inte når kommer PathMap att utforska hela den kända kartan innan den ger upp. Det innebär extremt kostsamma sökningar på stora kartor. Därför söker min algoritm från målet tillbaka till start. I teorin flyttar detta bara på problemet, men eftersom man i praktiken bara ser enstaka små områden som man inte kan nå är det en ganska stor optimering.

Mover är en abstraktionsnivå ovanför Hero, och signifierar bara en typ av Interactable som kan röra sig. Movers har förflyttningspoäng, och är på så sätt nära kopplade till MoverPathMap, som implementerar PathMap.

MoverPathMap har en del metoder som ska delas av både MainMap och Battlefield, till exempel passCost. PathMap söker bara den billigaste vägen enligt det mått som dess underklass erbjuder. Det är inte strikt nödvändigt för PathMap att veta att det t.ex. kostar två förflyttningspoäng per steg man tar på kartan, så metoden för att jämföra vägars kostnad är en del av MoverPathMap snarare än PathMap. MoverPathMap använder förflyttningspoäng för att mäta olika vägars längd, och alltså försöker PathMap hitta den väg som har lägst förflyttningskostnad, inte lägst fysiskt avstånd.

MoverPathMap erbjuder även en del metoder som är separata från vägsökningsalgoritmen, men gemensam för mina spelkartor. T.ex. kan metoden legalMoves räkna ut hur mycket av en möjlig väg som en Mover kan förflytta sig innan den får slut på förflyttningspoäng.

6.2.5 Spelmodell

Spelmodellen fördelar sitt ansvar över många mindre klasser. Det som GameModel ansvarar för är turordningen, lagegenskaper samt de olika objekten som för tillfället existerar i spelet.

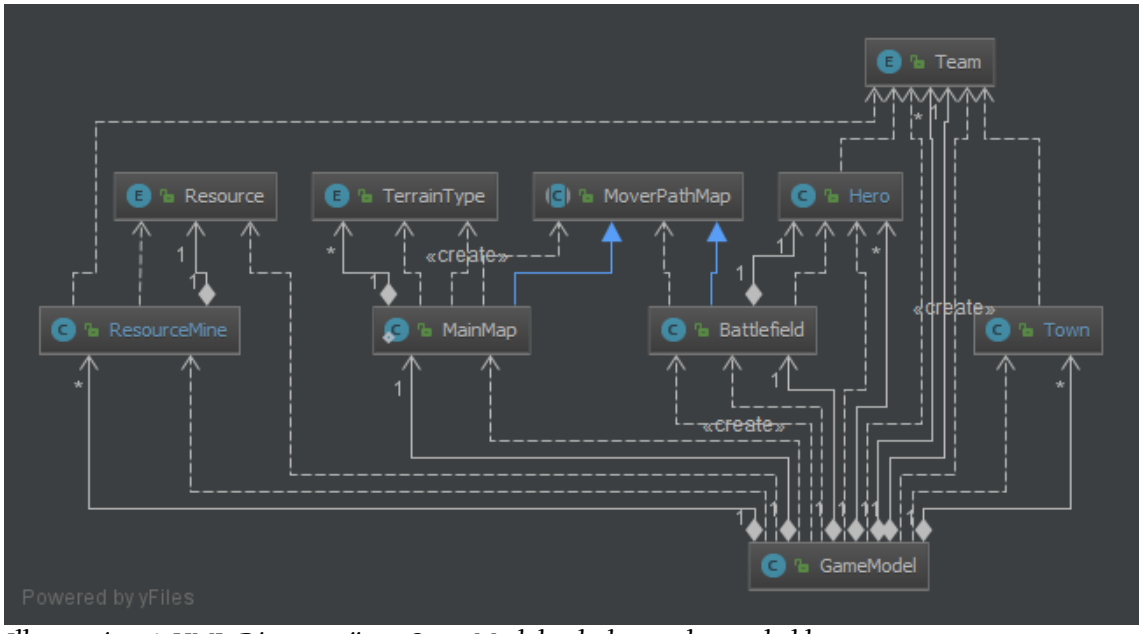

Illustration 6: UML-Diagram över GameModel och dess relaterade klasser

Det är till exempel GameModel som avgör om spelet är slut, tilldelar resurser till ett spelarlag när alla spelare genomfört sin spelrunda och vet om en spelare kan se en ruta på spelkartan eller inte. Se ”Illustration 6” för en översikt över GameModels struktur.

MainMap ansvarar för kartans utseende, egenskaper som terräng, samt att flytta hjältar omkring på kartan. MainMap innehåller även en del arrayer för att optimera Swings utritning av olika objekt på kartan.

Battlefield är inte fullständigt implementerad, men dess grund finns med som den är tänkt att fungera. Team är en Enum ämnad för att hålla reda på de olika lagen och dess enheter genom EnumMaps. Varje Interactable har en ägare, d.v.s. ett Team.

En procedurell lösning på implementationen av spelmodellen hade inneburit arrayer med information om hjältar, städer m.m. samt statiska hjälpfunktioner som arbetar på dessa arrayer som man önskar.

Eftersom de olika Interactable-objekten innehåller information om sig själva och funktioner för att läsa och påverka den information kan man arbeta på dem direkt.

Jag blev ändå tvungen att dubbellagra information om kartobjekt i MainMap. Att fråga varje interactable om sin position och sedan fråga GameModel om den positionen var synlig för spelaren i fråga blev en alltför komplex funktion att utföra under anropet till paintComponent.

