UPPSALA UNIVERSITET

MicroQ

Processorienterad programmering (1DT049) våren 2012.
Slutrapport för grupp 12

Björn Berggren 910705-0057
Marcus Enderskog 901025-2238
Simon Evertsson 911202-3370
Nanna Kjellin Lagerqvist 890103-1404

2012-05-24

[Skriv sammanfattningen av dokumentet här. Det är vanligtvis en kort sammanfattning av innehållet i dokumentet. Skriv sammanfattningen av dokumentet här. Det är vanligtvis en kort sammanfattning av innehållet i dokumentet.]

Innehållsförteckning

1 Inledning

Här ska vi skriva vår inledning. Innehåller vanligtvis bakgrund, syfte, målsättning, problemformulering och avgränsningar.

1.1 Bakgrund

Projektet MicroQ har en nära koppling både till tekniken och till datavetenskapen men också till studenters vardag. (Här måste vi skriva mera!)

1.2 Syfte

Det övergripande syftet med rapporten är att få kunskap och förståelse för hur en kö till mikrovågsugnar i studenters lunchrum (i fortsättningen kallad *mikrokö*) fungerar.

Vi vill undersöka hur olika parametrar så som antal mikrovågsugnar, antal studenter och hur lång tid maten ska värmas påverkar mikrokön.

1.3 Målsättning

Målet med rapporten är att ge ett underlag till att skapa en bättre miljö i studenters lunchrum. Genom ökat förståelse för hur en mikrokö fungerar kan eventuellt studenternas väntetider minimeras och utnyttjandet av varje mikro optimeras.

1.4 Problemformulering

För att besvara vårt syfte använder vi följande frågeställningar:

- Hur beror den genomsnittliga väntetiden i mikrokön av antalet mikrovågsugnar, hur ofta nya studenter anländer till kön samt hur länge varje student värmer sin mat?

- Hur länge står en mikro i genomsnitt tom, baserat på de givna parametrarna; hur många mikros som finns, hur ofta nya studenter anländer till kön samt hur länge varje student värmer sin mat?

1.5 Avgränsningar

Vi har valt att använda en simulering av en mikrokö för att besvara våra frågeställningar. Denna simulering använder givna parametrar som matas in av användaren innan simuleringen startas. Dessa parametrar är

- Hur ofta nya studenter anländer till kön. Egentligen sannolikheten att en ny student anländer, anges i procent

- Hur ofta en student byter tillstånd. Med tillstånd menas om studenten är närvarande och redo att värma, eller frånvarande.

- Väntetid, det vill säga så länge som en student kan vara frånvarande utan att förlora sin plats först i kön.

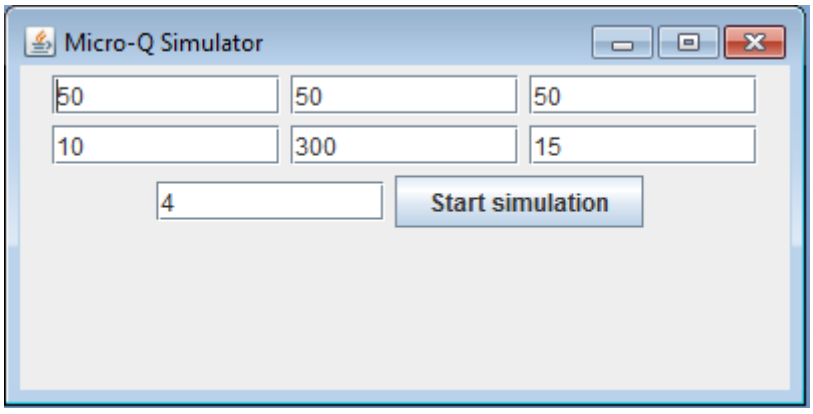
- Ett tidsintervall inom vilken tiden för att värma varje matlåda sedan slumpas fram, till exempel 2,00 till 5,00 minuter.

- Antal mikrovågsugnar

2 MicroQ

Här beskriver vi hur systemet kan användas.

