NTNU
Det skapende universitet

Verktøyintegrasjon i åpen kildekode-utvikling

Hogne Håskjold

Master i informatikk
Oppgaven levert: Juni 2006
Hovedveileder: Hallvard Trætteberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Forord

Denne rapporten er en masteroppgave gitt ved Institutt for datateknikk og informasjonsvitenskap ved NTNU. Oppgaven ble utformet i samarbeid med veileder Hallvard Trætteberg. Denne rapporten og forarbeidet som ligger til grunn for denne ble utført over 3 semestere 2005/2006.

Jeg vil rette en takk til min veileder Hallvard Trætteberg for konstruktive tilbakemeldinger og annet hjelp med dette arbeidet, min medstudent Arne Mathisen for produktiv dialog rundt implementasjonsarbeidet, samt alle utviklerne som svarte på min henvendelse over epost med spørsmål rundt åpen kildekode-utvikling.

Sammendrag

Denne oppgaven ser på verktøyintegrasjon i åpen kildekode-utvikling. Dette utviklingsparadigmet viser seg å i liten grad ha adoptert tradisjonelle programvareutviklingsmetoder. I stedet står sosiotekniske prosesser og verktøy som underbygger disse for effektiv kommunikasjon, læring og utvikling i fokus. Kunnskap blir skapt og delt i åpen kildekode-miljø ved hjelp av prosesser og tekniske løsninger som muliggjør effektiv læring og refleksjon blant deltagerne.

Ulike typer integrasjon kjent fra tradisjonell utvikling har blitt evaluert, og av disse blir presentasjonsintegrasjon funnet å være mest hensiktsmessig fordi denne kan potensielt øke effektiviteten i utviklingsarbeidet ved å tilby et felles grensesnitt til en rekke uavhengige verktøy samtidig som det ikke kommer i veien for kravene om portabilitet, fleksibilitet og åpenhet som er kritisk i åpen kildekode-utvikling. Data- og kontroll-integrasjon viste seg å være mindre hensiktsmessig fordi de ikke tillot den nødvendige fleksibiliteten og samhandlingsevnen med eksterne verktøy. Prosessintegrasjon i åpen kildekode-utvikling kan sies å til en viss grad oppnås gjennom de sosiotekniske prosessene som blir innskrevet i verktøyene som benyttes.

I evalueringen av egnede rammeverk ble Eclipse valgt på grunn av dets sterkt modulære og fleksible oppbygning, tilgjengelighet, aktive utvikling og “mind share”, samt at det støttet presentasjonsintegrasjon uten å sette føringen i forhold til de andre mindre egnede integrasjonstypene. I dette rammeverket ble det implementert et programvaretillegg som tillot lagring av synkron kommunikasjon i kontekst av koden slik at denne mer effektivt kunne innhentes og reflektert rundt. Testing viste at dette kunne ha potensiale så lenge det krevde lite ekstra innsats å legge inn informasjonen samtidig som man tillot fleksibilitet i koblingen. Presentasjonen av denne ekstra informasjonen viste seg å best kunne integreres ved å bruke eksisterende elementer i Eclipse, samt å klart vise i grensesnittet hvilke filer som var tilkoblet informasjon. Videre implementasjon og testing er nødvendig for å kunne konkludere med en mer solid vurdering av nytteverdien av en slik integrasjon, men oppgaven kan fungere som et utgangspunkt for videre arbeid.

Innhold

1 Introduksjon **1**

2 Åpen kildekode **3**

2.1 De første *hackerne* 4
2.2 Unix 5
2.3 ARPANET 6
2.4 Framveksten av proprietær programvare 7
2.5 The Free Software Foundation 8
2.6 Linux og World Wide Web 9
2.7 Diskusjon 10

3 Programvareutvikling **13**

3.1 Systemutvikling 13
3.1.1 Prosessmodeller 14
3.2 Prosesser i åpen kildekode-utvikling 18
3.2.1 *The Open Source Spiral Model* 18
3.2.2 Samarbeid i åpen kildekode-utvikling 21
3.3 Diskusjon 23

4 Kunnskap og programvareutvikling **27**

4.1 Mekanismer for “knowledge-creation” 28
4.2 Kunnskap i tradisjonell programvareutvikling 30
4.3 Kunnskap i åpen kildekode-utvikling 33

4.3.1 “Re-experience” gjennom redusert kompleksitet og “transactive group memory” 33
4.3.2 “Re-experience” gjennom veiledning, åpenhet og deltagelse 34
4.3.3 “Re-experience” gjennom asynkrone kommunikasjonskanaler og virtuell eksperimentering 34
4.4 Diskusjon 35

5 Verktøy i programvareutvikling 38
5.1 Verktøy i tradisjonell programvareutvikling 38
5.2 “Software informalisms” 40
5.2.1 “Community communications” 40
5.2.2 Nettbasert informasjon 42
5.2.3 Feilhåndteringssystemer 42
5.2.4 Dokumentasjon 42
5.3 Verktøyintegrasjon 43
5.4 Diskusjon 46

6 Metode 49
6.1 Forskningstradisjoner innen systemutvikling 49
6.2 Forskningsprosessen 51
6.3 Diskusjon 54

7 Forslag til verktøyintegrasjon 57
7.1 *community communications* 60
7.2 Kontekstuell informasjon 61
7.3 Reduksjon av kompleksitet 63
7.4 Nettbasert informasjon 64

8 Valg av utviklingsrammeverk 65
8.1 Eclipse 66
8.1.1 Programvaretilleggsarkitektur 66

8.1.2 Platform Runtime 67
8.1.3 Workspaces 67
8.1.4 Workbench og UI Toolkits 67
8.1.5 The Eclipse Communication Framework (ECF) 68
8.2 Andre aktuelle rammeverk 69
8.2.1 Emacs 69
8.2.2 NetBeans 70
8.3 Kommunikasjonsverktøy 71
8.3.1 Jabber - teknisk oversikt 72

9 Design - kommunikasjonsstøtte i et utviklingsrammeverk 74
9.1 Kommunikasjonsverktøy - *chat* 75
9.2 Personlig “issue tracker” 76
9.3 Kontekstuell informasjon 77
9.4 Scenario I - Bruk av personlig “issue tracker” 80
9.5 Scenario II - Få hjelp og yte hjelp 81
9.6 Scenario III - Feilsøk 83

10 Implementasjon 88

11 Prototype - testing og evaluering 91
11.1 Testoppgaver 92
11.1.1 Oppgavetekst 92
11.1.2 Testprosedyre 93
11.2 Testutførelsen 93
11.3 Resultater 94
11.3.1 Grensesnittendringer 95
11.3.2 Logiske endringer 98
11.4 Diskusjon 99

12 Undersøkelse om verktøyintegrasjon **101**

12.1 Diskusjon 101

13 Konklusjon **104**

13.1 Videre arbeid 106

14 Vedlegg **112**

Figurer

6.1 The General Methodology of Design Research 52

9.1 “Issue tracker”-arkfanen 80
9.2 Dialog som viser detaljer for en valgt feilrapport 81
9.3 Dialog som viser kontekstuell informasjon for en valgt fil . . . 82
9.4 Chatvindu 83
9.5 Diskusjon om koden 84
9.6 Dialog som viser kontekstuell informasjon for en valgt fil . . . 85
9.7 Dialog som viser kontaktliste og chatroom-liste 86
9.8 Dialog som viser mouseover for en kontakt 86
9.9 Dialog som en typisk samtale 87

10.1 Dialog som viser lagrede diskusjoner for den aktuelle filen . . . 89
10.2 Arkfane som viser lagrede diskusjoner for ’LogSession.java’ . . 90

11.1 Ny dialog for lagrede chats under properties 96
11.2 Nytt design av chatvindu 97

Kapittel 1

Introduksjon

W. Scacchi [41, s. 19] fant i sin studie "Understanding the Requrements for Developing Open Source Software Systems" at åpne kildekode-prosjekt i liten grad har adoptert tradisjonelle programvareutviklingsmetoder. Funksjonelle og ikke-funksjonelle krav i åpen kildekode-prosjekt blir fremsatt, analysert, spesifisert og validert gjennom en rekke nett-baserte verktøy som W. Scacchi kaller *software informalism*. Scacchi fokuserte på kravspesifisering, men de verktøyene han kategoriserer under *software informalism* er også relevant for andre deler av utviklingsprosessen og jeg vil se på disse med fokus på implementering.

Denne oppgaven vil se på hvordan *Software informalisms*, og da spesielt *Community communications* kan integreres tettere i et egnet utviklingsrammeverk. "Community communications" innbefatter en rekke elektroniske kommunikasjonsverktøy: Nettbaserte forum, epost og epost-lister, nyhetsgrupper, og lynmeldinger (*Instant Messaging*). En viktig egenskap i denne sammenhengen er om kommunikasjonen er synkron eller asynkron. Synkrone kommunikasjonskanaler er ikke spesielt hensiktsmessig for oppgaveorientert kommunikasjon, men er heller egnet til å raskt komme i kontakt med andre som har relevant kunnskap og for sosial interaksjon uavhengig av sted og tid. Asynkrone kommunikasjonskanaler derimot er bedre egnet til å bearbeide kunnskap og til refleksjon rundt denne [16, s. 19].

Oppgaven vil se på hvilke typer integrasjon og hvilke verktøy som er hensiktsmessig å integrere for åpen kildekode-utvikling. Verktøyintegrasjon har lenge vært i fokus i tradisjonell programvareutvikling. Jeg vil sette denne utviklingsmetodologien opp mot åpen kildekode-utvikling for å sette sistnevnte inn i en større sammenheng, samt for å kunne bruke det arbeidet som er gjort innen tradisjonell verktøyintegrasjon som et utgangspunkt for

mitt arbeid. Hemetsberger og Reinhardt [16, 1] beskriver hvordan kunnskap blir skapt og delt i åpen kildekode-miljø ved hjelp av prosesser og tekniske løsninger som muliggjør det de kaller *re-experience* av kunnskapen for de enkelte som er involvert. *Re-experience* omfatter prosesser og teknologi som muliggjør læring og refleksjon blant deltagerne. Dette er et viktig element i åpen kildekode-utvikling og jeg vil ha et fokus på hvordan dette kan støttes og potensielt forbedres ved økt integrasjon.

I bakgrunnskapitlene 2-5 ser jeg på henholdsvis åpen kildekode-utvikling, programvareutvikling, kunnskap og verktøy. Kapittel 6 beskriver min metodiske tilnærmelse. Kapittel 7 er et forslag til utviklingsproblemstilling basert på arbeidet i bakgrunnskapitlene. Kapittel 8 går på valg av rammeverk som er velegnet til å realisere utviklingsforslaget. Kapittel 9 og 10 er henholdsvis design og implementering av forslaget, mens kapittel 11 beskriver testene av prototypen og resultatene fra disse. Kapittel 12 diskuterer tilbakemeldingene fra åpen kildekode-utviklere som ble forespurt om å svare på spørsmål rundt verktøyintegrasjon og deres bruk av utviklingsverktøy.

Kapittel 2

Åpen kildekode

Åpen kildekode-programvare innebærer som navnet tilsier at kildekoden gjøres fritt tilgjengelig sammen med programvaren. Tanken bak er at når utviklere kan lese, distribuere og modifisere kildekoden for en gitt programvare vil dette bidra til at programvaren utvikler seg ved at folk forbedrer, tilpasser og retter opp feil i den [21].