Dessutom bestämmer jag kostnaden och passerbarheten i en kartruta med hjälp av Enum-klassen Passability. Eftersom jag ville ha Interactables med olika sorters Passability blev jag tvungen att lagra också den informationen i MainMap. Jag är inte nöjd med hur lösningen blev och misstänker att det finns ett bättre sätt att gå till väga, men jag lyckades inte hitta någon som inte försämrade grafikprestandan avsevärt eller tvingade mig att kombinera klassers funktionalitet.

6.2.6 Användargränssnitt

OpenHeroes använder Javas Swing-paket för att rita spelet. HeroesFrame är programfönstret i sig, och använder en CardLayout laddad med olika JComponents för att byta spelvy under spelets gång.

Att delegera all information om speltillståndet - genom ClientSession, genom GameHandler, genom GameModel, genom MainMap... - för varje objekt som ska behandlas vore för mycket jobb och skulle göra koden fullständigt oläsbar. Det vore inte heller gångbart att skapa specifika funktioner för varje sak i ClientSession, eftersom det skulle bli dess huvudsakliga uppgift då, en sorts mellanhandsklass. Därför delegerar jag genom get-metoder som hämtar GameModel och MainMap direkt.

När kartan ska ritas ut frågas alltså både GameModel och MainMap beroende på vad som efterfrågas. Detta innebär att ett starkt beroende har byggts upp mellan GameModel och MainMap. Av den anledningen har jag valt att implementera MainMap-klassen som en statisk nästlad klass av GameModel. Med andra ord, att vara en GameModel innebär att man har en MainMap. Jag har inte tolkat Battlefield så, eftersom en GameModel är värd till många olika Battlefield-instanser.

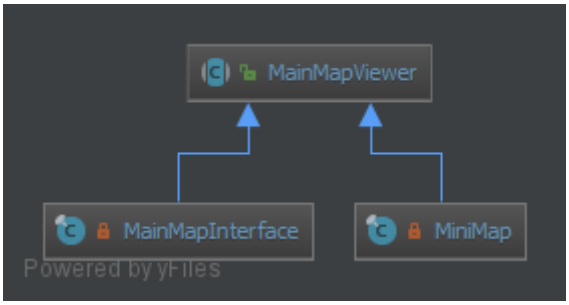

Illustration 7: UML-Diagram som visar ärvning mellan klasser

MainMapInterface och MiniMap är två UI-komponenter som ritar upp spelkartan, se ”Illustration 7”. De tolkar spelkartan på väldigt lika sätt, med enstaka fält i varje klass som bestämmer exakt vilka bilder som ska ritas och hur stora de ska vara. De använder metoder från superklassen MainMapViewer till detta.

MainMapInterface är en inre klass till HeroesFrame med ett väldigt speciellt ansvar: Det är den komponent som hanterar de kontextkänsliga händelserna i spelet. När man klickar på en Interactable på kartan kan det hända många olika saker beroende på olika faktorer:

- Är en hjälte vald?
- Har vi klickat på den här platsen förut?
- Finns det en interactable där vi har klickat?
- Äger vi den eventuella interactable-instansen, eller är den fientlig?

MainMapInterface håller reda på vilken hjälte vi har valt, vart vi har klickat samt tolkar vad vårt klick på spelkartan representerar i spelmodellen. Om en Interactable-instans är inblandad frågar vi den själv om vad som bör göras med den givet förutsättningarna. Resultatet av vårt klick hanteras sedan av metoden gameBoardClickHandler.

6.3. Användning av fritt material

Jag har skrivit samtlig kod i projektet förutom MigLayout. När jag skrev vägsökningsalgoritmen för PathMap så följde jag pseudo-koden som anges på:

https://en.wikipedia.org/wiki/A*_search_algorithm

Jag har använt Javas klassdokumentation samt Google för att fördjupa min förståelse av de olika klasserna i Javas klassbibliotek när jag stött på ett problem jag inte kunde lösa, men alltså aldrig kopierat kod.

FileIO-klassen som används för filläsning är en modifierad variant av den klass jag använde i Tetris-projektet för samma syfte. Även den skrev jag själv från grunden när Tetris-projektet genomfördes.

All grafik i spelet har jag ritat själv eller använt primitiva funktioner för att åstadkomma.

6.4. Användning av objektorientering

6.4.1. Subtypspolymorfism

Jag använder subtypspolymorfism på flera ställen i koden, men som exempel kan vi använda metoden *Interactable.interaction()*. När en hjälte ska gå omkring på kartan och interagera med olika Interactables så vill vi att rätt sak ska hända när vi klickar på en Interactable. Men hur många olika sorters Interactable finns det? Vad ska hända om vi är fredliga eller fientliga mot den?

Egentligen är det godtyckligt, och med subtypspolymorfism behöver vi inte planera det i förväg. Vi låter helt enkelt Interactable-klassen själv bestämma. När en hjälte klickar på en Interactable lämnar vi över den skyldiga hjälten och får tillbaka det meddelande vi bör skicka till servern för att interagera korrekt med den.

En procedurell lösning på det här problemet är en lång serie med if-satser där vi försöker ringa in vilken sorts objekt vi har klickat på, vilka förutsättningar som gäller med mera. Den objektorienterade lösningens främsta övertag är att koden är mer lättläst, lättare att förlänga i framtiden och mindre skör vid framtida förändringar.