2.1 Funktionalitet



Figur 2-1, Ruta där användaren får mata in parametrar till simuleringen.

Systemet kan användas för att skapa en simulering av en mikrokö. Användaren får själv mata in ett antal parametrar (se avsnitt *1.5 Avgränsningar*, ovan) innan simuleringen startas(*Fig. 2-1)*. När simuleringen startas kan användaren följa systemets exekvering visuellt dels genom en grafisk representation av lediga och upptagna mikrovågsugnar och kön av studenter och dels genom textbaserade meddelanden som beskriver systemets olika tillstånd.

2.2 Användargränssnitt

Systemet har både ett grafiskt användargränssnitt (GUI[1]) och ett terminalfönster för övervakning av tillstånd.

2.2.1 Grafik

Det grafiska användargränssnittet är helt och hållet byggt i java. Detta eftersom det finns stöd för grafik i form av Javas färdigskrivna grafikbibliotekbibliotek och verktyg för att tillverka GUI-layouter till IDE:n[2] Eclipse.

[1] Graphical User Interface (http://en.wikipedia.org/wiki/Graphical_user_interface)
[2] Integrated Development Environment (http://en.wikipedia.org/wiki/Integrated_development_environment)

När programmet startar så visas en ruta där användaren ombeds att mata in simuleringsparametrar. När användaren har matat in alla parametrar klickar denne på knappen "Start Simulation" för att starta simuleringen och stänga ner inmatningsrutan.

När simuleringen startas så öppnas en ny ruta vilken representerar simuleringen. I denna ruta visas ett antal cirklar som representerar mikrovågsugnar, med tillhörande tid i formatet MM:SS . Antalet cirklar är lika många som det antal mikrovågsugnar användaren angav i inmatningsrutan. Cirklarna ändrar färg beroende på om de är lediga (grön färg) eller upptagna (röd färg). Tiden som visas är den värmtid som återstår för tillfället. Då tiden visar "00:00" är mikron ledig.

Vidare visas en representation av kön i form av rektanglar med en unik siffra per rektangel. Rektanglarna är uppradade under mikrocirklarna och representerar de X första matlådorna i kön. Den rektangel som befinner sig längst till höger är den matlåda som befinner sig först i kö. När en mikro blir ledig, och om studenten som lådan tillhör är närvarande i lunchrummet, plockas lådan bort ur kön och en ledig mikro blir upptagen. Skulle inte studenten hinna tillbaka till lunchrummet inom den väntetid som användaren angivit så hamnar lådan automatiskt sist i kön.

När alla studenter har värmt sin mat beräknas simuleringen vara avslutad och en ruta med diverse simuleringsdata visas. När användaren är färdig med datat och vill avsluta programmet klickar denne på knappen "Exit".

2.2.2 Erlang

Erlangkonsolen kommer enbart visa meddelanden då eventuella fel uppstår. Till exempel om kontakten med Java skulle upphöra. För övrigt så visas endast meddelanden under uppstarten av systemet samt då programmet avslutas, som bekräftar att allt genomförts korrekt.

3 Programmeringsspråk

Vi har använt två programspråk. Detta val gjorde vi dels för att lära oss att skriva ett program i två olika språk och få delarna att fungera tillsammans, men också för att erlang och java har olika bra stöd för implementation av olika delar av programmet.

3.1 Fördelar

Vi har mest känt att det varit fördelar med att arbeta i två språk. Det gav en naturlig uppdelning av arbetsuppgifter inom gruppen och på så vis har varje persons kompetens utnyttjats, samtidigt som gruppen fått en chans att öva extra på kommunikation då det varit ett gemensamt ansvar att alla gruppmedlemmar hålls uppdaterade om vad som händer i projektets olika delar.

3.2 Nackdelar

Eftersom vi valde att implementera programmet i två språk så blev vi tvungna att hantera kommunikationen däremellan. När vi bestämde oss för just språken erlang och java så hade vi tänkt använda Erlide[3], men vi fick det helt enkelt inte att fungera och eftersom det riskerade att ta för mycket tid och fokus från kärnan i projektet så löste vi det genom att utveckla erlangdelen separat.

