Rabbits and wolves

Processorienterad programmering (1DT049) våren 2012. Slutrapport för grupp 8

Daniel Gustafsson – 841007
Andreas Lindmark – 820803
Johan Risch – 900222
Linnea Sandelin - 910730

Innehållsförteckning

1. Inledning

Projektets stora mål var att få kunskaper om concurrency inom programmering. Inför projektet fanns många olika förslag för vad som skulle kunna göras. De olika förslagen var bland annat:

- RABBITS AND WOLVES:

Sammanfattning: en simulering av en karta med kaniner, gräs och vargar. Kaninerna äter gräs och vargarna äter kaniner. Både kaniner och vargar kan svälta (gräsbrist eller brist på kaniner) eller dö av ålder, men gräset kommer alltid att kunna växa, dock minskar gräsnivån i varje ruta då kaniner äter. Kaniner och vargar parar sig om de är tillräckligt mätta och har åldern inne för det.
***Concurrency*:** varje kanin och varg är en egen process som kommunicerar med kartan.

- SCHEDULE:

***Sammanfattning*:** tidsbokningssystem där flera klienter kan kommunicera med en server och boka tidspass. Likt Doodle eller UU:s schemabokningssystem.
***Concurrency*:** flertalet klienter kan samtidigt försöka boka ett tidspass.

- RAY TRACE:

***Sammanfattning*:** projicering av en 3D-model på en 2D-yta genom användning av strålar.
***Concurrency*:** varje stråle är en egen process.

- TRAFFIC SIMULATION:

***Sammanfattning*:** simulering av ett vägsystem med trafik, trafikljus och dylikt. Varje fordon har egna parametrar såsom hastighet, acceleration etc. Liknar "Rabbits and wolves" då båda är simuleringar.
***Concurrency*:** varje fordon är en egen process som måste kolla om och när det får köra.

- MATRIX MULTIPLICATION:

***Sammanfattning*:** implementering av vanlig matrismultiplikation.
***Concurrency*:** varje cell i matrisen beräknas av en egen process.

- TRAVELLING SALESMAN:

***Sammanfattning*:** flera försäljare reser i en graf av noder och försöker hitta den kortaste vägen mellan olika noder. Även denna liknar "Rabbits and wolves" då det är en simulering.
***Concurrency*:** varje försäljare är en egen process som försöker hitta den snabbaste vägen mellan olika noder.

1.1. Syfte och målsättning

Projektet som valdes, efter omröstning, var ”Rabbits and wolves”. Syftet med detta projekt är att integrera två olika programmeringsspråk, Java och Erlang, samt att få kunskaper om hur concurrency tillämpas inom programmering. När projektet är färdigt är målsättningen att en färdig simulering av systemet ”Rabbits and wolves” skall finnas, där varje kanin och varg skall vara en egen process som kommunicerar med kartan och genom detta äter, förflyttar sig och parar sig för att i största möjliga mån överleva. Att simuleringen är lyckad inkluderar även att språken skall ha integrerats på ett lyckat sätt samt att concurrency:n i detta fall skall fungera.

1.2. Begränsningar

En begränsning som valt att göras är att användaren har små möjligheter att ställa in parametrar, exempelvis är vargars och kaniners ålder, könsmognad, hungersnivå samt prioriteringsordning för mat/förflyttning hårdkodat för att passa projektets storlek. Från början var även tanken att ha med en funktion så att användaren själv kan ställa in hastigheten för simuleringen och ändra denna under körningens gång, även denna idé har dock fått skippas då tiden ej räckt till.

Genetisk programmering (att vargar och kaniner får en färbättrad överlevnadsinstinkt genom arv från tidigare generation, exempelvis att de kan hitta mat på ett effektivare sätt) har även uteslutits då detta skulle bli för tidskrävande i jämförelse med storleken på arbetet. Mellan Java och Erlang finns även en begränsning i Jinterface:t, en så kallad flaskhals, som begränsar hur fort kommunikationen mellan kartan och vargarna/kaninerna faktiskt kan ske.