6.4.2. Ärvning med overriding

Detta görs nästan slentrianmässigt i mina JComponents där jag ersätter *paintComponent()*-metoden. Ett exempel från min egen kod vore förslagsvis metoden heuristicCostEstimate i PathMap. Genom att ersätta den med en lämplig funktion för att gissa kostnaden att förflytta sig på sin karta kan man styra hur envist A*-algoritmen försöker leta efter den bästa vägen till målet.

Inom procedurell programmeringsparadigm skulle man behöva skriva någon form av booleansk struktur och byta mellan funktioner vid behov. OO-orienterad ärvning är återigen mer lättläst, man förstår vad klassen man arbetar med gör på en gång utan att gräva ner sig i olika fall. Det är också lätt att i framtiden skapa en modifierad version av en klass genom att ärva från den och ändra de funktioner man vill.

6.4.3. Typhierarkier

Interactables utgör min mest utvecklade typhierarki, se ”Illustration 8”.

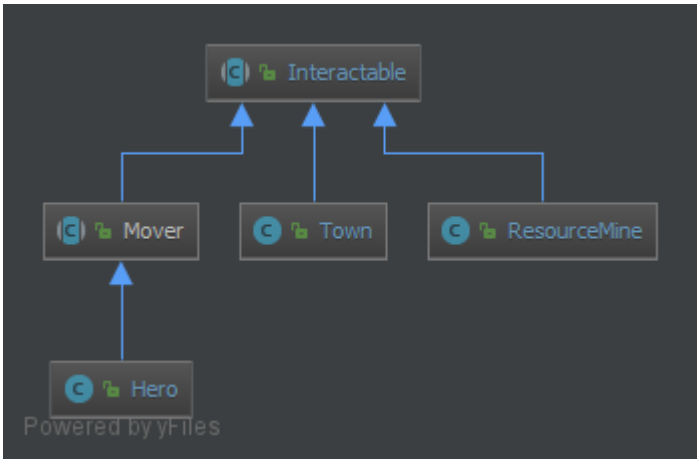

Illustration 8: UML-Diagram som visar typhierarki

Interactables förenar allt grundläggande som det innebär att vara ett spelobjekt. Mover är en förlängning som lägger till förflyttningspoäng, något som både Battlefield-Interactables och Hero-klassen behöver. Town ska vara mer distinkt än den är, med TownStructures, garnison m.m. ResourceMine genererar resurser åt spelaren.

Genom att ärva ner kan jag skriva metoder som fungerar på alla deras ärvande klasser. Detta sker överallt i min kod, men som exempel väljs Interactables i användargränssnittet på ett generellt sätt. Det är inte så viktigt just vilken sorts Interactable det är, jag behöver bara veta att objektet implementerar Interactable. Då kan jag ropa på dess leftClick, rightClick eller interaction-metoder vid behov.

Procedurell programmeringsparadigm skulle t.ex. kunna använda ett objektvärde eller dylikt för att kontrollera vilken sorts objekt det är som hanteras och tolka därefter. Inom OO-programmering är det redan givet vad vi arbetar på, och hur vi ska agera på det. Det är också lätt att förlänga funktionaliteten hos alla ärvande klasser om man upptäcker ett nytt behov, utan att behöva gå tillbaka och underhålla kod på alla olika ställen den används.

6.4.4. Inkapsling

Inkapsling är när implementationen av en lösning inte syns för klasser som inte direkt har med uppgiften att göra.

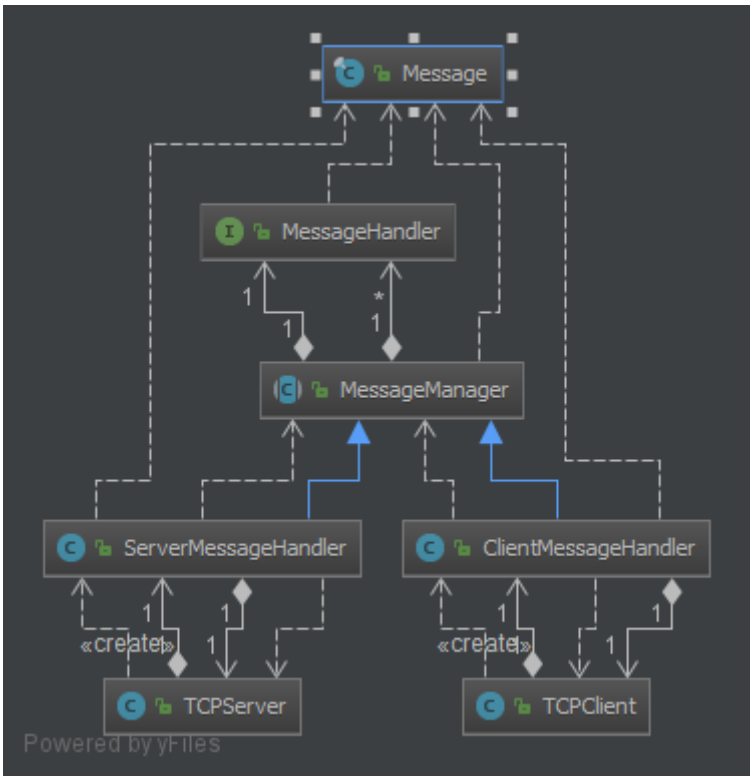

Illustration 9: UML-Diagram som visar exempel på inkapsling

I ”Illustration 9” syns hur Message-klassen är helt inkapslad från TCPServer och TCPClient.