Det benyttes gjerne flere begreper om denne typen programvaren, hvor de mest utbredte er *Open Source* og *Free Software*, eller en kombinasjon av disse: *FLOSS (Free/Libre/Open-Source Software)*. Hvilket begrep som benyttes er i stor grad et ideologisk spørsmål i og med at de omtaler den samme programvaren. Jeg vil i dette kapittelet bruke disse begrepene om hverandre avhengig av hva det er snakk om og for å belyse hva de underliggende ideologiske spørsmålene går ut på, mens for resten av oppgaven vil jeg benytte *åpen kildekode* når jeg omtaler denne type programvare.

Åpen kildekode-programvare utgis under en rekke ulike lisenser som har som formål å sikre at kildekoden forblir fri til videre utvikling og distribusjon og bruk. "Fri" og "Fritt tilgjengelig" i denne sammenhengen betyr ikke nødvendigvis det samme som at den er gratis, som Richard Stallman [46] formulerer det:

> You should think of "free" as in "free speech," not as in "free beer.".

Dette vil si at *Free Software* er et spørsmål om frihet til å kjøre, distribuere, studere, endre og forbedre programvaren, ikke et spørsmål om pris. Det som kjennetegner disse lisensene er at alle tillater (og mange av de krever) at kildekoden gjøres fritt tilgjengelig. Dette medfører at alle kan lese, modifisere og distribuere programvaren gitt at de følger lisensvilkårene. Det finnes en

rekke lisenser brukt av åpen kildekode-programvare. De “klassiske” lisensene som i stor grad fremdeles er blant de mest utbredte er følgende:

- GNU General Public License (GPL)
- GNU Library or LesserPublic License (LGPL)
- BSD license (Berkeley Software Distribution)
- MIT license

Senere har det kommet til en rekke andre lisenser som også har blitt populære, som for eksempel The Mozilla Public License etter at Mozilla ble distribuert som åpen kildekode i 1998 [21]. *Open Source Initiative* vedlikeholder en oversikt[1] over lisenser som følger *The Open Source Definition*[2] og dermed er lisenser som er godkjent som åpen kildekode-lisenser.

For å forstå hvordan *the Open Source Movement* har utviklet seg og fått den utbredelsen det har i dag må man tilbake til programvareutviklingen som foregikk ved universitetsmiljøene i USA på 60-tallet, da spesielt Artificial Intelligence Lab ved MIT. 60- og 70-tallet var også perioden da andre viktige faktorer som forgjengeren til Internett, ARPANET så dagens lys, samt utviklingen av Unix som alle er faktorer som har vært med på å forme åpen kildekode-miljøene vi ser i dag.

2.1 De første *hackerne*

I datamaskinenes spede barndom var det ikke noe reelt skille mellom maskinvare og programvare, og dermed mellom bruker og utvikler. Prisen på førstegenerasjons datamaskiner var skyhøy, en av de første kommersielt tilgjengelige maskinene: IBM 705 kostet i 1953 rundt 1,6 millioner dollar. Disse maskinene krevde at man skrev maskinvarespesifikk kode som krevde store ressurser og var en vanskelig jobb. I løpet av 60- og 70-tallet begynte prisene på maskinvare å gå drastisk ned, samt at det kom mindre mer praktiske maskiner på markedet. Samtidig hadde utviklere ved AT&T, Western Electric og Bell Telephone Laboratories overbevist ledelsen i sine respektive firma om at det var nødvendig med samarbeid om å utvikle et sett med grunnleggende verktøy, da spesielt kompilatorer, som skulle bidra

[1] http://www.opensource.org/licenses/
[2] http://www.opensource.org/docs/definition.php

til å redusere kompleksiteten, kostnadene og øke effektiviteten i programvareutvikling [54]. Dette dannet det nødvendige grunnlaget for de første programmeringsmiljøene som ble opphavet til mange av de grunnleggende tankene bak åpen kildekode.

Ut over 60- og 70-tallet hadde Artificial Intelligence Lab ved MIT utviklet seg til å bli et viktig sentrum for programvareutvikling, spesielt innen datakommunikasjon og tidsdelings-systemer. Dette miljøet hadde en kultur som var grunnlagt på åpenhet, deling og samarbeid. [54]. Eksperimentering og å bryte grenser stod i høysetet ved AI-labben ved MIT noe som krevde gode verktøy. Det var dermed viktig at utviklerne samarbeidet om å forbedre og vedlikeholde de grunnleggende verktøyene som benyttes i programvareutvikling. Fokuset lå på videreutvikling av felles goder som gagnet alle involverte. I realiteten utviklet de *Free Software* lenge før dette begrepet kom på banen. Denne gruppen betegnet seg selv som "hackers". Dette er et ord med mange konnotasjoner, både positive og negative. I denne sammenhengen brukes "hacker" om en produktiv teknisk kompetent person som utfordrer og utforsker begrensingene og mulighetene ved et system og som produserer resultater, om enn ikke nødvendigvis på en utpreget strukturert måte.

2.2 Unix

I 1964 startet forskere fra MIT, Bell Labs og General Electric jobben med å utvikle et nytt tidsdelings-system (flerbrukersystem) kalt Multics. Prosjektet viste seg etterhvert å bli for ambisiøst og komplekst og det ble avbrutt i 1969 uten å ha levert et system som fungerte tilfredsstillende. Igjen satt Ken Thompson og Dennis Ritchie, to ambisiøse forskere fra Bell Labs, som med verdifull erfaring fra Multics-prosjektet bestemte seg for å bruke denne erfaringen til å lage et nytt og enklere system. Incentivene til å lage et nytt system var ikke bare tekniske, men også kulturelle. I løpet av 60-tallet kom et paradigmeskifte i databransjen hvor arbeidsoppgaver i større og større grad begynte å bli organisert etter ren samlebåndsteori a la Henry Ford. På dette viset ble programmererne skilt fra operatørene av datamaskinene, noe programmererne fant å være et sterkt tap av selvstendighet og kontroll. Ken Thompson innså at denne organiseringen ikke fungerte spesielt godt og at dette var delaktig til at Multics feilet. Løsningen ble derfor å starte på nytt [54].

Sommeren 1969 så dermed UNICS dagens lys, som senere ville skifte navn

til Unix. Filosofien bak Unix var som Dennis Ritchie sa det “build small neat things instead of grandiose ones” [54, s. 26]. Unix ble dermed bestående av små og relativt enkle byggestener, og kan ses på som en verktøyboks av små enkle moduler som kan kombineres for å utføre komplekse jobber. Dette utviklet seg til å bli kjent som “a Unix philosophy”[54, s. 28]:

- Write programs that do one thing and do it well.
- Write programs that work together.
- Write programs that handle text streams because that is a universal interface.

Samtidig utviklet Dennis Ritchie et nytt programmeringsspråk kalt “C” som kunne kjøre på Unix. C var utviklet for å være behagelig å kode i, være fleksibelt og med få innskrenkninger. Tradisjonelt hadde operativsystem blitt skrevet i assembler for effektiv utnyttelse av maskinvaren. Dette betydde derimot at programvaren ble sterkt knyttet til maskinvaren og måtte skrives om for å kjøre på annen maskinvare. Thompson og Ritchie innså at maskinvaren og kompilatorteknologien på slutten av 70-tallet var i ferd med å bli kraftig nok til å at et helt operativsystem kunne skrives i C, og ved utgangen av 1978 hadde de med suksess implementert Unix i C og kjørt det på flere ulike maskinvareplattformer. Implikasjonene av dette var enorme. Det ble ikke lenger nødvendig å betale for maskinvarespesifikk programvare, utviklere kunne nå i stedet benytte den samme programvaren på flere plattformer. Videre var C mye enklere å programmere i enn de gamle systemene hvor man måtte benytte assembler. Denne kombinasjonen viste seg å være tilpasningsdyktig til en rekke ulike omgivelser, og spredte seg raskt blant *hackerne* [40].

2.3 ARPANET

I 1968 så oppstarten til et prosjekt som skulle være kritisk for den videre spredning av *Free Software* både som programvare og ideologi. Dette var året da ARPANET begynte som et eksperiment for å koble datamaskiner sammen i et nettverk. Over de neste årene vokste nettverket til å inkludere de fleste av initiativtakeren Pentagon’s Defense Advanced Research Projects Agency (DARPA) sine viktigste utviklingsfasiliteter [54]. ARPANET fortsatte å vokste ved å inkludere hundrevis av universitetet, militære fasiliteter og forskningslaboratorier, og ble på denne måten det første

høyhastighetsnettverket som hadde en utbredelse som strekte seg over et helt kontinent. For "hackere" over hele USA ble dette et vendepunkt. Isolerte små grupper av programmere som alle hadde utviklet sin lokale åpne kultur rundt programvareutvikling ble nå knyttet sammen og dannet et større felleskap [40].

ARPA ledet utviklingen av en stor samling programvare nødvendig for å håndtere og utvide ARPANET. Disse utviklet seg til standarder i form av "Request for Comments" som definerte protokoller for kommunikasjon. Grunnleggende protokoller i bruk den dag i dag ble utviklet på denne tiden. I 1972 kom Transmission Control Protocol (TCP) og i 1977 så Internet Protocol (IP) dagens lys. Disse muliggjorde at maskiner relativt lett kunne bli koblet til nettverket, og protokollene ble etterhvert støttet av Unix, som spredte disse standardene rundt til hele verden. På denne måten ble åpen kildekode-utviklere en nøkkelfaktor for nettets suksess ved å produsere en vedvarende strøm av programvare, protokoller og standarder som økte dets funksjonalitet og nytteverdi [34].

I begynnelsen av 80-tallet begynte kommersialiseringen av Internett. Programvare som ble fritt delt ble gradvis erstattet av proprietære programvareløsninger. Dette hadde flere grunner; det ble innsett at det raskt voksende Internettet hadde et enorm fortjenestepotensiale, samt at USA's regjering etter press fra industrien og ideologisk opposisjon mot sterk statlig styring, trakk seg tilbake fra å forsvare og promotere åpne standarder [42].

2.4 Framveksten av proprietær programvare

På 70-tallet kom de første "personlige datamaskinene" på markedet. Disse markerte et epokeskifte i databransjen ved at programvaren for disse ikke var fritt tilgjengelig. Man måtte betale for den og det var ikke tillatt å gi den videre til andre. Den kommersielle proprietære programvareindustrien var født. Unix fikk problemer på 80-tallet i møtet med denne nye programvareindustrien med Microsoft i spissen, som trakk til seg nøkkelpersoner som var fristet av de økonomiske aspektene til de proprietære alternativene. AT&T begynte selv å forstå at de kunne tjene penger på Unix og etter at restriksjonene som hindret AT&T i å kunne gå inn i databransjen ble opphevet tidlig på 80-tallet kastet de seg med full tyngde inn som en proprietær aktør på markedet. Lisenskostnadene for Unix skjøt i været og kom raskt opp i flere hundretusen dollar. Dette medførte at bare store aktører hadde råd til å kjøpe lisens og dette gikk hardt ut over forskere

ved universiteter og mindre bedrifter. Før AT&T gjorde Unix proprietært hadde det blitt utviklet en populær variant av systemet ved Berkley kalt BSD. Denne fikk nå også problemer siden den krevde lisens fra AT&T. Den proprietære koden ble derfor ut over 80- og 90-tallet gradvis fjernet og byttet ut med ubeheftet fri kode. Dette systemet er opphavet til dagens ulike BSD-systemet: OpenBSD, FreeBSD, NetBSD, m.m. som er lisensiert under den liberale BSD-lisensen [54].