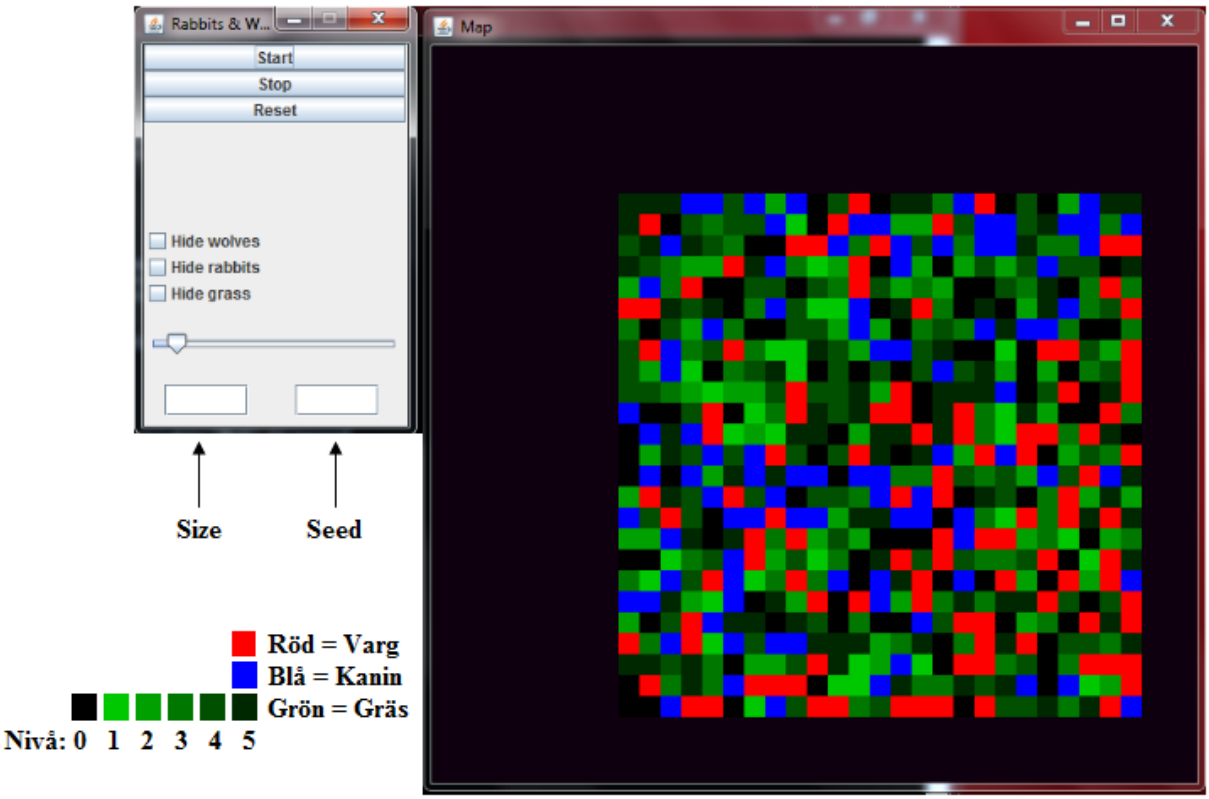
2. Rabbits and wolves

När programmet startas får användaren upp två fönster, se Figur 1. Ett fönster med en karta, samt ett fönster med olika knappar benämnda med ”start”, ”stop” och ”reset”. Två textfält för att ange dels en seed för slumpgeneratorn, samt storleken på kartan (den är kvadratisk så endast en siffra behöver anges) syns även i detta fönster. Tre rutor ”Hide rabbits”, ”Hide wolves” och ”Hide grass” kan även markeras.

2.1. Knappar och parameterinställningar

- När användaren trycker på knappen ”start” startas simuleringen och systemet kör igång.
- När användaren trycker på knappen ”stop” pausas simuleringen och är avbruten tills användaren återigen trycker på ”start”.
- När användaren trycker på knappen ”reset” återställs systemet till hur det såg ut vid senaste start då ändring av parametrar (size och seed) ägde rum.