Förvisso transporterar båda nätverksklasserna instanser av Message, men de bryr sig faktiskt inte i hur dom implementerats!

Det är istället Server- och ClientMessageHandler som ärver från superklassen MessageManager som hanterar Message-objekten.

Bilden beskriver ett extremt exempel av subtypspolymorfism, där MessageHandlers utgör objekten som tolkar Message-objekt baserat på deras MessageType. Den här implementationen gav tyvärr inte så mycket gentemot switch-strukturen den ersatte, men det var lärorikt att göra!

6.4.5. Konstruktorer

Hero initialiseras med sin egen konstruktor, och använder även superkonstruktorer för att till exempel tilldelas ett unikt ID. Den procedurella motsvarigheten till att initialisera ett objekt vore att bygga metoder som genererar datastrukturer efter behov. Objektkonceptet saknas där, så det handlar då snarare om en samling data som måste organiseras på något vis av programmeraren.

Eftersom alla objekt har egna tillstånd kan vi arbeta på och driva deras tillstånd som vi önskar. Istället för att till exempel ha en tabell över alla hjältar och deras egenskaper, så vet vi förslagsvis att vårt Hero-objekt har 15 poäng i styrka, den informationen är en del av instansen.

Detta sparar en väldig massa arbete med datastrukturer som bara är löst kopplade till det vi vill åstadkomma och är den främsta fördelen med den objektorienterade lösningen.

6.5. Motiverade designbeslut med alternativ

Under projektets gång har jag jämfört många olika varianter av de implementationer som finns i koden idag. Ofta har det visat sig att en implementation var klart mycket bättre, men ibland har jag haft flera val som varit mer eller mindre lika rimliga, eller bra lösningar som tar något avsteg från strikta OO-principer. Nedan följer några av de beslut jag tagit under projektets gång samt motiveringarna för dessa.

6.5.1 Singleton - GameResources

GameResourceManager kändes kontroversiell när jag började implementera den, och det var inte den första lösningen på trådningen jag använde. Till en början underhöll jag en serie av klasser som instanserade egna trådar för olika delar av nätverkskommunikationen. Målet var att dessa skulle avslutas på ett snällt sätt när ett spel var över så att man kunde starta ett nytt.

Problemet var att jag ganska lätt kunde tappa kontrollen över trådar när jag ville avsluta en spelomgång, och då försvann också alla mina referenser till trådarna. Konsekvensen blev att porten som jag spelat på låste och det blev krångligt att starta nya spel.

Jag behövde något sätt som jag kunde nå mina trådar och sockets på efter att jag var klar med dom, om dom inte skulle avslutas korrekt. GRM som Singleton håller alla trådar och sockets i sitt interna tillstånd och delar med sig av dem vid behov.

Alternativet vore att konsekvent introducera dekonstruktionsbeteenden i varje klass som startade en tråd. Detta skulle ha komplicerat koden till större grad än vad jag hade tid med, eftersom kursen inte meriterar trådning.

6.5.2 Message

OpenHeroes skulle tillåta flerspelarläge över nätverk. För att åstadkomma detta behövde jag kommunicera mellan klienter. Jag valde att bygga mitt eget nätverksprotokoll baserat på strängar. Olika objekt gavs en strängmotsvarighet, semikolon innebar att ett nytt objekt skulle följa. Radbrytning innebar att meddelandet var slut.

När meddelandet läses av på mottagarsidan tolkas innebörden av Message-instansens Header och alla Java-objekt byggs upp enligt deras motsvarigheter hos mottagaren.

Alternativet till detta hade varit att använda en dataström direkt och skicka Message-objekten som dom är. Så länge alla komponenter är serialiserbara så tror jag faktiskt att det skulle vara en bättre lösning.

Om man skickade hela Message-objekt så skulle jag dessutom kunna låta en Message-instans representera en GameModel-förändring. Det vill säga, mottagaren behöver bara säga åt Message att bearbeta GameModel via någon gränssnittsmetod, och sedan är uppgiften utförd. Det finns vissa begränsningar där och jag har inte utvecklat det konceptet fullt ut, men jag tror att det lurar en bättre lösning i att skicka Message-objekt i sin helhet oavsett vad.

6.5.3 Interactable

Jag behövde representera mina kartobjekt på ett konsekvent sätt, och enkelt kunna lägga till nya kartobjekt vid behov. Men kontextkänsliga användarinteraktioner kan bli extremt komplexa, och ju fler föremål jag har desto fler fall måste jag hantera.

Genom att implementera en superklass Interactable och låta den bestämma vad som bör göras med den löste jag många problem, men jag överlät också en del av spelstyrningen till själva spelobjektet.

Det kändes fel att lämna över ansvaret för användergränssnittet till spelobjekt, och min första lösning var snarare att försöka dela upp i fall och svara på varje fall. Tanken var att HeroesFrame och deras klasser skulle ta upp ansvaret för användarens indata, och att dela med sig av det ansvaret bröt mot klasskohesionen. Efter att jag ändrade strategi och lät Interactables ta över så känns det istället naturligt att de ska bestämma vad som händer med dem.