[3] Erlide – the IDE. Powered by Eclipse (http://erlide.org/)

3.3 Erlang

Erlang är ett funktionellt högnivåspråk utvecklat av telekomföretaget Ericsson i slutet av 1980-talet. ”*Erlang fungerar bra som bas för system som ska vara felsäkra och fungera hela tiden, vara skalbara och distribuerade och hantera många samtidiga händelser*”.[4] (Larsson, 2009)

Vi valde att använda erlang till de delar av programmet som består av flera processer. Erlang har actormodellen[5] som är en modell för asynkron kommunikation och implementerar en mycket kodeffektiv[1] struktur för att skicka meddelanden mellan processer. Vidare har erlang lättviktsprocesser, *”Erlang processes are much more lightweight than OS threads”* [6], vilket gör det enkelt att skapa nya processer som får sköta olika delar av programmet. (Marklund, Process Oriented Programming (1DT049), Generators, Coroutines, Actors and Erlang, 2012)

3.4 Java

Java är även det ett högnivåspråk men är till skillnad från erlang är det inte ett funktionellt utan ett imperativt objektorienterat programmeringsspråk. Java har inbyggt stöd för synkronisering[7] av delade resurser vid användande av trådar[8](Här kanske vi måste förtydliga?), samt för att implementera ett grafiskt användarinterface. I projektet använder vi java till de delar av programmet som hanterar kön, mikrovågsugnar samt tidsmätning. (Lampka & Marklund, 2012) (Oracle.com)

Vi valde att använda java eftersom språket tillhandahåller stöd för automatiserade tester, JUnit[9], samt automatiserad generering av dokumentation, Javadoc[10]. Språket innehåller också många färdigskrivna bibliotek som vi använt till exempelvis köhantering och de grafiska delarna i programmet. Ericsson har även skrivit ett gränssnitt[11] mellan Erlang och Java vilket möjliggjorde kommunikation mellan Java och Erlang. Detta var en förutsättning för att kunna implementera programmets delar i två skilda språk. (JUnit.org) (Oracle.com) (Ericsson AB, 2012)

4 Systemarkitektur

Systemets olika delar samt samverkan däremellan.

4.1 Arkitektur java

Java kommunicerar med erlang via mailboxes[12]. Intern kommunikation mellan trådarna i javadelen av programmet sker med hjälp av shared resources[13] i form av statiska variabler och metodanrop.

4.1.1 SimulationMain

[4] http://www.idg.se/2.1085/1.213789/svensk-sprakdoldis-gor-succ%C3%A9
[5] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang s. 14-17*
[6] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang s. 18*
[7] http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html
[8] kai.lampka@it.uu.se, karl.marklund@it.uu.se, Uppsala Universitet 27 01 2012, *Chapter 4: Threads*
[9] http://kentbeck.github.com/junit/javadoc/latest/index.html?overview-summary.html
[10] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[11] http://www.erlang.org/doc/apps/jinterface/index.html
[12] mailboxes
[13] Shared resources

SimulationMain är det första som körs i java när programmet startar och det är här den grafiska delen av simuleringen körs. Till att börja med så startas en ny InputBox-tråd där användaren får mata in simuleringsparameterar. SimulationMain väntar sedan på en semafor[14] som släpps efter att användaren klickat på "Start Simulation"-knappen. Efter att semaforen har tagits så fortsätter exekveringen och en initiering av GUI:t startas. Där skapas simuleringsfönstret samt en ny QueueRepresentationinstans. Efter att initieringen är färdig startas Simulationtråden och en instans av SimulationData skapas.

Efter det hamnar SimulationMain i sin ritningsloop. Loopen kommer köras så länge som simuleringen är igång och kontakt mellan java och erlang existerar. I loopen körs drawmetoden ungefär 30 gånger i sekunden. Drawmetoden ritar ut alla objekt som skall synas i fönstret. Det vill säga alla Microwaveobjekt i microwavesfältet och de X första matlådorna från QueueRepresentation.