- Seed:en som anges kommer att avgöra hur kartan byggs upp (gräsnivåer, antal kaniner och vargar vid start) i förhållande till storleken på kartan genom att skickas till en slumpgenerator. Siffran som här matas in skall vara ett reellt tal. Anges samma seed två körningar i rad, skall utseendet för kartan i de båda utgångslägena bli identiska. Anges ej något värde för detta kommer ett standardvärde ges av programmet.
- I fältet med ”size” anges ett reellt tal n > 3, denna sätter då kartstorleken till n x n rutor. Anges ej något värde för detta kommer ett standardvärde ges av programmet.
- Markeras någon av ”Hide” knapparna kommer den eller de markerade att döljas i kartan. Markeras exempelvis ”Hide rabbits” kommer endast vargar och gräs att vara synliga i karta, dock finns kaninerna fortfarande kvar och jobbar, men de är osynliga för användaren.



Figur 1. Utseende när programmet startas

3. Programmeringsspråk

När projektet påbörjades fanns flertalet funderingar kring vilket/vilka språk som var lämpliga att använda. Det kändes passande att välja två språk för storleken på detta projekt. Att integrera olika språk är något som är nyttigt att kunna, men med projektets storlek i åtanke kändes det mycket olämpligt att välja mer än två språk. Tankarna bakom språkvalen var dels

att ha ett språk som alla medlemmar i gruppen programmerat i förut men vill utveckla sina kunskaper inom, samt ett nytt språk då detta ansågs vara en ypperlig chans att lära sig. Det ansågs även lämpligt att ha två olika typer av språk, i detta fall blev det ett funktionellt (Erlang) och ett objektorienterat (Java).

3.1. Val av språk

Vid valet av vilka språk som skulle användas behövde några olika saker tas med i beaktan. Till att börja med drogs slutsatsen att två språk skulle behövas, ett språk lämpat för att skriva de delar som innehåller concurrency i och ett lämpat för att skriva ett GUI menat att göra programmet lättanvänt i. Programmeringsspråken som valdes var Java och Erlang.

Erlang är ett språk som är mycket lämpligt att skriva i när concurrency skall vara med i bilden. Detta då det i detta språk ”kostar” lite att ha många processer, och det är mycket lätt att hantera många processer samtidigt. I Java är det mycket mer omständigt att hantera trådar och därmed både mer tidskrävande och riskablare att försöka få till concurrency:n i detta språk. Att programmera i Erlang var något som var nytt för alla medlemmar i projektet, då det är ett välanvänt språk som är nyttigt att kunna, ansågs projektet vara ett mycket bra tillfälle för att lära sig språket i fråga. Med detta i beaktan valdes Erlang som det språk kaninerna och vargarna skulle programmeras i.

Java är ett mycket bra språk när det gäller att producera GUI:n. Tack vare Javas bibliotek Swing kan man programmera snygga och lättanvända GUI:n inom lämpliga tidsramar. Att försöka få till ett GUI i Erlang kändes mycket svårt och olämpligt. Fördelen med att programmera i Java är att alla medlemmar i projektet tidigare gjort det och därför har kunskaper inom språket. Man kan då hjälpa varandra och utveckla sina kunskaper, vilket inte alls är någon nackdel med tanke på hur stort och välanvänt Java är. Tack vare detta valdes Java som det språk GUI:t skulle skapas i.

4. Systemarkitektur

Systemet är uppdelat i fyra betydande delar. Dessa delar är:

- Vargar och kaniner programmerade i Erlang
- Karta programmerad i Java
- GUI programmerat i Java
- Jinterface med skrivna metoder i Java

Se bild över systemarkitekturen i Figur 2.

4.1. Vargar och kaniner

Varje varg och kanin i systemet är en egen process som kommunicerar med kartan. Kaninerna och vargarna åldras vid varje tidssteg som går och blir även hungriga om de ej får äta. Vid varje tidssteg kan kaninerna och vargarna antingen:

- Äta
- Gå
- Para sig
- ”Sova”

Om en kanin står på en ruta med gräs och är hungrig, äter den, om den är hungrig och har en ruta med gräs bredvid sig rör den sig mot denna ruta. Om en varg står bredvid en kanin och är hungrig, äter den upp kaninen. Kaninen meddelas då om att den dödats och processen avslutas. När en kanin eller varg föds har de ålder 0 och hunger 0. Om en kanins ålder överstiger 70 (70 tidsenheter sedan födsel) eller en vargs 40 kommer de att dö. Om en kanins hunger överstiger 5 (5 tidsenheter i följd utan att få äta) eller en vargs överstiger 10 kommer de att dö av svält.