6.5.4 MainMapInterface

MainMapInterface var från början enbart en View där man kunde se spelkartan. När jag behövde ett sätt att tolka indata från klick på skärmen gavs HeroesView det ansvaret.

För att göra det behövde den bli medveten om en massa saker som inte per automatik hör till att rita upp spelet: Vilken hjälte är vald, vilken väg ska den gå, kan den nå fram?

HeroesView blev helt plötsligt MainMapInterface som visste en massa saker om speltillståndet, och ett tag till och med skickade meddelanden direkt.

Jag har separerat ut mycket av jungeln som växte fram när jag tillät HeroesView att axla flera ansvar, men jag har inte fullt ut lyckats separera kontrolldatan ännu.

Alternativet hade varit att skapa en ny klass som höll reda på användarens indata och diverse kontroll-relaterade fält. Då hade jag kunnat behålla Heroesview som en ren visningskomponent och haft tydligare ansvarsområdet för mina klasser.

6.5.5 MessageManager m.m.

Mitt program använder väldigt många Switch-satser. Men när jag tänker efter så är ju switch-satser snarare procedurell programmeringsparadigm än OO-programmering! Jag bestämde mig för att undersöka vart jag hade gått fel.

Sagt och gjort, jag läste på om subtypspolymorfism som ett alternativ till Switch-satser och skred till verket. Alla mina switch-satser för MessageType på nätverksanslutningens kontaktyta ersattes med subtypspolymorfism.

MessageHandler är ett interface som tolkar ett Message. Alla klasser som implementerar MessageHandler har en funktion, *handleMessage()*. MessageManager har en EnumMap där den mappar instanser av MessageHandlers till MessageType-headers. På så vis hittas rätt MessageHandler på Log(1) tid varje gång. Perfekt!

Tyvärr blev min kod mycket mindre lättläst på grund av detta, och jag spenderade faktiskt åtskilliga timmar med att konstruera om även GameHandler enligt denna design i hopp om att hitta någon gemensam nämnare och därmed fördel. Resultatet blev faktiskt sämre; mer kodduplicering och särskilda fall. Det blev så illa att jag inte kunde motivera förändringen när den var implementerad och var tvungen att dra tillbaka hela omfaktoriseringen.

6.5.6 ResourceProducer – eller inte!

Vid slutet av varje spelrunda ska varje spelare tilldelas resurser. Men vilka objekt ska tilldela resurser? Alla Interactables gör inte det. Jag behövde hitta en gemensam nämnare för objekt som producerar resurser.

ResourceProducer är ett interface som jag implementerade för Town- och ResourceMine att dela på. De ska trots allt båda tillföra resurser till spelaren när alla spelare tagit sin spelrunda.

När jag faktiskt delade ut resurserna visade det sig onödigt att gruppera Town och ResourceMine i någon gemensam datastruktur, så jag slopade detta interface. Både Town och ResourceMine har förvisso den metoden som var associerad med interface-klassen, men den fyllde aldrig någon funktion som ett koncept i sig.

I framtiden kanske jag ser ett större behov av att gruppera klasser som producerar resurser och implementerar detta interface igen, men som projektet ser ut just nu var det inte motiverat.

6.5.7 Team, Enum

Jag behövde ett sätt att koppla spelare till sina nätverksanslutningar. De har förvisso sina spelarnamn, men vad händer om de har samma namn? Servern skulle bli förvirrad om vem som har skickat meddelanden om namn användes för att sortera anslutningar.

Team fyllde redan en liknande roll i GameModel där den separerade anslutna spelare från lagen i speltillståndet. Jag valde att använda Team-enumen en andra gång för att också representera server- och klientnamn.

Ett alternativ hade varit att tilldela något annat objekt som identifierar varje anslutning, till exempel en sträng. Nackdelen med det är att man introducerar ett nytt fält som man måste hålla reda på.

Jag tycker att det var en bra lösning eftersom det antalet lag alltid matchar antalet klienter, och jag inte behöver underhålla någon form av counter eller dylikt på server- och klientsidan.

6.5.8 Grafiska resurser

Jag använder en del olika bilder för att rita upp olika spelobjekt. Jag behövde bestämma vart referenserna till dessa bilder skulle sparas, och hur de skulle laddas.

Jag valde att använda en statisk hjälpklass för att ladda mina bilder. Hjälpklasser är ett anti-mönster in OO-programmering, men jag tycker att det var motiverat i det här fallet för det är inte varje enskilt objekts ansvar att ladda in bilder. De behöver bara berätta vilken bild det är de vill använda och be hjälpklassen ladda den.

Alternativet hade varit att fördela allt ansvar att ladda spelresurser på en klass och sedan på något sätt hämta dom vid behov. Det hade skapat dataregister där man associerar bilder med objekttyper, så jag föredrar att koppla bilderna direkt till objekten som använder dom.

6.5.9 Passability i Interactables

Passability är en Enum som representerar kostnaden att passera en ruta. TerrainType har en Passability, det känns naturligt. Men Interactables har också Passability! Jag ville nämligen att hjältar skulle kunna passera vissa sorters Interactables. Ett trivialt exempel är porten vid en Town, eller ingången till en ResourceMine.

Varje Interactable innehåller en PassabilityMap som frågas varje gång någonting försöker vägsöka till eller omkring den Interactable-instansen. Detta lät mig bestämma skräddarsydda kostnadskartor för en enskild typ av Interactable.