När simuleringen är färdig så bryts loopen och simuleringsdatat från SimulationData visas på skärmen. Om loopen bryts av att kontakten med erlang avbrutits visas istället en ruta med meddelandet "Connection Lost". Efter att rutan stängts ner så avslutas programmet.

4.1.2 InputBox

InputBox visar ett fönster där användaren får mata in parametrar som krävs för simuleringen. När användaren klickar på "Start Simulation" så skapas så många Microwave objekt som användaren angivit. Sedan skapas Simulation-tråden och de inmatade parametrarna skickas med i konstruktorn. Slutligen så släpper InputBox setup_done-semaforen så att SimulationMain kan fortsätta sin exekvering.

4.1.3 SimulationData

Samlar data under körningen för att ge användaren väsentlig information vid simuleringens slut, så som total kötid för varje enskild student samt ett genomsnitt för samtliga. Den sammanställda informationen visas i ett nytt fönster sist i programmet.

4.1.4 Microwave

Den grafiska representationen av mikrovågsugnarna. Uppdaterar sin kvarvarande tid och färg varje sekund. Dess drawmetod kallas då mikron skall ritas ut.

4.1.5 QueueRepresentation

4.1.6 Simulation

I Simulations konstruktor sätts parametrarna som angetts av användaren. Simulation skapar även en ny instans av QueueManager samt startar en StudentSpawner[15]-tråd. När SimulationMain sedan startar Simulationtråden påbörjas simuleringen. (Marklund, 2012)

Simulation kommer då kalla på QueueManager's readyCheckmetod så länge som det finns någon student i kön och det finns en ledig mikro. Detta kommer genomföras tills det att antalet studenter som har värmt klart sin mat är lika med antalet studenter som användaren angivit.

[14] kai.lampka@it.uu.se, karl.marklund@it.uu.se, Uppsala Universitet 18 02 2012, *Synchronization – coordinating acces to shared resources*
[15] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang s. 30*

När alla studenter värmt färdigt, skickar Simulation ett terminatemeddelande till erlang och bryter ritningsloopen i SimulationMain.

4.1.7 ConnectionCheck

Denna tråd kontrollerar att uppkopplingen mellan java och erlang fortfarande existerar. Skulle den inte göra det så sätter den fältet connected i SimulationMain till false för att bryta ritningsloopen.

4.1.8 StudentSpawner

Klassen StudentSpawner har till uppgift att starta upp Communicatortrådar som skapar nya Studentobjekt. En sådan tråd skapas varje sekund med en viss procentuell sannolikhet, inmatad av användaren. Nya trådar fortsätter skapas tills att antalet skapade studenter är lika många som det värde användaren angivit.

4.1.9 QueueManager

Denna klass lagrar kön och tillhandahåller metoder för att lägga till och plocka ur studenter ur kön. Simulation kallar på metoden readyCheck när en mikro är ledig och en students tillstånd måste bestämmas. QueueManager startar då en ny Communicatortråd som ska kontrollera att studenten är redo. Klassen tillhandahåller även SimulationData de klockslag som behövs för att sammanställa de slutgiltiga resultaten.

När en ny student, eller en gammal student som inte var redo, skall läggas i kön så kallar Communicator på addStudent-metoden samt startar eller återupptar tidräkningen för denne beroende på om den är nyskapad eller har köat förut. Studenten läggs då sist i kön och semaforen queue_sem i Simulation släpps för att meddela Simulation om att det finns ett Studentobjekt i kön.

4.1.10 Communicator (java)

Instanser av Communicator är de trådar som sköter det mesta av kommunikationen med erlang. Communicator kan skapas på två sätt. Antingen då en ny studentprocess ska spawnas i erlang eller då en mikro blivit ledig och en students tillstånd söks. En mailbox med ett unikt id för varje ny instans av Communicator skapas så att erlang vet var den ska skicka svaret.