Om en kanin eller varg står på en ruta där det ej finns mat eller möjlighet att para sig försöker de att röra på sig. Detta gör de genom att ta emot en lista med möjliga rutor att gå till (vänster, höger, upp, ner samt diagonalt) och kontrollerar på vilken av dessa rutor det finns mest tillgång till mat och inte står någon annan varelse (vargar går dock mot rutor med kaniner eftersom detta är deras mat). Finns ingen mat i någon av rutorna väljer den en slumpmässig ruta som är tom.

Om två kaniner eller vargar står brevid varandra, har ålder inne och är tillräckligt mätta, se Tabell 1 för krav, parar de sig och en ny kanin eller varg skapas på kartan.

Om en kanin eller varg inte får/kan göra något av ovanstående under ett tidssteg kommer de att ”sova”, de kommer då att vänta 200 ms och sedan försöka på nytt.

	Maxålder	Maxhunger	Krav för parning
Kanin	70	5	Ålder > 15 & hunger < 3
Varg	40	10	Ålder > 20 & hunger < 3

Tabell 1. Attribut för kaniner och vargar

4.2. Karta

Kartan är uppbyggd av en 2D-array. I varje ruta i arrayen finns det info om gräshalt och om det står någon varelse i rutan. Om ingen varelse äter av gräset i en ruta kommer detta att växa i varje tidssteg, tills en bestämd maxnivå nås. Om en kanin äter gräs i en ruta sjunker gräsnivån med en enhet per tidssteg tills kaninen blir mätt eller tills gräsnivån är 0.

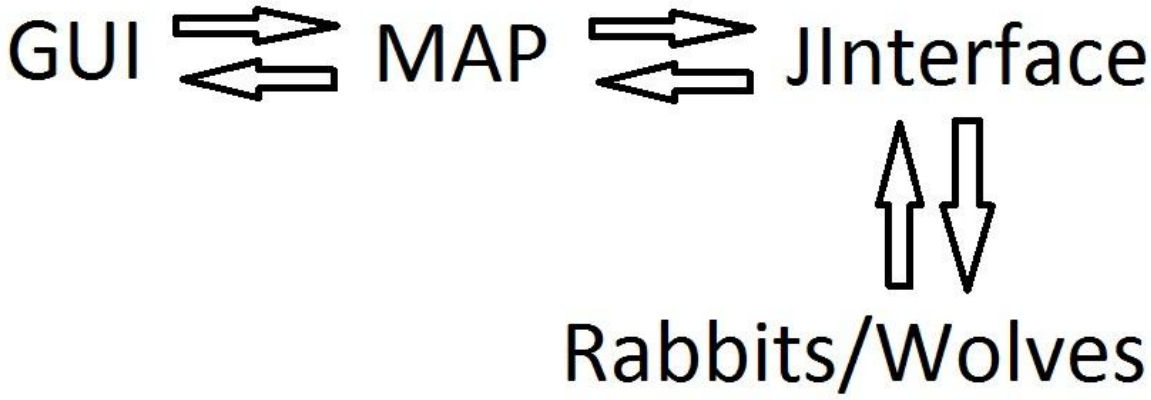
Kartan får förfrågningar från kaninerna och vargarna om de får röra sig, och får en uppdatering av kaninerna om de ätit och gräsnivån därför behöver sänkas. Kartan skickar även ut meddelanden till varelserna om de av någon anledning dött (t.ex. blivit uppätna) och därför skall dödas.

4.3. GUI

GUI:t är skrivet med hjälp av Java och Swing. GUI:t kommunicerar med kartan och får uppdateringar av kartan så fort något ändrats. När någon tråd i kartan är klar meddelas GUI:t och kartan, som är synlig för användaren, uppdateras.

4.4. Jinterface

Jinterface sköter kommunikationen mellan Java- och Erlangdelarna. Här skapas noder på Javasidan som kan ansluta till Erlangnoderna, meddelanden som skickas från vargarna och kaninerna översätts till kartan och tvärtom.