Nackdelen är att detta lägger till mycket kod- och beräkningskomplexitet. Jag blev tvungen att skapa en ny datastruktur bara för att hålla reda på vart kartobjekt befann sig, annars skulle paintComponent ta för lång tid på sig att genomföra beräkningarna för om det fanns en interactable, om spelaren såg den m.m.

Jag är kluven över om det var värt funktionaliteten att skapa en dubbellagring och extra beräkningar i mina repaint()-anrop. Det har också inneburit svårigheter i att fullständigt separera MainMap från GameModel på ett rent sätt.

Att helt enkelt låta alla Interactables ha en enskild punkt man skulle vara nära för att interagera med den hade fungerat, även om det varit lite tråkigare. Om jag skulle göra om projektet är jag osäker på hur jag skulle valt att göra.

6.5.10 GameModels ansvar

Jag behövde hålla reda på varje lags specifika information, som vilka hjältar, städer och gruvor de ägde, vilka delar av kartan de såg samt hur mycket resurser de hade.

Det kändes logiskt att låta MainMap ansvara för dom sakerna, särskilt eftersom MainMap behövde särskild tillgång till kartans synlighet, d.v.s. ”Fog of War”. Det var inte förrän mycket senare jag insåg hur mycket mina klassers smälte samman på grund av detta. GameModels ansvar urholkades nästan totalt och MainMap representerade speltillståndet mer än GameModel. Jag blev tvungen att skapa en hjälpklass för att ta reda på objekts tillstånd.

Det var då jag insåg att ägarskap borde vara en egenskap hos varje Interactable, inte GameModel eller MainMap. Då kunde GameModel bara ha en lista över vilka hjältar som fanns, och när man behövde veta saker om en Interactable så frågade man bara den! Jag byggde faktiskt om projektet för att fixa skadan som detta designbeslut orsakade.

7. Användarmanual

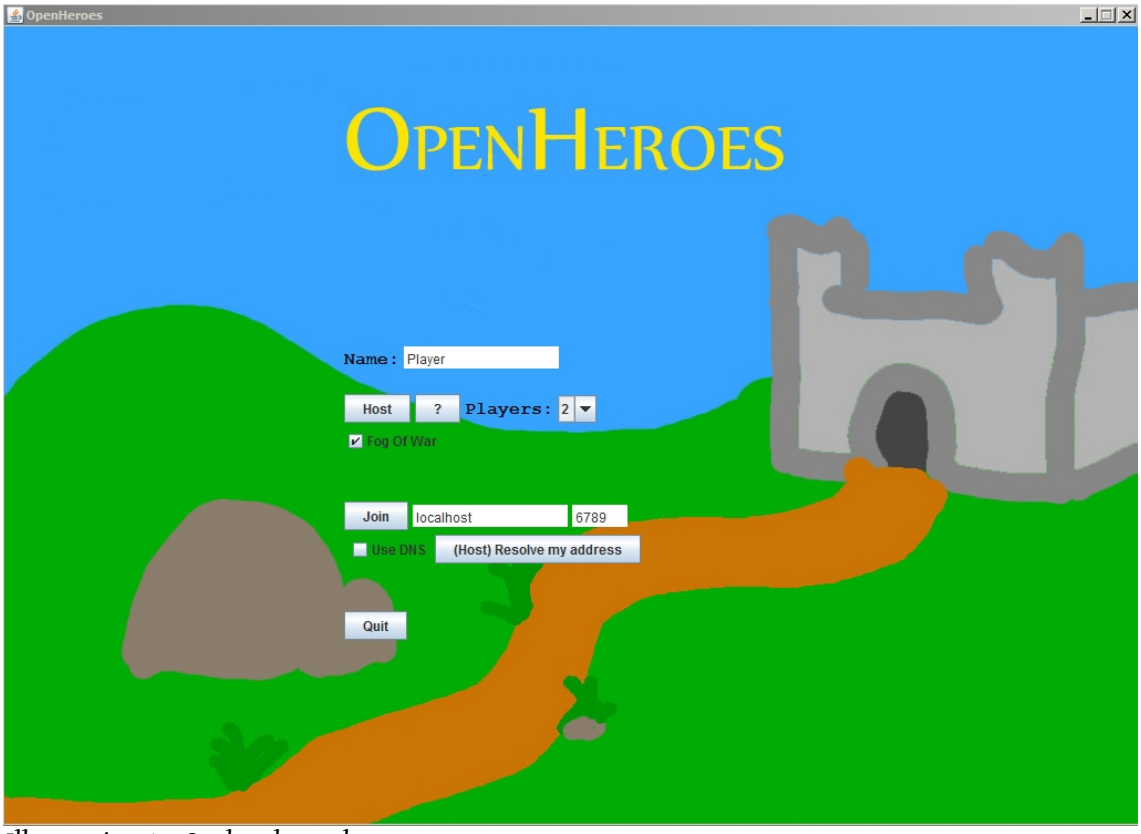

Illustration 10: Spelets huvudmeny

Huvudmenyn låter dig välja namn, vara värd för ett spel eller gå med i någon annans spel, se ”Illustration 10”. Det är viktigt i Java vilken IP-adress man binder sig till som värd, därför har jag inkluderat en ”snabb-knapp” för att välja nätverksinterface. Den använder Javas egna klasser och jag kan inte lova att den väljer rätt alla gånger! Den lilla hjälpknappen bredvid ”Host”-knappen förmedlar samma idé som jag beskrivit i den här paragrafen.