Då en ny student ska spawnas så skickas meddelandet {***mailbox-id***, ***'new_client'***, ***"Switch rate"***} till erlang.Variabeln ***mailbox-id*** är det unika id som denna Communicator-instans fått tilldelat, ***'new_client'*** är meddelandet som gör att erlang vet att en ny tillståndsprocess ska skapas. ***"Switch rate"*** är den procentuella sannolikhet att en student ska byta tillstånd, vilket angivits av användaren.

Communicator väntar sedan på svaret från erlang som innehåller den nyskapade tillståndsprocessens PID. StudentSpawner:s räknare över antalet skapade studenter ökas sedan och den nya studenten läggs till i kön i QueueManager.

Om en befintlig students tillstånd ska kontrolleras så skickas meddelandet {***mailbox-id***, ***'ready'***, ***PID***} Variabeln ***mailbox-id*** är åter igen det unika id som denna Communicator-instans fått tilldelat, ***'ready'*** är meddelandet som gör att erlang vet att en client_state-process tillstånd ska kontrolleras. ***PID*** är PID:t till client_state-processen som hanterar studentens tillstånd.

Communicator väntar på svar från erlang så lång tid som användaren matat in. Om svaret anländer innan tidsgränsen nåtts så sätts timern i ett Microwaveobjekt till studentens värmtid och startas.

Communicator sover sedan under värmtiden och skickar sedan ett meddelande till erlang så att studentens client_state-process tas bort.

Om svaret inte kommit inom tidsgränsen så läggs Studentobjektet tillbaka i kön och semaforen microwave_sem släpps så att Simulation får veta att en mikro är ledig. Efter det så skickas ett meddelande från Communicator till Simulation att ytterligare en student har värmt färdigt sin mat och att en mikro är ledig.

4.2 Arkitektur erlang

Erlang kommunicerar både internt (mellan processer) och externt (mellan erlang och java) med hjälp av message passing[16]. Om en erlangprocess vill skicka ett meddelande till java så måste den känna till vilken mailbox den ska skicka till samt vilken javanod som den tillhör. (Marklund, 2012)

För att skicka meddelanden internt mellan två erlangprocesser så räcker det att känna till mottagarprocessens PID[17], men om processen ska kunna svara tillbaka så måste avsändarprocessens PID skickas med (förslagsvis i en tupel[18], för att även kunna skicka med ett meddelande och en eventuell atom[19]). (Marklund, 2012)

4.2.1 init

Funktionen init i modulen[20] jcom är det första som körs i erlang. Den upprättar kommunikationen med java och spawnar en ny process som kör funktionen connection_lost i modulen connection_check. Därefter kallas communicator.

4.2.2 connection_check

Modulen connection_check har en funktion vars enda uppgift är att kontrollera att uppkopplingen mellan erlang och java fortfarande fungerar. Om uppkopplingen någon gång skulle tappas så stänger den ner erlang delen i programmet och skriver ut ett felmeddelande.

4.2.3 communicator (erlang)

Funktionen communicator i modulen jcom har som huvuduppgift att skicka och ta emot meddelanden, både till java och internt. Beroende av vilka parametrar den tar emot ifrån java så kommer den att adressera olika erlangprocesser med olika meddelande och på så vis styra vad processerna ska göra, exempelvis byta tillstånd (ready eller away) och terminiera.

4.2.4 spawner

Om communicator får ett 'new_client'-meddelande från java så kallar communicator på funktionen spawner. Dess uppgift är att spawna en client_state-process samt en state_switcher-process som

---

[16] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang s. 14-17*
[17] karl.marklund@it.uu.se, Uppsala Universitet 24 01 2012, *The Process Concept and Inter Process Communication s. 26*
[18] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang s. 28*
[19] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Generators Couroutines, Actors and Erlang s. 27*

[20] karl.marklund@it.uu.se, Uppsala Universitet 20 03 2012, *Erlang: modules, TDD with EUnit, process supervision s. 7*

binds till den nya client_state-processen. Sedan returneras PID för den nya client_state-processen tillbaka till communicator.