BSD-variantene var ikke de eneste derivative versjonene av Unix som var utbredt. Ut over 80-tallet begynte Unix å få problemer med stor spredning innen implementasjoner og standarder. Dette førte til en rekke kompatibilitetsproblemer mellom ulike versjoner av programvaren og forskjellig typer maskinvare. Dette kunne gått forbi uten den helt store konsekvensene, men etter at økonomien igjen fikk en opptur i USA rundt 1992, med databransjen og Internett i fokus, førte alle disse problemene til at Unix mistet tiltro akkurat da større investeringer i datasystemer ble gjort. Utviklingen til Unix ga også åpen kildekode-miljøene et uklart bilde utad, da spesielt ovenfor kommersielle aktører [54].

2.5 The Free Software Foundation

Med framveksten av den proprietære programvareindustrien begynte *hacker*-miljøet ved MIT å gå i oppløsning, og den åpne kulturen med fri deling av kode begynte sakte men sikkert å forsvinne. Mange av de flinkeste *hackerne* forsvant ut i lukrative jobber, nye systemer kom ikke lenger med tilgjengelig kildekode, og restriktive vilkår ble mer og mer vanlig. En av *hackerne* ved MIT, Richard Stallman så med bekymring på denne utviklingen. Han forsøkte først å jobbe mot denne utviklingen internt ved MIT, men når det ikke førte fram sa han i 1984 opp stillingen sin for å vie all sin oppmerksomhet til å promotere *Free Software* [54].

R. Stallmans respons var å opprette en ikke-profitt organisasjon kalt *The Free Software Foundation* for å støtte arbeidet. Han starter så arbeidet med *The GNU Project* som hadde som mål å skape et fritt operativsystem som alternativ til det nå proprietære UNIX-operativsystemet. Stallmans *GNU Manifesto* beskriver grunnlaget for det han kaller *Free Software* hvor han spesifiserer fire kriterier for at programvare skal kunne kalles fritt [54, s. 48]:

- Freedom to run the program for any purpose

- Freedom to study how the program works and modify it to suit your needs
- Freedom to redistribute copies, either gratis or for a monetary fee
- Freedom to change and improve the program and to redistribute modified versions of the program to the public so others can benefit from your changes

For at disse frihetene skulle kunne ivaretas introduserte Stallman GPL-lisensen som er basert på det han kaller "copyleft" som har som formål å sikre at programvaren og alle derivative versjoner er og forblir fri. På denne måten kan ikke programvare under GPL-lisensen bli gjort proprietær eller kombineres med kode som ikke er fri. *The GNU Project* omfatter også utviklingen av en rekke programmer som er blant de mest brukte innen "unix-kompatibel"-programvare, som editoren *Emacs*, kompilatoren *GCC* og debuggeren *GDB*. Målet om å lage et komplett fritt system ble derimot aldri realisert, en viktig del manglet: systemkjernen [54]. Utvikling av en kjerne kalt *HURD* hadde pågått i flere år, men aldri materialisert seg til et komplett og produksjonsklart system.

2.6 Linux og World Wide Web

I januar 1991 begynte Linus Torvalds arbeidet med et nytt operativsystemet som kunne kjøre på en personlige datamaskin (PC). Han modellerte systemet etter Unix og utpå høsten gjorde han første versjon av systemet som han kalte Linux tilgjengelig på nettet sammen med en beskjed om at dette var laget av en *hacker* for *hackers* og at han ville sette pris på enhver hjelp han kunne få til videre utvikling. Responsen var enorm og før året var omme hadde over 100 personer vist sin interesse. Linux leverte den manglende biten som skulle til for å skape et helt fritt operativsystem og ble kombinert med GNU-verktøyene til det frie operativsystemet GNU/Linux. Dette kom på en tid da det proprietære Unix-markedet var i krise og den proprietære industrien med Microsoft i spissen økte sin markedsandel i nærmest alle segmenter av programvareindustrien. Det andre frie alternativene, BSD-systemene, hadde fragmentert på samme måte som Unix, noe som hemmet utviklingen av de enkelte systemene [54].

Et interessant aspekt her var Linus Torvalds sin tilnærming til utviklingen av Linux. Hvor kommersielle aktører og BSD-systemene ble utviklet av små

grupper av tett sammenknyttede utviklere, slapp Torvalds Linux fritt ut på nettet og et stort antall utviklere fra hele verden kom sammen og samarbeidet om å videreutvikle systemet. Denne utviklingsmodellen viste seg å fungere svært godt og rundt 1993 hadde Linux utviklet seg til å et stabilt og pålitelig system som kunne konkurrere mot en rekke av de proprietære systemene på markedet [40]. Programmerere som så etter en vei å slippe unna "the UNIX standard war" og proprietær "lock-in" begynte å bruke det nye frie alternativet og ga prosjektet stadig større tyngde og tilstedeværelse [42].

Fremveksten av Linux kom samtidig med en annen revolusjon som skulle vise seg å være av stor viktighet for åpen kildekode, nemlig World Wide Web (WWW) som førte til en eksplosiv vekst av Internettet som gjorde det til et massemedium og nærmest allemannseie. I 1998 slapp Netscape kildekoden til nettleseren Netscape Navigator, noe som ga åpen kildekode en signifikant økning i anerkjennelse blant kommersielle aktører [34]. Motivert av Netscape sin frigjøring av kildekode ble *The Open Source Initiative* (OSI) opprettet i 1998. OSI introduserte termen *Open Source* som en erstatning for den mer kontroversielle termen *Free Software*. OSI er en ikke-profitt organisasjon som er dedikert til å gjøre åpen kildekode-programvare mer tiltrekkende for kommersielle aktører, med et sterkere fokus på praktiske over etiske problemstillinger [20].

Gjennom Internett og WWW har åpen kildekode-utvikling vokst kraftig. G. von Krogh og E. von Hippel [51, s. 211] kommer med et viktig poeng her, hvor de påpeker at framveksten og utbredelsen av Internett gir en unik mulighet til den distribuerte utviklingsmodellen som åpen kildekode-prosjekter avhenger av. Tilgjengeligheten av hensiktsmessige rammeverk for utvikling ble plutselig mye bedre.

2.7 Diskusjon

Som man ser har ikke åpen kildekode utviklet seg i isolasjon fra resten av programvareindustrien, og ikke uten sine opp og nedturer. Historien viser at utviklingen av fri programvare er tett knyttet til hvordan resten av programvareindustrien har utviklet seg siden 60-tallet, og dermed også tett knyttet til proprietær programvare. Den første tiden med fri deling av programvare, ARPANET og fokus på samarbeid og tekniske nyvinninger ble slått ned av deregulering, privatisering og lukking av programvare i jakten på profitt [42]. Free Software Foundation ble senere starten på en ny oppblomstring sammen med Linux og utbredelsen av Internett.

Åpen kildekode-programvare blir i dag utviklet av tusenvis av fysisk adskilte personer som kommuniseres og jobber sammen over Internett. De jobber i en desentralisert, åpen og tilsynelatende kaotisk prosess. Til tross for mangelen av sterk styring og organisering har åpen kildekode-programvareutvikling vist seg å kunne utnytte og organisere kompetansen til deltakerne og produsere konkurransedyktig programvare av høy kvalitet [40] [25] [26].

Proprietær programvareutvikling og åpen kildekode-utvikling står fremdeles i dag langt unna hverandre når det gjelder deres grunnlag for forretningsmodell. Proprietære aktører baserer seg på intellektuell eiendom som må beskyttes og holdes hemmelig for å være et grunnlag for fortjeneste. "Society has a vital interest in encouraging and rewarding innovation" [51, s. 212]. Dette har tradisjonelt sett blitt representert ved de to modellene "Private-collective" som proprietær utvikling legger til grunn og "collective action model".

"Private-collective"-modellen hevder at innovasjon blir støttet av private investeringer som så vil gi inntjening for investorene. For å støtte opp om denne gis innovatører fordeler via patenter, copyrights og "trade secrets". Enhver lekkasje i denne modellen av proprietær kunnskap vil dermed redusere inntjeningen [51, s. 212-213].

"Collective action"-modellen på sin side går på å forske frem kunnskap til alles beste. Den hevder at de involverte må gi slipp på kontrollen over kunnskapen som produseres, og gjøre den til et offentlig gode. Dette er den modellen som er dominant i blant annet en rekke forskningmiljøer [51, s. 213].

Det interessante med åpen kildekode-utvikling er at den plasserer seg i mellom disse to eksisterende modellene. G. von Krogh & E. von Hippel kaller dette "private-collective innovation-model". Her bruker deltakerne av sine egne ressurser som de investerer i å produsere ny programvare. De krever ikke eierskap over koden, men gir fri tilgang til koden for alle. Dette ser da ut til å gi det beste fra begge de foregående modellene - ny kunnskap blir skapt fra private investeringer, for så å bli frigitt som et offentlig gode [51, s. 213].

Her ligger også derimot et paradoks. Som Lerner & Tirole [27] skriver: "Why should thousands of top-notch programmers contribute freely to the provision of a public good?" Et kritisk element her er nok at selv etter at programvaren har blitt frigitt til allmennheten, så betyr ikke dette nødvendigvis et tap av profitt og fordeler for de som frigir den. Dette kan ha flere årsaker, som f.eks økt spredning som kan gi inntekter via nettverks-effekter. G. von Krogh & E. von Hippel presiserer også at deltakerne får en gevinst som økt kompetanse, utfordrende og interessant arbeid, og anerkjennelse i miljøet. I tillegg er de ofte selv brukere av programvaren de utvikler.

Hvor relevant er så åpen kildekode i dag? Hvis man ser på sluttbrukermarkedet har proprietære aktører en overveldende dominans, men hvis man ser litt under overflaten på viktig infrastruktur i dagens digitale samfunn tegner det seg et annet bilde. Internett er tuftet på en rekke kritiske åpen kildekode-programvare. Viktige protokoller som blant annet TCP/IP er åpne standarder utviklet ved Berkeley, BIND brukes for å administrere DNS som alle nettsteder avhenger av for å være tilgjengelige, Sendmail er hjertet i epost-backbone, Apache er den mest dominante nettserveren med over 60% markedsandel[3], Perl, Python og PHP er de dominerende skriptspråkene for dynamiske nettsteder, osv. I tillegg finner man åpen kildekode i bruk av tjenesteleverandører, som for eksempel Google som kjører på et massivt nettverk av 100000 Linux-servere. Tim O'Reilly karakteriserer åpen kildekode som grunnlaget for et kommende paradigmeskifte hvor "software commoditization", samarbeid over elektroniske nettverk, og programvare som tjenester står i fokus. Åpen kildekode bidrar til å standardisere mer programvare og dermed blir denne en "commodity", nye sekundære markeder kommer til rundt disse og blir nye mulige vekstområder for proprietære aktører. Tjenester basert på standardisert programvare ser ut til å bli mer og mer viktig her [37]. Åpen kildekode ser dermed ut til å ville kunne ha spille en viktig rolle i hvordan programvareindustrien vil utvikle seg i årene fremover.