Det går även att försöka tolka DNS, d.v.s. ange ett namn som tolkas av en DNS-server som en IP-adress. Använder man den funktionaliteten kontrolleras inte rimligheten av det du matar in.

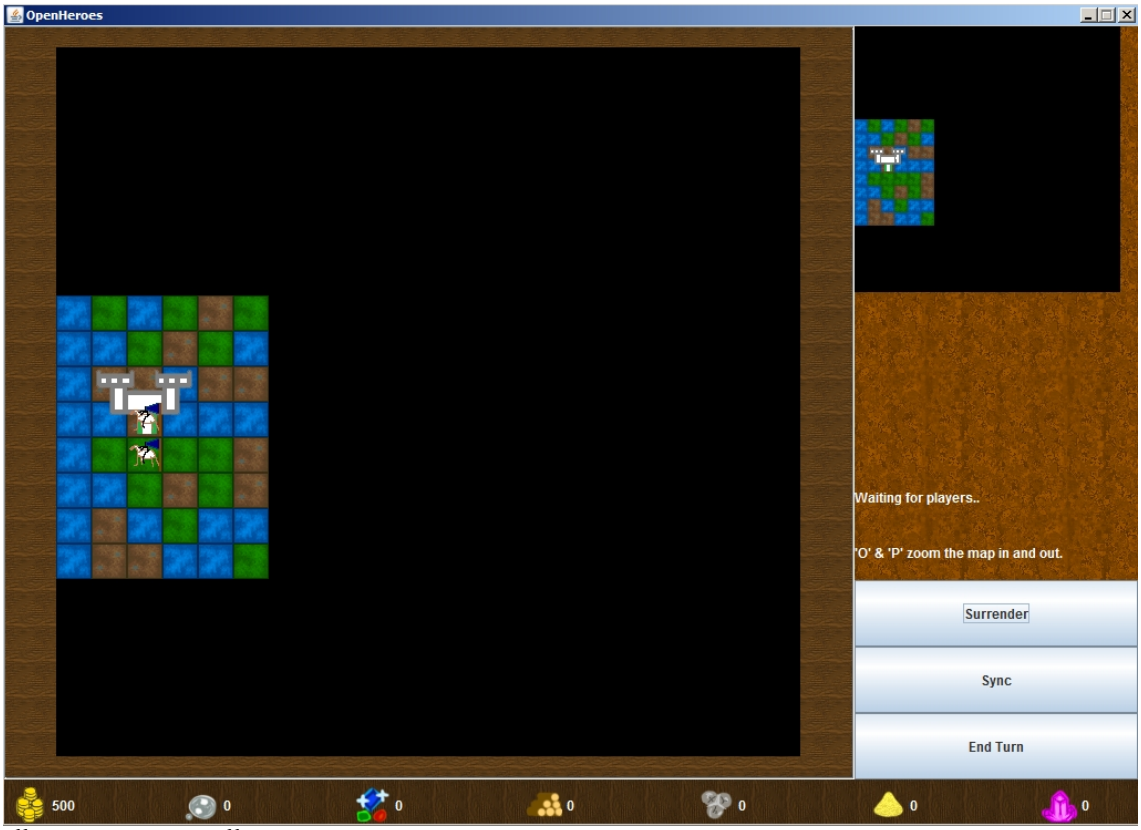

Illustration 11: Spelkartan

När en värd startar spelet laddas kartan på en gång, men det går inte att göra någonting annat än att ge upp förrän alla spelare har anslutit, se ”Illustration 11” samt ”Illustration 12”.

När alla spelare anslutit tar varje spelare sin omgång i den tur som slumpades när kartan skapades. Efter att alla spelare spelat klart tilldelas resurser baserat på vad man äger på kartan. Man kan zooma huvudkartan in och ut med 'O' och 'P'.

Illustration 12: Hjältar utforskar spelkartan

Kontroller:

Vänsterklicka på hjältar och städer för att interagera med dem.

Första gången man vänsterklickar på en hjälte väljs den, därefter kan man interagera

med andra saker på kartan.

En hjälte måste aktivt interagera med ett kartobjekt för att påverka det. Det räcker alltså inte att hjälten går förbi objektets interaktionspunkt av en slump!

Högerklicka för att avmarkera den nuvarande hjälten.

'O' respektive 'P' zoomar kartan in och ut.

Om man högerklickar på spelkartan och håller knappen nedtryckt, sedan drar musen utanför spelkartan så kommer kartan att scrolla, om inte hela kartan redan syns.

Kontrollpanelen har några knappar:

Surrender – Ger upp spelet efter en bekräftelse.

Sync – Begär att servern skickar en ny version av speltillståndet.

End turn – Avslutar spelturen, om det är din speltur.

Illustration 13: Det går att köpa hjältar från städer som laget kontrollerar.

Om man klickar på en stad när man inte har en hjälte vald får man valet att köpa en ny hjälte för 500 guld, se ”Illustration 13”. Klienten ansvarar för att du faktiskt har så mycket guld, men om du hackar spelet och skickar en begäran till servern ändå så kommer den att säga ifrån och uppdatera ditt speltillstånd.