4.2.5 state_switcher

Modulen state_switcher är en process som vars enda uppgift är att skicka 'switch'-meddelanden till den client_state-process som den är bunden till så att client_state byter tillstånd. Ett 'switch'-meddelande skickas enbart om ett slumpmässigt genererat heltal mellan 1 och 100 är mindre än eller lika med det tal som användaren matat in som "Switch rate".

4.2.6 client_state

Den metod som representerar och håller en students nuvarande tillstånd är client_state. Den arbetar i en loop som tar emot meddelanden från communicator och sin state_switcher-process. Om den befinner sig i sitt redotillstånd och får ett 'ready'-meddelande från communicator så svarar den communicator direkt att den är redo. Om den däremot inte skulle vara i sitt redotillstånd så skickas svaret när den nästa gång får ett 'switch'-meddelande från sin state_switcher-process.

5 Samtidighet (concurrency)

Här diskuteras de modeller vi använt för concurrency.

5.1 Modeller

Vi använder message passing för kommunikation mellan erlangprocesser och mellan erlang och java. Fördelen med detta är att vi slipper hantera delat minne, som hade varit ett alternativt sätt att kommunicera på. Tack vare det så kan vi utnyttja parallellism maximalt i erlangprocesserna.

I erlang använder vi flera processer och i java använder vi istället trådar. Detta valde vi att göra eftersom detta är vad respektive programspråk har bäst stöd för i form av exempelvis färdigskriva bibliotek. Trådarna i java använder delat minne och därför måste vissa delar av programmet exekvera seriellt.

5.2 Samtidighet

En av tankarna bakom detta projekt var att öva oss på att programmera concurrent, det vill säga att använda fördelarna med samtidighet. Detta kunde vi utnyttja i erlangprocesserna samt SimulationMain, StudentSpawner, Communicator och Simulation i java. Tack vare att erlangprocesserna inte använder delat minne så kan de exekvera helt parallellt.

Trådar som skapas i java använder gemensamma resurser (delat minne) och därför måste vissa delar av den koden synkroniseras för att bli threadsafe[21], därför exekveras en del av programmet seriellt.

5.3 Synkronisering

Vi använder synkronisering för hantera mikrovågsugnarna. Mikrovågsugnarna representeras av en semafor vars värde är lika med antalet lediga mikrovågsugnar. Det är nödvändigt med denna synkronisering eftersom en student representeras som en tråd i java och när flera studenter skapas så har de tillgång till gemensamma resurser. (Lampka & Marklund, 2012)

[21] Threadsafe

5.4 För- och nackdelar

Fördelar med denna implementation är bland annat att GUI:t är skapas väldigt enkelt då alla bibliotek som används är färdigskrivna och inbyggda i java. Smidig synkronisering av åtkomsten till mikrovågsugnarna. Erlangdelarna blir väldigt små och överskådliga då inga tunga beräkningar utförs utan allt sköts nästan uteslutande genom meddelandeskickning.

En nackdel är den lilla prestandaförlust vi får då många meddelanden måste skickas mellan java och erlang.

Tester med Eunit[22] har varit svåra att skriva i den implementation vi valt eftersom många av funktionerna inte returnerar något värde.

5.5 Alternativ

Ett alternativ till denna implementation hade varit att skriva all kod i ett språk. Då hade problem som kan uppstå då man jobbar mellan två olika språk kunna undvikas.

5.6 Deadlocks

Som tidigare nämnts så använder vi oss av semaforer för att sköta åtkomsten till en mikro. En deadlock skulle kunna uppstå om någon Communicatortråd oväntat skulle dö efter att mikrosemaforen plockats. Då skulle en mikro alltid fortsätta vara upptagen även om ingen värmer sin mat i den.

Ett annat tillfälle då körningen skulle låsas är om InputBoxtråden skulle dö innan den har släppt sin setup_done-semafor som SimulationMain väntar på.

Det är alltid svårt att helt undvika deadlocks, och det är även svårt att bevisa frånvaron av deadlocks.

6 Algoritmer och datastrukturer

Förklara och diskutera viktiga algoritmer och datastrukturer mer detaljerat, gärna med hjälp av figurer. Här kan även korta kodexempel ges.