Figur 2. Systemarkitektur för systemet ”Rabbits and wolves”

5. Samtidighet (concurrency)

I Erlangdelen är varje kanin och varg en egen process som handlar självständigt, och ”samtidigt”, samt under kommunikation med kartan. Kartan har två trådpooler, en för trådar som exekverar en ”runnable” vid förfrågning, samt en med de olika typerna av ”runnables”. En ”runnable” är ett interface som gör att man kan köra en klass på en tråd. Kartan läser kontinuerligt nya meddelanden från kommunikatorn (från Erlangdelen) som den matchar ut och utifrån detta startar rätt typ av ”runnable”. Trådarna låser endast de noder de behöver beröra, detta görs genom att börja vid låga index och gå till högre.

5.1. Actor model

I Erlangdelen används den så kallade ”Actor modellen”, som innebär att varje process är egen och kan ta emot och skicka meddelanden samt starta nya processer. Varje kanin/varg skickar meddelanden till kartan som behandlas av Jinterface:t innan det tas emot av Javadelen. Kaninen/vargen tar sedan emot meddelanden från kartan på motsvarande sätt och startar nya processer (nya kaniner/vargar) vid förfrågan om detta.

5.2. Synkronisering

Synkronisering sker varje gång kartan skickar meddelanden till GUI:t om att något uppdaterats och GUI:t då uppdaterar kartan som är synlig för användaren. En form av synkronisering är även mellan kartan och kaninerna/vargarna. Detta genom att när en kanin eller varg vill röra sig skickar den meddelandet ”move” och koordinater till kartan, kartan svarar sedan med ett ”ok” eller ”no”. Innan detta svar kommit kan kaninen/vargen inte göra något (om det ej tar mer än 200 ms och kaninen/vargen frågar på nytt igen).

5.3. För- och nackdelar

Fördelarna med den valda implementationen är att kaninerna och vargarna är självständiga och kan arbeta ”samtidigt” då de alla är en egen process. Att skriva GUI:t i Java med hjälp av Swing var även det mycket lämpligt då det finns bra vertyg för att bygga upp GUI:n. Att skriva kartan i Java har även det varit lämpligt då det är lätt att implementera 2D-arrayer.

En nackdel som uppstått är att det bildats en flaskhals vid kommunikationen mellan Erlang och Java då all kommunikation sker genom en ”väg”. Hade mer tid funnits hade detta utvecklats och flera ”vägar” lagts till så att flaskhalssyndromet undvikts.

En annan implementation som hade kunnat vara lämplig är att programmera hela projektet i ett språk. Mycket prestanda tappas vid kommunikation mellan dessa språk.

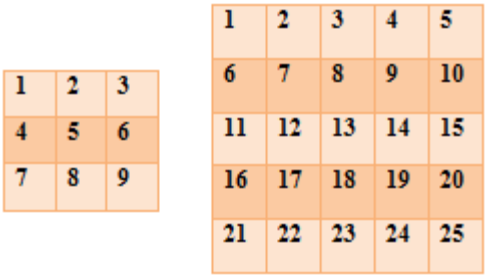
Vid den valda implementationen undviks deadlocks genom så kallad ”absolut ordning”. Vargarna och kaninerna väljer alltid rutor de försöker gå till i en speciell ordning och låser därmed även rutor i samma ordning, så två stycken kommer aldrig att försöka gå till samma ruta exakt samtidigt. Försöker två stycken ta ett lås på samma ruta kommer den som hinner först få låset, medan den andra väntar. Antingen kommer varelsen med låset att släppa detta och den nya kommer därmed kunna ta låset, eller så kommer (i värsta fall) den som har låset att dö av ålder eller svält och det släpps därmed tack var detta.