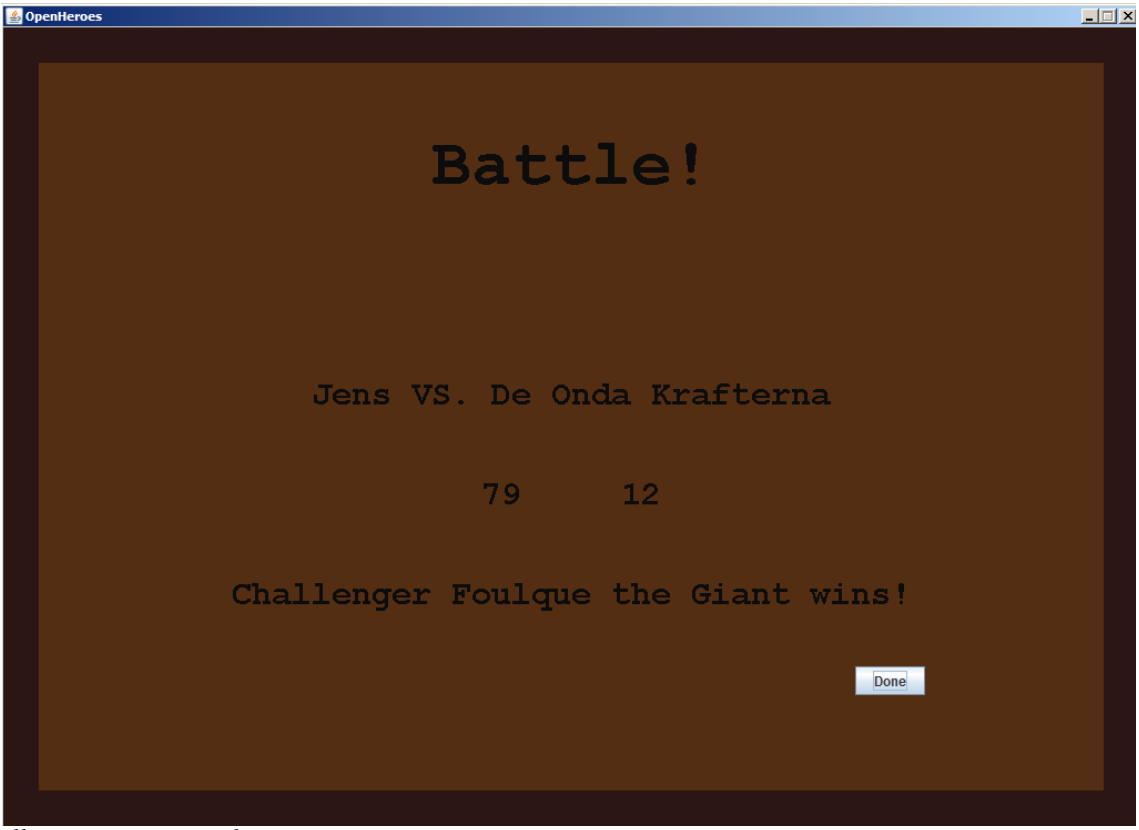

Illustration 14: Stridsvyn

När man kontrollerar en hjälte och skickar den emot en fientlig hjälte genom att klicka två gånger på den så går de i strid, se ”Illustration 14”. Eftersom stridsfält inte implementerats ännu finns det en tillfällig lösning, där spelservern slumpar två tal och skickar dem till varje klient. Den som får högst resultat vinner. Utmanarens resultat är alltid till vänster.

8. Slutgiltiga betygsambitioner

Min ambition är att få en femma i den här kursen!

9. Utvärdering och erfarenheter

Mycket av det jag gjorde i OpenHeroes gjorde jag för första gången. Nätverkskommunikation och trådning är ingenting som jag hållt på med förut så det var definitivt en utmaning. Jag ångrar lite att jag försökte mig på detta eftersom kursen inte krävde det så blev det en stor mängd extra arbete, men det var också väldigt lärorikt.

Jag gick på alla föreläsningar, men använde ingen hjälp från labbtiderna. Problemen jag råkade ut för var inget som labbassistenterna förutsattes ha kunskap om, så jag förlitade mig på Java-dokumentationen, kurshemsidan och Google när jag behövde veta mer om olika metoder och koncept. Dessutom kändes det bättre att jobba hemifrån.

Jag har lagt ner alldeles för mycket tid på detta projekt. Min rädsla var att arbetet jag lagt ner på nätverks- och trådningsdelen inte skulle räknas inom ramarna för kursen så det slutade med att jag la ner extremt mycket tid på att försöka säkerställa att jag verkligen jobbade med korrekt OO-programmering.

Arbetet fortskred dock ungefär som jag tänkte mig, jag har följt mina milstolpar. En del milstolpar krävde att jag byggde grunden för senare milstolpar innan jag implementerade dom, annars skulle jag blivit tvungen att bygga om koden senare.

Det har inte varit svårt att hitta tid till projektet för mig, jag har lagt ledig tid hemma på det. Men det har krävt mycket tid att polera projektet.

Om jag skulle göra om den här kursen så skulle jag inte välja att jobba med nätverk eller trådning. Det var otroligt lärorikt men också väldigt stressigt att försöka implementera tillsammans med all övrig funktionalitet.

Jag tycker att kursen varit utformad på ett bra sätt, kanske skulle man kunna visa på JavaFX som ett alternativ till Swing.

Inlämningsinformationen är lite utspridd på olika sidor, till exempel nämns inte att man ska ha ett labbomslag med sig till demoredovisning under fliken projektbedömning på hemsidan.