7 Förslag på förbättringar

Här beskrivs sådant ni vet kan göras bättre men inte haft tid att prioritera.

Något som vi skulle vilja göra om vi hade haft mer tid är att visa tillstånden (till exempel ready eller away) för de studenter vars matlådor ligger bland synliga matlådorna i GUI:t. Det hade då ytterligare förtydligat vad som händer under simuleringens gång.

Detta hade kunnat implementeras genom att varje studentprocess i erlang skickat ett meddelande till en ”lyssnartråd” i java varje gång den bytte tillstånd. Javatråden skulle sedan uppdatera ett objekt som representerar en students tillstånd, vilket sedan hade ritats ut av GUI:t.

[22] http://www.erlang.org/doc/apps/eunit/chapter.html

8 Reflektion

Vad har ni lärt er till följd av det genomförda projektet. Vad har varit oväntat svårt? Vad skulle ni gjort annorlunda om ni fick börja om? Här kan ni beröra både både tekniska aspekter och samarbetet i gruppen.

Att sätta upp och använda ett repository har varit onödigt svårt. Nu vet vi hur vi ska göra det på ett bättre fungerande sätt. Det visade sig att vi hade missförstått hur själva fildelningen genom repositoryt går till. Vi använde Git.

9 Installation och utveckling

Frågor som bör besvaras i detta avsnitt:

9.1 Versioner

Vilka versioner av till exempel Erlang, Java, Python etc har använts?

9.2 Ladda ner

Koden finns tillgänglig på GitHub.

9.3 Katalogstruktur

Kort beskrivning av systemets katalogstruktur

9.4 Kompilering

Hur kompileras systemet?

9.5 Automatiserade tester

Vilket stöd för automatiserad testning finns och hur används det, till exempel JUnit (Java), EUnint (Erlang)?

9.6 Automatiserad generering av dokumentation

Vilket stöd för automatiserad genering av dokumentation finns det och hur används det, till exempel Doxygen (C, C++, C#, Fortran, Java, Objective-C, PHP, Python), EDoc (Erlang), Pydoc (Python), Javadoc (Java)?

Vi har använt EDoc och Javadoc som både funkar bra och är smidiga att använda när man väl vant sig.

9.7 Start av systemet

Hur startas systemet?

make start(fungerar inte ännu) eller startup.bat.

10 Litteraturförteckning

Ericsson AB. (den 01 04 2012). *erlang.org.* Hämtat från jinterface Reference Manual: http://www.erlang.org/doc/apps/jinterface/index.html den 24 05 2012

JUnit.org. (u.d.). Hämtat från JUnit API: http://kentbeck.github.com/junit/javadoc/latest/index.html?overview-summary.html den 23 05 2012

Lampka, K., & Marklund, K. (den 27 01 2012). Process Oriented Programming (1DT049), Chapter 4: Threads. Uppsala.

Lampka, K., & Marklund, K. (den 18 02 2012). Process-oriented Programming (1DT049), Synchronization --coordinating access to shared resources. Uppsala.

Larsson, P. (den 23 02 2009). *Svensk språkdoldis gör succé, Computer Sweden*. Hämtat från IDG: http://www.idg.se/2.1085/1.213789/svensk-sprakdoldis-gor-succ%C3%A9 den 22 05 2012

Marklund, K. (den 23 03 2012). Process Oriented Programming (1DT049), Erlang: modules, TDD with EUnit, process supervision. Uppsala.

Marklund, K. (den 20 03 2012). Process Oriented Programming (1DT049), Generators, Coroutines, Actors and Erlang. Uppsala.

Marklund, K. (den 24 01 2012). Process Oriented Programming (1DT049), The Process Concept and Inter Process Communication. Uppsala.

Oracle.com. (u.d.). Hämtat från Javadoc Tool Home Page: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html den 23 05 2012

Oracle.com. (u.d.). Hämtat från The Java Tutorials, Synchronized Methods: http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html den 24 05 2012