6. Algoritmer och datastrukturer

För kaniner och vargar gäller prioriteringsordningen som följer:

- Om de är hungriga och det finns mat (gräs eller kaniner), ät.
- Om de har åldern inne, de är tillräckligt mätta, en till varelse av samma art står bredvid samt en ledig ruta i närheten finns för den nya avkomman, parning.
- Försök till förflyttning, sker genom:
    - Kaniner:
        - Kontrollera rutor kring rutan den står i med avseende på gräshalt, se Figur 3 för ordning.
        - Välj ruta med mest gräs, slumpmässig ruta blad de med högst halt om flera har samma nivå.
    - Vargar:
        - Dela upp omkringliggande rutor med avseende på föda och möjlighet till förflyttning, se ordning i Figur 3.
        - Om vargen är hungrig, gå mot ruta med kanin i närheten.
        - Om vargen är mätt, gå mot tom ruta.
- Om ej något av ovanstående fungerat, vänta 200 ms och försök från början igen.

En av de viktigaste delarna är att systemet synkroniseras med hjälp av ”absolut ordning”, se Figur 3, för att undvika deadlocks. Detta diskuteras närmare i del 5.3.



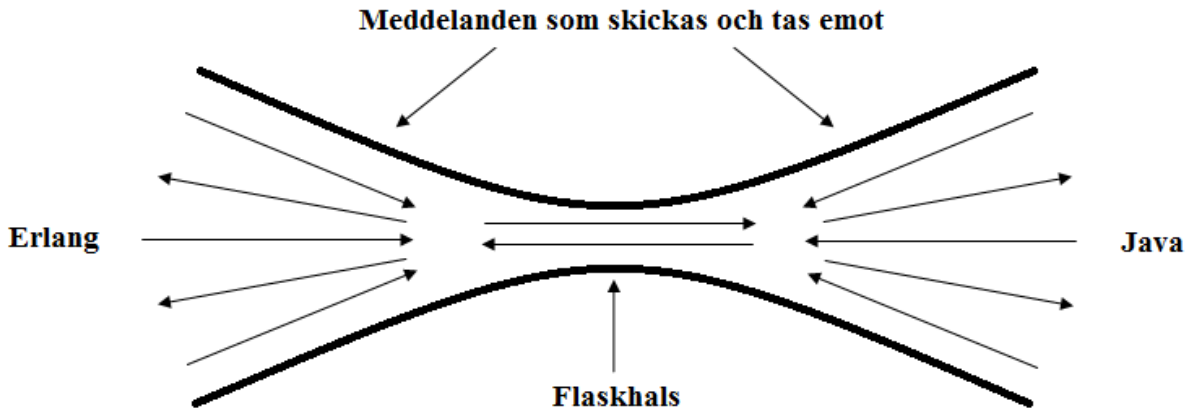
Figur 3. Ordningen kaniner (till vänster) och vargar (till höger) kontrollerar (och låser) rutor när de skall förflytta sig. Den aktuella kaninen står i ruta 5, och vargen i ruta 13.

7. Förslag till förbättringar

En viktig förbättring av systemet är att få bort flaskhalsen vid kommunikationen mellan Java och Erlang, se illustration i Figur 4. Denna flaskhals bidrar till prestandasänkning då meddelandeskickningen endast kan ske via en ”väg”. Om mer tid funnit skulle detta åtgärdats genom att lägga till fler ”vägar”.

Något som även var planerat att adderas till projektet vid mer tid var genetisk programmering. Vargarna och kaninerna hade då ärvt erfarenheter från sina föregångare och därmed blivit smartare i avseenden som att hitta mat och överleva.

Ytterligare en sak att lägga till i systemet är att användaren själv skall kunna mata in fler parametrar såsom maxålder för kaninerna/vargarna, hur snabbt gräset skall växa, ålder för könsmognad, hur många steg en varelse skall kunna röra sig, hur mycket varelserna skall kunna äta och dylikt.



Figur 4. Flaskhals som uppstår vid kommunikation mellan Erlang och Java

8. Reflektion

Vi har alla dragit stora lärdomar av detta projekt. En av de största lärdomar vi alla dragit är hur man integrerar två olika programmeringsspråk. Att få Erlang och Java att kommunicera ihop har inte varit det lättaste men vi har alla lärt oss något under processen. Det har även varit mycket nyttigt att lära sig strukturera upp arbetet på ett tydligare sätt än under tidigare projekt. I detta projekt har alla haft tydliga ansvarsområden och vi har både själva, och tillsammans, lärt oss strukturera upp arbetet och planera arbetsgången med hänsyn till tid och resurser. Dock har vi, trots specifika arbetsområden, hjälpt varandra och suttit och arbetat tillsammans i stor utsträckning. Hade alla suttit på egen hand hade integreringen av de olika delarna förmodligen tagit mycket längre tid.