[3]http://news.netcraft.com/archives/web_server_survey.html

Kapittel 3

Programvareutvikling

Fagfeltet programvareutvikling (Software Systems Development) omfatter systematisk design, implementering, testing og vedlikehold av programvare for datasystemer. I hvor stor grad disse ulike elementene er involvert avhenger av konteksten til utviklingen. Programvareutvikling varierer stort i omfang, kompleksitet og ansvarsområde. Det er stor forskjell i kravene til utvikling av "Enterprise Information Systems" som banktjenester og systemer innen medisin og kirurgi i forhold til programmer utviklet av privatpersoner for eget bruk. Ulike metoder og prosesser er blitt utviklet for å imøtekomme dette store spennet i rammevilkår.

Programvareutvikling som disiplin har gjennomgått en stor utvikling i løpet av andre halvdel av det tjuende århundre. I de første årene var det maskinvare som stod i fokus, som den desidert største utgiftsposten i datidens datasystemer. Programvare på sin side ble gitt liten oppmerksomhet, og ble utviklet uten nevneverdig styring eller bruk av formelle prosesser eller metoder. Ut over 60- og 70-tallet ble programvare en stadig viktigere del av systemet, samtidig som kompleksiteten økte dramatisk. Dette medførte blant annet at vedlikehold av programvare begynte å bli en betydelig kostnadsfaktor. Med dette skifte i fokus fra maskinvare til programvare, ble det kritisk å utvikle programvareutvikling som disiplin for å skape et rammeverk for prosjektstyring og veldefinerte prosesser for programvareutvikling [39].

3.1 Systemutvikling

IEEE [9] definerer *systemutvikling* på følgende måte "Software engineering: (1) The application of a systematic, disciplined, quantifiable approach to

the development, operation, and maintenance of software; that is, the application of engineering software. (2) The study of approaches as in (1).". Ingeniørvitenskap er anvendelse av vitenskaplig og teknisk kunnskap til å løse problemer ved å benytte teori, metoder og verktøy tilgjengelig innenfor gitte tekniske, organisatoriske og finansielle rammer. *Systemutvikling* omfatter alle aspekter ved utvikling, endring og vedlikehold av komplekse systemer hvor programvare spiller en nøkkelrolle. Systemutvikling omfatter dermed også maskinvareutvikling, fremgangsmåter, prosessdesign og systemanvendelse i tillegg til programvareutvikling [44].

Pressman [39] beskriver systemutvikling som en lagdelt disiplin med grunnlag i et organisatorisk fokus på kvalitet. Disse lagene er henholdsvis prosess, metode og verktøy. Prosesslaget er fundamentet for systemutviklingsarbeidet, og definerer et rammeverk som danner grunnlaget for prosjektstyring. Systemutviklingsmetoder og verktøy blir så brukt i kontekst av dette rammeverket. Systemutviklingsmetoder innebærer teknisk "how-to" for å utvikle programvare gjennom en rekke aktiviteter som analyse, design, programmering, testing og vedlikehold. Verktøy har som mål å effektivisere og automatisere prosess- og metode-lagene, samt å bistå i analyse, design, implementering og testing av systemet.

Prosess i programvareutviklings-sammenheng er et sett av aktiviteter og resultatene fra disse som resulterer i et programvareprodukt. Det et sett fundamentale prosesser som er felles for all tradisjonell programvareutvikling:

- Kravspesifisering - spesifisering av ønsket funksjonalitet og definering av rammebetingelser
- Design - Omfatter hvordan funksjonaliteten skal bli gitt av de ulike systemkomponentene
- Implementasjon - koding av selve programvaren
- Testing - Validering av programvaren for å finne og fjerne feil og sørge for at den oppfyller de gitte kravene
- Vedlikehold - fortløpende feilretting og tilpasning av programvare i drift

3.1.1 Prosessmodeller

Prosessmodeller er forenklede beskrivelser av prosessene som inngår i utvikling av programvare. Felles for disse modellene er at de har en kravspesifikasjon som utgangspunkt, og levering av et produkt som endelig mål.

Modellene omfatter i stor grad de generelle stegene beskrevet over, men har forskjellig fokus og gjennomføring. Det finnes en rekke kjente modeller, hvor man kan trekke frem den lineære fossefallsmodellen, evolusjonær utvikling som fokuserer på prototyping, *formal system development* som er basert på formelle matematiske transformasjoner, gjenbruksorienterte modeller som fokuserer på gjenbruk av komponenter, samt iterative modeller. Jeg velger å trekke frem fossefallsmodellen og iterative modeller som kan sees på som å være i hver sin ende av skalaen i forhold til åpen kildekode-utvikling, og som vil bli brukt som et sammengrunnlag. Fossefallsmodellen er veldig rigid og strukturert og står i sterk kontrast til åpen kildekode-prosesser, mens de iterative modellene er de som ser ut til å ligge nærmest prosessene som foregår i åpen kildekode-utvikling.

Fossefallsmodellen

Fossefallsmodellen (også kjent som livssyklusmodellen) er den eldste og mest brukte av systemutviklingsmodellene. Den har en systematisk og sekvensiell tilnærming til programvareutvikling, som begynner med analyse og fortsetter sekvensielt med design, koding, testing og vedlikehold. Hvert steg resulterer i dokumentasjon som blir evaluert og godkjent. Det påfølgende steget startes ikke opp før det foregående er avsluttet. I praksis er disse stegene vanskelig å skille fra hverandre og har en tendens til å overlappe og ha en pågående informasjonstrøm mellom seg. En sekvensiell prosess blir dermed lite fleksibel og dårlig tilpasset hvordan programvareutvikling foregår i praksis, da spesielt fordi kostnaden ved å produsere og validere dokumentasjonen er såpass høy at iterasjoner blir en kostbar affære [44].

Hovedproblemet med denne modellen er derfor dens lite fleksible inndeling i distinkte steg. Dette gjør modellen i liten grad i stand til å håndtere endringer i krav og er dermed best egnet for prosjekter hvor man har stor oversikt og innsikt i systemkravene. På en annen side reflekterer denne modellen ingeniørvitenskaplige prinsipper og er dermed fremdeles i utstrakt bruk for større programvareprosjekter [44].

Iterative prosesser

For større systemer vil det ofte være nødvendig å bruke flere ulike prosessmodeller, samt støtte prosessiterasjon hvor deler av prosessen kan gjentas ved kravendringer for å tilpasse systemet til disse nye/endrede kravene. To hybride modeller som støtter iterativ utvikling er inkrementell utvikling og

spiralmodellen [44].

Inkrementell utvikling Denne modellen kombinasjon elementer fra den fossefallsmodellen med den tilpasningsdyktigheten fra evolusjonær utvikling. Hver iterasjon består av en lineær sekvens: analyse, design, implementering, testing og leverer et ferdigstilt inkrement av systemet [39]. På denne måten kan inkrementell utvikling dra fordel av fossefallsmodellens strukturerte fremgangsmåte for å produsere robuste systemet kombinert med større fleksibilitet ved at man lettere kan iterere over alle stegene. I inkrementell utvikling identifiserer kunden hvilke tjenester de ønsker og gir hver tjeneste en prioritet. En rekke inkrementer blir så definert basert på prioriteten gitt til de ulike tjenestene hvor hvert inkrement leverer en delmengde av systemets funksjonalitet. De inkrementene med høyest prioritet blir levert først. Med systeminkrementene definert blir kravene til første inkrement identifisert i større detalj og utviklet ved hjelp av en hensiktsmessig prosssesmetode. Under utviklingen av inkrementet vil videre kravanalyse for senere inkrementer kunne finne sted, men nye krav for inkrementet under utvikling blir ikke akseptert [44].

Den inkrementelle modellen har flere fordeler ved tidlig levering av de viktigste delene. Kundene slipper å vente til hele systemet er ferdig før de kan ta deler i bruk. Kundene kan raskere få erfaring med systemet og gi mer verdifull tilbakemelding. Ved å levere de viktigste delene først fører dette også til at de gjennomgår mest testing. Alt i alt fører dette til en lavere risiko for prosjektet. For at denne modellen skal fungere optimalt er det en del krav som må tilfredsstilles. Inkrementer bør være relativt små, noe som kan føre til problemer når man skal få kundens krav til å passe inn i disse. Det kan også være vanskelig å identifisere felles komponenter som alle inkrementer krever er til stede [44].

En relativt nylig videreføring av inkrement utvikling er "agile methods" som *extreme programming* (XP) [33]. XP er en av flere modeller som ofte kommer under benevnelsen "agile methods". Den baserer seg på fire kjerneverdier:

- Kommunikasjon - en nødvendighet for å oppnå suksess
- Enkelhet - lag løsningene så enkel som mulig, ikke prøv å forutse fremtiden
- Tilbakemelding - slipp nye versjoner ofte for rask tilbakemelding, fortløpende integrasjon og testing

- Mot - vær offensiv og gå for en løsning

XP gjennomføres gjennom tolv metoder som bygger på de fire kjerneverdiene. *P*lanlegging: avgjør omfanget til neste iterasjon i konsultasjon med både utviklere og kunde. **Små programvareslipp**: få programvaren ut i bruk raskest mulig for tidlig tilbakemelding. **System-metafor**: få en helhetlig forståelse av systemet. **Enkelt design**: hold designet simpelt og la funksjonaliteten styre designet. **Testing**: utstrakt bruk av testing. **Refactoring**: forbedring av design uten å endre programmets oppførsel. **Pair Programming**: jobb sammen med en partner på produksjonskode. **Felles eierskap**: alle på teamet kan endre på alle deler av systemet. **Fortløpende integrasjon**: utviklerne integrerer og bygger programvaren mange ganger daglig. **40-timers uke**: raske iterasjoner krever stor innsats. **Kunde som deltager**: kunden er en del av teamet. **Kodestandarder**: kommunikasjon er en nøkkelfaktor, gode kodestandarder forbedrer kommunikasjon.

Spiralmodellen I stedet for å presentere programvareprosessen som en sekvens av aktiviteter, beskriver spiralmodellen prosessen som en spiral. Den ble først introdusert av Boehm (1988) og er blitt en kjent og utbredt modell. Denne baserer seg på evolusjonær utvikling hvor man utfører stegene i fossefallsmodellen for hvert iterasjon (loop). Formålet er å redusere risiko ved at man ikke definerer alt fullt ut i starten, men implementerer utvalgt funksjonalitet og får tilbakemelding på denne før de begynner på en ny runde i spiralen hvor man bruker disse tilbakemeldingene som grunnlag for videre arbeid. Hver loop i spiralen representerer en fase i prosessen: gjennomførbarhetsanalyse, kravspesifikasjon, design, etc. Hver loop i spiralen er delt i fire sektorer:

1. Definere mål
2. Risikoanalyse og tiltak for risikoreduksjon
3. Utvikling og testing
4. Planlegging

Den viktigste forskjellen som skiller spiralmodellen fra andre modeller er at den eksplisitt vurderer risiko. En syklus starter ved at man definerer mål og finner alternative måter å oppfylle disse, samt finner alternativenes begrensninger. Alternativene blir deretter veid opp mot målene og resulterer i identifisering av risikokilder. Deretter blir disse risikokildene evaluert

grundigere gjennom mer detaljert analyse, prototyping og simulering. Spiralmodellen har ingen fastlagte spesifikasjon- eller design-faser, den bygger på andre prosessmodeller som brukes avhengig av behov [44].