Ytterligare en stor lärdom av arbetet har varit de Erlangkunskaper vi fått. Innan detta hade vi endast genomfört små laborationer i detta språk medan vi nu fått arbeta mycket mer grundligt med det. Även detta tog ett tag att lära sig och syntaxen var, efter att ha programmerat i imperativa och objektorienterade språk under ett år, lite ovan. Dock är detta något vi alla tror är bra att kunna så det har varit både kul och nyttigt.

Något som har varit lärorikt är även att se att man ibland under projektets gång måste omstrukturera stora delar av arbetet som man trodde var självklara. Vi hade från början en

tydlig plan över hur systemarkitekturen skulle se ut, planen var då att GUI:t skulle skrivas i Java medan både kartan och kaninerna/vargarna skulle vara programmerade i Erlang. Under första perioden av vårt arbete upptäckte vi dock att det var mycket svårt och tidskrävande att försöka använda sig av 2D-arrayer med information i varje ruta i Erlang. Så vi fick snabbt omstrukturera detta och flytta över denna stora bit till Java för att hinna med det som var planerat på den tid som fanns att tillgå.

Det som har tagit mest tid under arbetets gång har varit att läsa in sig på olika områden och att sedan testa sig fram för att få förståelse för det man försöker göra. Bland annat har detta krävts för att sätta sig in i hur man programmerar i Erlang samt hur man får Jinterface att fungera. I tidigare projekt vi deltagit i har det varit lättare att köra igång direkt jämfört med detta.

8.1. Önskade ändringar

Vad vi skulle göra annorlunda om vi gjorde om projektet nu är att läsa in oss lite mer på vad som var lämpligt att göra i de olika språken innan vi faktiskt delade upp arbetet i olika delar. Som tillvägagångssättet var nu valde vi att göra kartdelen i Erlang då vi tyckte att det var lämpligt att ha en server i det språket. Vad vi inte hade kollat upp var att det var svårt att tillämpa 2D-array på ett smidigt sätt för vårt behov i detta språk, en mycket viktig del för själva kartan.

Något som vi även tror kunnat vara lämpligt är att arbeta i par istället för en och en. Som arbetet varit nu har det fungerat att alla jobbat med olika delar, men i ett större projekt är det nog lättare och effektivare om man kan diskutera med en partner och då även hitta fel medan man programmerar. Man är då även två som är insatta på samma del av projektet vilket vi tror kan vara positivt.

9. Installation och fortsatt utveckling

Under arbetet har Eclipse Indigo Service Release 1 & 2 använts för att programmera i. Detta för att det varit smidigt att använda en IDE där kompilering och testkörning kan göras direkt utan att behöva återvända till terminalen mellan varje körning. Av Erlang har version R15B01 använts, och av Java version 1.6.

Det versionshanteringssystem som använts är GitHub och arbetet finns tillgängligt under länken: https://github.com/theKGS/pop_raw.git

Uppdelningen av katalogstrukturen har gått till så att Java- och Erlangdelen har varsin mapp. Under dessa ligger sedan olika mappar i Javadelen med filer för GUI:t och kartan. I Erlangmappen ligger flertalet moduler för bland annat kaniner och vargar m.fl.

För att kompilera programmet finns det en makefil och en jarfil. För att starta programmet körs endast jarfilen som då även kompilerar Erlangdelen med hjälp av makefilen.

9.1. Dokumentation och testning

När det gäller testning så har JUnit och EUnit använts. Testning av GUI:t har skett genom att prova trycka på alla knappar och funktioner som finns och se att det beter sig som det ska. GUI:n är mycket svåra att testa och det har ej funnits tid till att sätta sig in i detta. Det har även varit för svårt att testa de delar i Java som hanterar trådar i kartan på annat sätt än att studera simulationen och se att allt beter sig som väntat.

Dokumentation finns genererad med hjälp av Javadoc och Edoc. Dokumentationen är skapad och genererad av gruppens medlemmar och finns sedan tillgänglig som en html-sida för användaren att läsa.