3.2 Prosesser i åpen kildekode-utvikling

Når man ser på åpen kildekode-programvareutvikling så viser det seg at de i liten grad har adoptert tradisjonelle programvareutviklingsmetoder og prosessmodeller. Jeg vil her fokusere på *The Open Source Spiral Model* [3] som er et forsøk på å forklare åpen kildekode-utvikling som en prosessmodell, samt Weber [54] sine prinsipper for åpen kildekode-utvikling.

3.2.1 *The Open Source Spiral Model*

Som både Steve McConnel [29] og Bollinger et.al. [3] påpeker så kan man knapt påstå at prosessene i åpen kildekode-utvikling kvalifiserer som en metodologi. Man kan heller si det omfatter et sett prinsipper som definerer den absolutt minimale prosessen som er nødvendig for å sette en gruppe mennesker i stand til å produsere programvare av høy kvalitet. Bollinger et.al. [3] beskriver åpen kildekode-utvikling som en rask og iterativ versjon av Barry Boehm spiralmodel [2] for programvareutvikling. I åpen kildekode-utvikling sitter disse spiralene så tett at det utenfra kan se ut som et stort steg. På innsiden derimot foregår det raske iterasjoner over kravspesifisering, design, koding og testing som kan være nede i ikke mer en noen timer i omfang [3, s. 10]. Et viktig element å trekke frem i denne modellen er "release early, release often". Iterasjonene går ikke bare på selve utviklingen, men inkluderer også brukerne ved å slippe nye versjonen tidlig og ofte. På denne måten får man ikke bare inkorporert tilbakemeldinger fra brukerne raskt, men ved å utnytte Internett som medium til det fulle oppnår en effekt som skalerer opp til et nivå som rivaliserer kompleksiteten til selve utviklingsprosessen [40].

I tradisjonell proprietær utvikling har man produsenter og konsumere, tilbud og etterspørsel. Prosessene i åpen kildekode-utvikling forkaster dette som utgangspunkt. Skillet mellom produsenter (utviklere) og konsumenter (brukere) viskes dermed ut, og mange av brukerne blir i stedet en tett integrert del av utviklingsprosessen. Prosessen er ikke grunnlagt i noe systematiske prosedyrer eller formelle argumenter, men studier av åpen kildekode-prosjekter i praksis har gitt innsikt i hvordan den utspiller seg, hvor blant annet Eric Raymond har publisert analyser av disse prosessene.

Jeg vil her presentere 8 prinsipper for åpen kildekode-utvikling som Weber [54] beskriver med utgangspunkt I Eric Raymonds arbeid samt intervjuer og egne observasjoner. Disse vil være med å utdype de prosessene som ligger i *The Open Source spiral model*.

Interesse og incentiv

Utviklere i åpen kildekode-prosjekt velger selv å delta, og hvilke oppgaver de ønsker å jobbe på. Det er derfor naturlig at folk trekkes mot de oppgavene som er spennende og signifikant i et prosjekt. Dette kan gå på det å lage ny og spennende funksjonalitet eller gjøre vanskelige oppgaver på en elegant måte, men utviklere ser også etter muligheter for å lære nye ting og få økt kompetanse. Når det gjelder oppgaver som ikke er spesielt interessant, men som like fullt er viktig for prosjektet finnes det underliggende fordeler ved å utføre disse for deltagerne. Man er interessert i at prosjektet skal lykkes og bli brukt for at det skal ha noen verdi, derfor må de mindre spennende oppgavene også bli utført for at arbeidet som er lagt inn i de mer spennende delene ikke skal være bortkastet.

"Scratch an itch"

For å oppfordre til frivillige bidrag hjelper det med en eller annen type belønning. En meget god belønning i denne sammenhengen er en løsning på et konkret problem man møter i det daglige virkemål. Ved å lage et program som løser problemet får man "scratched the itch". For å gi et perspektiv på hvor viktig denne typen programmer er; det er estimert at 75% av all kode er skrevet internt hos en eller annen bedrift eller privatperson for privat bruk for løse ett eller annen spesifik problemstilling. Åpen kildekode-utvikling utpeker seg med at den klarer å hente ut denne innsatsen og gjøre den tilgjengelig og åpen og videreutviklet i samarbeid med andre.

Unngå å finne opp hjulet på nytt

Koding er tidkrevende og vanskelig, så utviklere leter alltid etter metoder for å effektivisere utviklingen. Dette gjelder ikke minst for åpen kildekode-utviklere som ikke får betalt for sitt arbeid. Så i stedet for å lage alt fra grunn av leter man etter eksisterende løsninger som kan tilpasses og benyttes. Åpen kildekode legger til rette for dette ved at koden er fritt tilgjengelig under en

lisens som sikrer at den vil forbli fri og tilgjengelig uavhengig av rundtliggende forhold som kan gi problemer i proprietær utvikling.

Parallelle arbeidsprosesser

Forskeren Danny Hillis utalte følgende: “There are only two ways we know of to make extremely complicated things. One is by engineering, and the other is evolution.” [4]. Programvareutvikling er en komplisert oppgave. Tradisjonell programvareutvikling har sine røtter i ingeniørvitenskapen, åpen kildekode-utvikling derimot ligner mer på evolusjon. Viktige problemer vil tiltrekke seg en rekke ulike personer som vil jobbe med hver sin løsning enten for seg selv eller i samarbeid med andre. De beste løsningene vil vinne frem, mens de andre blir forkastet. På denne måten finner man frivillig parallell bearbeidelse av et problem.

“Law of large numbers”

Funksjonalitetstesting av programvare er et meget komplekst område. Ethvert ikke-trivielle dataprogram har for alle praktiske formål et funksjonelt nærmest uendelig antall stier gjennom koden. Testing vil stort sett bare avdekke en brøkdel av disse. Jo flere feil du genererer, og jo tidligere du genererer disse, jo større sjanse har du til å luke ut et større antall av problemene med det ferdige produktet. Proprietære utviklere står ovenfor et dillema her. Utviklerne ønsker tidlig testing, mens tidlige versjoner med mange feil er en katastrofe fra et markedsførings-synspunkt. Åpen kildekode-utviklere trenger ikke ta hensyn til dette. Man kan si at åpen kildekode-programvare alltid er under utvikling og dermed aldri “ferdig”, men samtidig er den fullt tilgjengelig og dermed tillater kontinuerlig modifisering. Et stort antall testere med ulik bakgrunn og bruk av programvaren vil føre til at flere feil blir funnet raskere, feilene blir kommunisert til et stort antall personer som øker sannsynligheter for at noen raskt finner en løsning som så kan bli integrert inn i prosjektet igjen på en effektiv måte.

Dokumentering av arbeid

Dokumentering av kode er viktig i alle prosjekter, dog kanskje i enda større grad i åpen kildekode-utvikling. En desentralisert, distribuert utvikling kan vanskelig fungere uten dokumentasjon. God dokumentasjon er vitalt for å kunne gjøre nye bidragsytere i stand til å sette seg inn i koden. På grunn av

tilgjengeligheten til åpen kildekode så er det også viktig å gjøre den forståelig slik at den vil videreføre kunnskapen til de som skrev den i utgangspunktet. Virkeligheten derimot er ikke alltid så rosenrød, som i proprietær utvikling blir dokumentering også ofte oversett i åpen kildekode-utvikling.

"Release early, release often"

For å utnytte prosessene som ligger i åpen kildekode-utvikling til det fulle trenger brukerne/utviklerne kontinuerlig tilgang til produktet som utvikles. Åpen kildekode-utvikling er dermed ikke bare evolusjonær fordi den omfatter parallell utvikling, men også gjennom at den kan akselerere utbyttinga av nye generasjoner av programvaren som forsterker raten av feilretting. Tilbakemelding og oppdateringsfrekvensen i åpen kildekode-prosjekter er dermed ekstremt høy, men den er likevell ikke helt kaotisk og med så mange blindgater som biologisk evolusjon har. Prosjektene har mekanismer som holder dette i sjakk gjennom en rekke implisitte sosiale normer og regler.

Stor grad av kommunikasjon

Et særtrekk ved åpen kildekode-utvikling er graden av skriftlig kommunikasjon. Alle aspekter av utviklingen blir diskutert ved hjelp av en rekke kommunikasjonsverktøy på nettet. Volumet det er snakk om er enormt, og omhandler alt fra spesifikke tekniske problemstillinger til generelle politiske og økonomiske aspekter rundt programvareutvikling. Deler av denne kommunikasjonen er organisert rundt det å finne et konsensus for en teknisk problemstilling, mens andre diskusjoner er rene meningsytringer om en rekke tema. Fokuset er på frihet til å uttrykke seg og/eller å komme frem til en løsning. Tonen kan derfor til tider være relativt krass og rett på sak.

3.2.2 Samarbeid i åpen kildekode-utvikling

Weber [54] identifiserer tre faktorer ved samarbeid i åpen kildekode-utvikling som er karakteristiske. Disse er bruk av teknologi, programvarelisenser, og de sosiale strukturene som former bruken av teknologien.

Teknologi som skaper muligheter for samarbeid

Nettverk har lenge vært en essensiell del av åpen kildekode-utvikling. Tidlige fellesskap som ved MIT og Bell Labs var geografisk begrenset. Internett

(og Arpanet før det) ble nøkkelen for innovasjon. Med Internett forsvant barrierene for teknisk kompatibilitet mellom nettverkene og de geografiske begrensningene for deling av kode. Elektroniske kommunikasjonsverktøy har sine problemer med skjult kunnskap, men de skalerer effektivt i mye større grad fysiske omgivelser. Muligheten for å bringe sammen mennesker med en variert og bred kunnskap og ekspertise øker også. En stadig økende grad av tilgang og hastighet muliggjør effektiv bruk av en rekke verktøy, noe åpen kildekode-utviklere har utnyttet til det fulle [54].

Lisenser som sosiale strukturer

Samarbeid i åpen kildekode-prosjekt er fundamentalt grunnlagt i rettighetene og begrensningene gitt av lisensene de benytter. Hovedformålet er å maksimere bruken, veksten, utviklingen og distribusjonen av "free software". Tanken bak er at folk er kreativ og ønsker å skape nye ting, og det største hinderet for dette er hvis de ikke får tilgang til "byggestenene". I et åpen kildekode-prosjekt forsøker man dermed å skape en sosial struktur som ekspanderer i stedet for å begrense tilgangen til byggestenene som et felles gode. Dette gjøres gjennom en rekke lisenser som sikrer alle tilgang til en større og større kodebase som et felles gode [54].

Arkitektur formet av sosial organisering

Melvin Conway [6] kom for over 30 år siden med utsagnet at sammenhengen mellom arkitektur og organisasjon i programvareutvikling er drevet av kommunikasjonsbehovene til de involverte som forsøker å samarbeide best mulig. Av dette følger Conways lov som sier at strukturen til et system vil følge strukturer av organisasjonen som designet det. En problemstilling her er at programvarearkitektur er ustabil og i stadig endring, mens formelle organisatoriske strukturer raskt låser seg i faste mønstre og blir vanskelig å endre. Disse formelle strukturene kan til og med legge hindringer i veien for effektiv samarbeid.

Siden åpen kildekode-utvikling er på frivillig basis og av en uformell art står det ovenfor store utfordringer i utformingen av teknologisk arkitektur i forhold til Conways lov. Dette er en av grunnene til at teknisk rasjonalitet alltid er bygd på et kulturelt rammeverk og de organisatoriske karakteristikkene ved utviklingen. Når det snakkes om "clean code" er det ikke bare snakk om distinkte tekniske karakteristikker ved koden, men også hvordan den kan håndteres av et organisert fellesskap. Tekniske avgjørelser avhenger

dermed også av hvordan disse passer inn i den organisatoriske strukturen. For eksempel modularisering av kildekode reflekterer i stor grad de komplekse problemene ved samarbeid rundt paralellprosessering på stor skala i åpen kildekode-utvikling [54].

3.3 Diskusjon

Å utvikle kompleks programvare er en vanskelig og krevende oppgave. I 1986 skrev Frederick Brooks boka "No Silver Bullet: Essence and Accident of Software Engineering". En kontroversiell bok, men presenterte likefullt en talende beskrivelse av den underliggende strukturen til problematikken rundt programvareutvikling, og hvorfor den er vanskelig å forbedre. Brooks dro inspirasjon fra Aristoteles ved å separere problemstillingen i to typer programvareutviklingsproblemer: *essense* og *accident*. Essense er de iboende utfordringene som ligger i problemstillingen, mens accident er de vanskelighetene som inngår i en gitt omgivelse for utvikling, samt feil som oppstår som ikke er direkte relatert til oppgavens natur. Brooks nøkkelargument var her at de fundamentale utfordringene lå i essense, ikke accidents. Essense er det konseptuelle arbeidet som danner grunnlaget for en implementasjon: datasett, avhengigheter mellom data, algoritmene, etc. Selve implementeringen er en krevende jobb, men praktiske problemstillinger i den sammenhengen er i følge Brooks accidents. Accidents kan reduseres ved å forbedre prosessen. Hvis dette stemmer så vil enkle modeller feile fordi man ikke kan abstrahere bort den iboende kompleksiteten i oppgaven. For programvareutvikling blir kompleksiteten økt av at programvare blir brukt i et enorm antall ulike sammenhenger og i et kulturelt og teknologisk omgivelse som er i stadig forandring [54].

Nøkkelen til programvareutvikling er i følge Brooks konseptuell integritet, som arkitektonisk samsvar følger av en hovedplan. Dette gir visse fordringer for arbeidsfordeling. For å utvikle et godt design kan man ikke ha for mange mennesker involvert, og man må ha et klart skille mellom arkitekturplanen og implementasjonen. Dette gir klare assosiasjoner til Henry Fords tilnærming til industriell organisering med samlebånd og klar arbeidsfordeling hvor arbeidet blir overvåket, evaluert og avvik blir kompensert for. Dette er høyst effektivt for visse type produkter, men man ser en del problemer med denne innen programvareutvikling. Overvåkning og evaluering av en kompleks oppgave som programvareutvikling er dyrt og med mange feilkilder.

Stor innsats er blitt lagt i å motvirke disse problemene, men kanskje størst

suksess har vært på det Brooks mente var minst viktig, nemlig accidents. Hva er så gjort? åpen kildekode-utvikling og proprietær utvikling har både slående likheter og grunnleggende forskjeller i hvordan de forsøker å møte disse problemstillingene. Mange av åpen kildekode-utviklere jobber også med proprietær programvare. På samme måte er det sjelden å finne utviklere som bare benytter proprietær kode i sin utvikling. Det er en stor grad av kunnskapsutveksling mellom de to paradigmene til tross for ulikhetene. I både åpen kildekode-utvikling og proprietær utvikling er det en del egenskaper som er viktig for begge, men som utarter seg noe forskjellig [8].

Både i åpen kildekode-utvikling og proprietær utvikling er kodegjenbruk viktig og utbredt både av kostnadshensyn og tidsbruk. I mange bedrifter råder det såkalte "not invented here"-syndromet hvor man for enhver pris forsøker å unngå å bruke eksternt utviklet kode av ulike grunner. Men til og med disse benytter standard biblioteker, operativsystem-komponenter, kompilatorer og lignende ekstern programvare. Å starte helt fra bunnen av er meget sjelden et alternativ i dagens komplekse programvareindustri. I åpen kildekode-utvikling kan man ta dette et steg lenger. Siden kildekoden er tilgjengelig trenger man ikke å avhenge av eksterne leverandører for tilpasning, oppdatering og feilretting av koden man ikke selv utvikler. Man kan selv tilpasse og rette feil, noe som øker fleksibiliteten og ofte tiden det tar å rette feil [8].

Distribuert utvikling er et fundamentalt kjennetegn for åpen kildekode-utvikling, men dette er heller ikke helt ukjent i proprietær utvikling. Store selskaper har ofte mange avdelinger samt at *outsourcing* har blitt mer og mer utbredt. Proprietær utvikling står dermed også ovenfor utfordringene rundt internasjonalt spredte utviklingsteam som jobber med samme eller relaterte prosjekter. Verktøy som står sentralt her for begge utviklingsparadigmene er versjonskontrollsystemer, feilhåndteringssystemet, kommunikasjonsverktøy, og mekanismer for distribusjon [8].

Skalering er en relevant problemstilling for både proprietær og åpen kildekode-utvikling. Åpen kildekode har ingen magisk løsning på dette, modularisering er like viktig som i proprietær utvikling. De fleste Åpen kildekode-prosjekt har bare en håndfull hovedutviklere, og større prosjekter som Linux er sterkt modularisert for å kunne utvikles effektivt. For å kunne gjøre dette må modulene ha hensiktsmessige grensesnitt. Igjen har åpen kildekode her en fordel med at alt er åpent og dermed transparent når det gjelder samhandlingsevne [8].

Hvis vi så ser nærmere på utviklingsmodellene bak åpen kildekode-utvikling og proprietær utvikling ser man at de skiller seg fra hverandre på en del

vesentlige punkt. Utvikling av kompleks programvare stiller store krav, både teknisk og administrativt. Flere strategier for å håndtere denne kompleksiteten er forsøkt i tradisjonell utvikling. Prosessmodellene jeg har beskrevet tidligere er eksempler på strukturerte metoder som forsøker å håndtere denne kompleksiteten. Disse metodene har ofte i tråd med sin strukturerte natur problemer med å i stor nok grad være fleksible og tilpasningsdyktige. Programvareutvikling innebærer en rekke kreative prosesser som vanskelig kan struktureres til rutiner, og en rekke problemer vil stå udefinerte fordi de er på et høyere detaljnivå en metodene omfatter [25]. Tradisjonelle prosessmodeller har utviklet seg for å forsøke å motarbeide disse problemene. De har i stor grad gått i retning av å bli mer iterativ og fleksible, og gamle modeller som den rene fossefallsmodellen har vist seg å være lite skikket til å møte kravene til fleksibilitet og tilpasningsdyktighet i utviklingsprosessen. Like fullt er programvareutvikling med de mer fleksible modellene fremdeles en stor utfordring. Hvordan er så åpen kildekode-utvikling i forhold.

the Open Source Spiral Model kan se ut til å unngå en rekke av de vanlige problemene de tradisjonelle modellene står ovenfor. Dens utpregede iterative natur legger til rette for å kunne produsere god kode ved at endringer kan bli grundig sjekket ved neste runde i mikrospiralen. På denne måten kan man unngå at problemer først blir oppdaget etter at man har levert et produkt til en kunde [3]. Hvis man skal forsøke å direkte sammenligne denne modellen med tradisjonelle prosessmodeller kan den se ut til å være en ekstrem versjon av inkrementelle modeller innen "agile methods" og Boehms spiralmodell. På mange måter ligger den ganske nærme *Extreme Programming* med korte iterasjoner med fortløpende integrasjon og testing, tett forhold til kunde, felles kodeeierskap og individuell frihet.

Ved å utvikle programvare på en distribuert måte i et åpent fellesskap, så tillater man at brukere også kan være bidragsytere [25]. Dette er vesentlig, spesielt i lys av Hippels [18] funn som sier at mange nyvinninger kommer fra brukerne og ikke fra produsentene. Tradisjonell proprietær utvikling har ikke denne muligheten til åpent og direkte deltagelse av brukerne. Videre tillates en kontinuerlig debugging og design av programvaren, hvor kode kan bli tilpasses og endret i den desentraliserte strukturen, for så å føres tilbake inn i den offisielle kodebasen. Eric S. Raymond oppsummerer dette med "Given enough eyeballs, all bugs are shallow" [40, s. 2]. Den underliggende problemstillingen er her at det ikke er fullstendig overlapp mellom testerens mentale modell og utviklernes mentale modell for systemet. I tradisjonell proprietær utvikling sitter de fast i disse rollene og kan dermed ofte snakke forbi hverandre. åpen kildekode-utvikling kan derimot bryte dette mønsteret,

og legger i mye større grad til rette for at bruker og utvikler kan danne en felles forståelse basert på fritt tilgjengelig kildekode, og dermed kommunisere mer effektivt rundt denne [40, s. 10].

Med en så stor grad av brukermedvirkning skulle man tro at prosessen ble heller kaotisk. Diskusjoner på epostlister og andre elektroniske medium rundt utviklingen, hvilke egenskaper ønskes, hva bør det fokuseres på, hva virker og hva virker ikke osv foregår i tilsynelatende kaos hvor konflikter er vanlig. Dette betyr derimot ikke at prosessen ikke har noe styring overhode. Kommunikasjonen foregår som oftest i henhold til normer som sier at man kan få si det man mener, men når man kommer til innlemmelse av ny kode foregår det på en metodisk og systematisk måte. For eksempel kode som blir sendt inn for inkludering i Linux er det en fast prosedyre som må følges. Først er det forventet at koderen tester og evaluerer på egen hånd, samt får andre til å teste før koden blir sendt inn. Når koden er blitt sendt inn går den så igjennom et hierarki av *maintainers* som er ansvarlig for deler av koden før den blir godkjent eller avvist av de som sitter på toppen av hierarkiet. Mindre prosjekt har ikke nødvendigvis et så stort hierarki, men fungerer i stor grad på samme måte [54].

Kapittel 4

Kunnskap og programvareutvikling

Som beskrevet i forrige kapittel har tradisjonell programvareutvikling røtter i ingeniørvitenskapen. Fokuset her har vært på å beskrive hvordan programvareutvikling bør gjennomføres ved å anvende vitenskaplig og teknisk kunnskap. De formelle metodene og modellene i tradisjonell utvikling er opptatt av å kunne bryte ned utviklingen i strukturerte prosesser som resulterer i et produkt. Disse tar i liten grad hensyn til konteksten til systemet og utviklingen av systemet.

Programvareutvikling er en kompleks prosess som står ovenfor mange utfordringer. En stadig økende kompleksitet og omfang av programvaren samt at det teknologiske grunnlaget er i kontinuerlig endring gjør at det er behov for å samle kunnskap og erfaringer og skape gjenbruk av dette i programvareutviklingsprosessen. Det har derfor vært en gradvis endring i hvordan man ser på programvareutvikling, hvor man har gått bort det rene produksjonsynspunktet fra tradisjonell ingeniørvitenskap til å se på det som en intellektuell aktivitet med kunnskap og læring i fokus. Programvareutviklere sitter inne med høyst verdifull kunnskap om produktutvikling, selve utviklingsprosessen, prosjektstyring og relatert teknologi. Programvareutvikling er i stor grad kunnskapsintensivt arbeid, hvor kunnskap forekommer både implisitt og eksplisitt, som er i stadig endring pga påvirkning fra teknologi samt organisatoriske og sosiale strukturer [1]. Jeg mener man kan se et grunnlag for at dette skiftet i fokus er hensiktsmessig når man ser på hvordan åpen kildekode-utvikling behandler kunnskap og læring.

Kunnskap, kunnskapsforvaltning og læring er et stort fagfelt som blir for

omfattende å gå grundig inn på i denne oppgaven. Jeg vil begrense meg til å fokusere på hvilke mekanismer som ligger til grunn for lagring av kunnskap og læring og refleksjon rundt denne kunnskapen som finnes i henholdsvis proprietær og åpen kildekode-utvikling. Jeg vil gjøre dette med bakgrunn i I. Nonaka & H. Takechi [36] sitt arbeid rundt "knowledge-creation".

4.1 Mekanismer for "knowledge-creation"

I "A Theory of the Firm's Knowledge-Creation Dynamics" ser I. Nonaka & H. Takechi på kunnskap i japanske bedrifter. Fokuset er på kunnskapskreasjon i organisasjoner. Deres ontologiske grunnlag ligger i entiteter som skaper kunnskap. Kunnskap er i bunn og grunn bare skapt av enkeltmennesker, en organisasjon kan ikke skape kunnskap uten mennesker. En organisasjon kan derimot støtte opp under kreative mennesker eller skape en kontekst hvor kunnskap kan skapes. Organisatorisk "knowledge-creation" må dermed forstås som en forsterkende effekt på kunnskap skapt av enkeltmennesker og som fanger opp og fremhever denne som en del av kunnskapsnettverket til organisasjonen. Som epistemologisk grunnlag bruker de Michael Polanyi's skiller mellom taus kunnskap (*tacid knowledge*) og eksplisitt kunnskap (*explicit knowledge*). Taus kunnskap er personlig, kontekstspesifikk kunnskap som er vanskelig å formalisere og konkretisere. Eksplisitt kunnskap på sin side er kunnskap som er kodifisert og som kan formidles på en formell og systematisk måte. Polanyis argumenterer for at mennesker tilegner seg kunnskap ved å aktivt skape og organisere sine egne opplevelser. Dette vil si at eksplisitt kunnskap bare er en liten del av kunnskapen folk besitter [36].

I. Nonaka & H. Takechi ser ikke på eksplisitt kunnskap og taus kunnskap som adskilte, men heller som gjensidig komplementære. Kunnskapsformene samhandler og endrer form fra den ene til den andre i menneskers kreative aktiviteter. De legger til grunn for modellen antagelsen om at kunnskap blir skapt og formet gjennom sosial interaksjon mellom eksplisitt og taus kunnskap. De kaller denne interaksjonen for "knowledge conversion", som omfatter fire former for kunnskapsinteraksjon [36].

Sosialisering - fra taus kunnskap til taus kunnskap: Sosialisering er en prosess hvor man deler erfaringer og dermed skaper taus kunnskap i form av delte mentale modeller og tekniske ferdigheter. Uten disse delte erfaringer er det veldig vanskelig å sette seg inn i andres tankesett og tilegne seg taus kunnskap [36, 220].

Eksternalisering - fra taus kunnskap til eksplisitt kunnskap: Eks-

ternalisering er en prosess hvor man innskriver (inscribe) taus kunnskap til eksplisitte konsepter, som kan være i form av f.eks metaforer, analogier, konsepter eller modeller. Denne innskrevne kunnskapen er ofte utilstrekkelig og inkonsistent, som gjør at det ofte kreves refleksjon over kunnskapen eller interaksjon med andre individer for å tolke den [36, s. 221].

Kombinasjon - fra eksplisitt kunnskap til eksplisitt kunnskap: Kombinasjon er en prosess hvor man systematiserer konsepter i kunnskaps-systemer. Dette innebærer å kombinere forskjellig eksplisitt kunnskap, og endring av informasjon gjennom sortering, tillegg, kategorisering, osv som kan føre til ny kunnskap [36, s. 222].

Internalisering - fra eksplisitt kunnskap til taus kunnskap: Internalisering er en prosess hvor man går fra eksplisitt kunnskap til taus kunnskap. Denne prosessen er nært knyttet til "learning by doing". Når man internaliserer erfaringer gjennom sosialisering, eksternalisering og kombinasjon skaper man taus kunnskap i form av delte mentale modeller og teknisk kunnskap [36, s. 222].

Kunnskap blir skapt av individer som organisasjonen så må fange opp og forsterke gjennom de fire formene for kunnskapsinteraksjon og krystallisere denne på et høyere ontologisk nivå. I. Nonaka & H. Takechi kaller dette for en kunnskaps-spiral hvor kunnskap skalerer når den forflytter seg oppover de ontologiske nivåene. Kunnskapkreasjon i organisasjoner går dermed fra individuelt nivå opp til stadig større fellesskap som samhandler med hverandre; grupper, avdelinger, organisasjoner, etc [36].

Rollen til organisasjonen i kunnskapskreasjonsprosessen er å skape en velegnet kontekst for å skape og dele kunnskap. I. Nonaka & H. Takechi identifiserer fem nødvendige forutsetninger for at kunnskaps-spiralen skal fungere: målsetninger, autonomi, variasjon og "kreativt kaos", redundans, tilrettelegging for mangfold.

Kunnskapsspiralen er drevet av organisasjonens målsetninger og strategier for å tilegne seg, skape, ta vare på og utnytte kunnskap. Det mest kritiske elementet i en organisasjons strategi er å ha en visjon for hvilken type kunnskap man skal utvikle og skape et kunnskapsforvaltningsystem på bakgrunn av. Denne visjonen legger også grunnlaget for å kunne bedømme kvaliteten og verdien av kunnskapen for organisasjonen.

Ved å la folk ha størst mulig autonomitet øker man sjansene for at de kan bidra med ny og uventet innsikt og være motivert til å skape ny kunnskap. Originale idéer kommer fra autonome individer, blir spredt innad i teamet, for så å bli organisatoriske idéer.

Den tredje faktoren er variasjon og "kreativt kaos", som stimulerer interaksjonen mellom organisasjonen og eksterne omgivelser. Variasjon og svingninger er ikke det samme som fullstendig uorden, I. Nonaka & H. Takechi beskriver det som "order without recursiveness" [36, s. 228]. Variasjon bryter ned faste rutiner, vaner og kognitive rammeverk. Dette gjør at man reflekterer over det man jobber med og får et nytt perspektiv. "Kreativt kaos" oppstår når en organisasjon står ovenfor en krise som øker spenningen innad og fører til at man får større fokus på å finne en løsning på problemet man står ovenfor.

Redundans er ofte noe man vil unngå i effektivitetens navn, men når det gjelder kunnskap er dette ofte hensiktsmessig. Kunnskap bør deles mellom avdelinger som ikke nødvendigvis behøver denne umiddelbart. Redundans vil hjelpe å formidle skjult kunnskap ved å gi en bedre kontekst. Dette øker informasjonsmengden og må derfor evalueres opp mot kostnadene ved å prosessere informasjonen.

Den siste faktoren er å legge til rette for mangfold. For at en bedrift skal kunne tilpasse seg en dynamisk og mangfoldig omgivelse må den selv ha et stort nok mangfold til å kunne møte utfordringene omgivelsen stiller. Individer i organisasjonen kan møte en rekke utfordringer så lenge de har det nødvendige mangfoldet for å aksessere, kombinere og prosessere informasjon på en effektiv måte.

4.2 Kunnskap i tradisjonell programvareutvikling

Programvareutvikling er som tidligere sagt et kunnskapsintensivt fagfelt. Kunnskap og kunnskapsforvaltning er relevant for en rekke ulike aspekter av utviklingsprosessen, fra strategiske og organisatoriske problemstillinger ned til de rent tekniske. Disse inkluderer blant annet [1]:

- Tids- og kostnads-estimering
- Prosjektstyring
- Kommunikasjon med klienter og brukere
- Problemløsning i programvareutviklingen
- Gjenbruk av kode
- Opplæring og kompetansehevelse av ansatte

- Vedlikehold og brukerstøtte

Man kunne derfor anta at programvareutviklere i tradisjonell utviklingsmiljøer hadde stor kompetanse på kunnskapsforvaltning, men det er lite som tyder på at dette er tilfellet. Det finnes en rekke kunnskapsforvaltningssystemer som er i bruk i dag, men få av disse ser ut til å fokusere på programvareutvikling. Hvis man ser nærmere på programvareutvikling fra et kunnskapsforvaltningsperspektiv finner man en rekke relevante problemstillinger som kan være med å forklare dette [1].

Man kan sette et grovt skille i programvareutvikling mellom utvikling og vedlikehold/kundestøtte. En viktig egenskap ved dette skillet er at de ofte blir utført av adskilte grupper mennesker. Dette skillet fører til problemer for kunnskapsforvaltning fordi det er viktig å kunne dele kunnskap mellom disse to gruppene. Videre innebærer programvareutvikling både organisatoriske og tekniske aspekter som må spille sammen. Programvareutviklingsteam er også lite stabile, med stor utskiftning av personer. Dette medfører at kunnskap ofte blir knyttet til enkeltpersoner i stedet for til en større gruppe. Kravene til kompetanse går også fra det mest generelle til ytterst spesifikk kunnskap og kan ofte komme i konflikt når man skal ta designavgjørelser. Utviklingsprosjekt er alltid i stadig endring og må balansere mellom rask respons til problemer og hensyn til hele livsløpet til programvaren. Man må derfor ta stilling til hvilken kunnskap skal man ta vare på, og hvilken skal man forkaste. Det er altså en stor rekke problemstillinger som har innvirkning på kunnskapsforvaltning i programvareutvikling [1].

Fra et organisatorisk synspunkt går kunnskapsforvaltningsprosessen gjennom flere steg. Først blir kunnskapen skapt/innhentet, deretter går den igjennom en syklus: beholde og lagre kunnskapen, bruke kunnskapen, raffinere og oppdatere kunnskapen. I tillegg kan den bli delt med utenforstående i parallell med kunnskapssyklusen. Når man så betrakter denne prosessen i forhold til programvareutviklingsaktiviteter, ønsker man å kunne integrere den i de ulike aktivitetene på en effektiv måte. Aurum et. al. presenterer tre slike løsninger: teknologisk, menneskerelatert og prosess [1].

Teknologiske løsninger omfatter installasjon av ny teknologi og mer effektiv bruk av eksisterende teknologi. Spesifikke teknologier kan inkludere data mining, databaser og intranettløsninger. Aktiviteter omfatter standardisering av programvare og maskinvare, strømlinjeforme systemer ved å fjerne duplikate systemer eller data.

Menneskerelaterte løsninger fokuserer på personalutvikling og å beholde kompetanse innad i bedriften. Opplæring, skape en god bedriftskultur, og

øke motivasjonen blant de ansatte er viktige elementer her.

Prosessløsninger omfatter prosess-spesifikasjoner og instruksjoner, men også på samspillet mellom formelle og uformelle metoder for kunnskapsdeling. Fokuset er på å jobbe smartere og mer effektivt og med bedre oversikt over hva som foregår i bedriften.

Jeg vil avslutte dette avsnittet med å presentere to overordna strategier for kunnskapsforvaltning. Disse er *kodifisering* og *personifisering* som presentert av Hansen et. al. [15]. Kodifiserings-strategier er vanligvis assosiert med teknologiske løsninger som intranett og kunnskapsbaser. Personifiserings-strategier på sin side fokuserer på mellom-menneskelige løsninger som "communities of practice" og historiefortelling [1].

Kodifiserings-strategien

Kodifiserings-strategien faller seg naturlig for tekniske aktiviteter, hvor en løsning for et problem relativt lett kan la seg overføre til et nytt problem. For rene programmeringsproblemstillinger ser denne strategien dermed ut til å være hensiktsmessig. Det har også være gjort en stor innsats i å utvikle verktøy for å støtte design og utvikling, som for eksempel Computer Aided System Engineering (CASE) verktøy og workbench verktøy. Disse er nyttig for å lagre, dele og benytte kunnskapsforvaltningsaktiviteter, men er i mindre grad i stand til å raffinere kunnskap og ikke velegnet overhode for å skape ny kunnskap.

Problemsporing, problemløsning og metodedokumentasjon er andre kategorier av kunnskapsforvaltning som passer inn i denne strategien. Fokuset har i stor grad ligget på å samle inn og lagre kunnskap for enkeltprosjekt, mens det hadde vært mye å hente på effektiv deling av analyser og designkunnskap også mellom uavhengige prosjekter.

Personifiserings-strategien

Mens kodifiserings-strategien ser ut til å passe best for tekniske aktiviteter, er personfiserings-stretegien naturlig nok mer velegnet for prosjektstyring og organisatoriske aktiviteter. Personfiserings-strategien kan være veldig effektiv for å skape og raffinere kunnskap, samt å dele og lagre denne kunnskapen. Den er mindre velegnet til å hjelpe med å kunne benytte denne kunnskapen på en effektiv måte. I programvareutvikling vi denne strategien være hensiktsmessig

for styring av menneskelige ressurser, sammen med mer overordnet tekniske aktiviteter som analyse og implementasjon hvor kunnskap står i fokus.

4.3 Kunnskap i åpen kildekode-utvikling

Åpen kildekode-programvareutvikling basererer seg i stor grad på mer implisitte prosesser i motsetning til de eksplisitte prosessene som ligger til grunn for tradisjonell programvareutvikling. Tradisjonelle utviklingsaktiviteter har dermed ikke samme statusen som eksplisitt uttalte prosesser som blir tildelt de enkelte deltakerne. Det viser seg derimot at i stedet for disse tradisjonelle metodene så er det sosiotekniske prosesser som har et større fokus og som er de viktigste. Disse går på å utvikle konstruktive sosiale forhold, uformelt forhandlede sosiale arrangementer, og et engasjement og forpliktelse til å vedlikeholde og bidra med programvare og representasjoner [41, s. 22].

Utviklere står ovenfor en stor utfordring i forbindelse med deling av taus kunnskap. "Physical activities and face-to-face interaction are the key to sharing tacit knowledge." [35]. Utviklingen blir i veldig stor grad utført av mennesker som kommuniserer elektronisk og er fysisk adskilt. For å overkomme dette problemet benytter prosjektdeltakere en rekke metoder. Hemetsberger og Reinhardt beskriver hvordan kunnskap i åpen kildekode-miljø blir delt og reflektert over ved hjelp av prosesser og tekniske løsninger som muliggjør det de kaller "re-experience" [16, s. 1] av kunnskapen for de involverte.

Hemetsberger og Reinhardt så på KDE-prosjektet[1], som er et desktop-miljø for GNU/Linux-systemer for å se hvordan kunnskap ble "re-experienced" av deltakerne. De beskriver tre prosesser som ble brukt for å oppnå dette.

4.3.1 "Re-experience" gjennom redusert kompleksitet og "transactive group memory"

Det er kritisk å bygge opp og organisere ny informasjon i prosjektet. For å håndtere de store mengdene informasjon benyttes en rekke løsninger for å redusere kompleksiteten til informasjonen, og for å strukturere den. Feilhåndteringssystem, versjonskontroll av koden, og epostlister er eksempler på dette [16, s. 10].

[1] www.kde.org

Disse verktøyene bidrar til å redusere kompleksiteten ved å muliggjøre modularisering og distribuering av arbeidsoppgaver i tillegg til at informasjonen kan fanges opp og lagres på en strukturert måte. Bug-trackere brukes for å registrere feil og ønsker fra brukere. Disse blir kategorisert og øremerket bestemte utviklere som får i oppgave å fikse de. Versjonskontrollsystemer benyttes for å holde styr på endringene som skjer, og for å fasilitere en distribuert utviklingsmodell. På denne måten kan endringer inkorporeres på en strukturert måte og man kan få oversikt over hva andre har gjort.

Epostlistene brukt i KDE-prosjektet ble funnet å være det viktigste verktøyet i prosjektets kunnskapssystem, hvor saker blir diskutert, reflektert rundt og lagret [16, s. 11]. All denne informasjonen blir lagret og gjort tilgjengelig for alle involverte. På denne måten kan man si at informasjonen representerer felleskapets delte minne, som blir uavhengig av de enkelte individene (*transactive memory*) [16, s. 13]. Den blir således tilgjengelig i den formen den utviklet seg i, og lar deltakerne følge hele diskusjoner og reflektere over disse.

4.3.2 "Re-experience" gjennom veiledning, åpenhet og deltagelse

Integrasjon av nye medlemmer skjer gjennom nedfelte prosedyrer og veiledning gjennom regler og normer som gjelder for prosjektet. Det er kritisk for nybegynnere å kunne observere de eksisterende sosiale normer og praksis for å kunne bli en del av fellesskapet og kunne ta del i i det delte minnet via "re-experience". Den reflekterende naturen til den tilgjengelige informasjon gjør at nye medlemmer kan sette seg inn i prosjektet uten å måtte ty til direkte interaksjon med andre deltakerne [16, s. 13].

En annen nøkkelfaktor er fri tilgang til all tidligere kommunikasjon og kode. Nye medlemmer kan følge tidligere kommunikasjon og prosesser og dermed være i stand til å "re-experience" læringen andre medlemmer har undergått [16, s. 16].

4.3.3 "Re-experience" gjennom asynkrone kommunikasjonskanaler og virtuell eksperimentering

Ny kunnskap blir i stor grad til via asynkrone kommunikasjonsverktøy. Synkrone kommunikasjonsverktøy tillater ikke videre refleksjon gjennom "double-loop" og dermed skaping av ny kunnskap. Asynkrone verktøy derimot

legger til rette for bruk av tekniske historier, scenarioer og refleksjon rundt disse. På denne måten blir kunnskap transformert, reflektert rundt, og kombinert til ny kunnskap av fellesskapet som en helhet, hvor fokuset blir på forståelse av problemet [16, s. 18-19].

Å skape ny kunnskap avhenger av om deltagerne er i stand til å reflektere over det de gjør på et metanivå og tenke over arbeidet på en måte Argyris (1992) kaller "double-loop manner". Argyris definerer "single-loop learning" som det å se på læring som bare ren problemløsning. "Double-loop learning" derimot går ut over det å bare å forstå følgende av en gitt handling. Ved "Double-loop learning" så avsløres og reflekteres det over de ubevisste mentale modellene som ligger til grunn for en gitt handling [16, s. 6]

4.4 Diskusjon

I tradisjonelt programvareutvikling har man begynt å gå bort produksjonssynspunktet på programvareutvikling til å se på det som en intellektuell aktivitet. Kunnskap og kunnskapforvaltning har fått et mye større fokus. Hvis man ser på kunnskap i proprietær utvikling i forhold til I. Nonaka & H. Takechi sin kunnskapsspiral ser man at det ulike formene for kunnskapsinteraksjon har ulik dekning avhengig av hvilken strategi som er valgt. Kodifiserings-strategien fokuserer på å bygge raske og pålitelige systemer av høy kvalitet ved å utnytte gjenbruk av kodifisert kunnskap. Denne baserer seg på systemer som kodifiserer, lagrer, forvalter og legger til rette for gjenbruk. Man henter ut kunnskap fra enkeltmennesker og gjør den uavhengig av den personen. Personifiseringsstrategien på sin side fokuserer på å skape kreative, analytiske og strengt strukturert kunnskap om overordna strategiske problemer ved å basere seg på individuell ekspertise. Forsøker å skape nettverk for å bringe folk sammen slik at de kan få delt taus kunnskap. Dialog og direkte kontakt blir brukt over kodifisert kunnskap i datasystemer i denne strategien [15].

Kunnskapsforvaltning i programvareutvikling har flere bruksområder, blant annet problem- og løsning-oppfølging, metodedokumentasjon og håndtering av menneskelige ressurser. I tillegg kan effektiv kunnskapsforvaltning legge til rette for mer deling av informasjon fra analyse og design fasene. Dette gjøres best på en desentralisert måte, mens kunnskapsforvaltningsstrategier er i stor grad top-down og blir dermed kunstig adskilt fra selve utviklingsprosessen. Et studie av Kautz et al. [23] så på rollen IT-system har i kunnskapsforvaltning hos en mindre dansk programvareutvikler og fant at IT-systemer var viktig,

men likevell underordnet åpenhet, tillit og respekt blant de ansatte for å legge til rette for læring [1].

Åpen kildekode-prosjekter er nesten utelukkende administrert over Internett. Effektiv bruk av kunnskap krever at man konsentrerer kunnskapsressursene i tid og sted. Internett muliggjør å samle og arkivere kunnskapsressurser og skape plattformer som konsentrerer og gjør kunnskap tilgjengelig. Disse plattformene er meget velegnet til å formidle kodifisert kunnskap, mens skjult kunnskap vanskelig lar seg formidle på denne måten som er et hinder for effektivt samarbeid [16].

Hemetsberger og Reinhard identifiserte bruk av mekanismer og verktøy for det de kaller "re-experience" av kunnskap for å overkomme disse problemene. Dette gikk på alt fra verktøy for å redusere kompleksiteten, til kulturelle normer, kommunikasjonsverktøy, individuell og kollektiv refleksjon og beskrivelser og scenarioer. Spesielt kulturen med grunnlag i frihet, åpenhet og ønsket om å hjelpe ble funnet å være en grunnstein for å skape og dele kunnskap. Ved at store deler av kommunikasjonen blir arkivert muliggjør at deltagerne kan observere denne sosiale praksisen og reflektere rundt den. På denne måten kan man si at de som observerer og lærer ikke tilegner seg kunnskap fra enkeltpersoner, men fra hele det sosiale kollektivet. Gjennom refleksjon over andres synspunkter og sine egne skaper kollektivet en felles forståelse rundt gitte problemstillinger, men *ikke* løsninger fordi dette vil hindre videre refleksjon. På denne måten støtter åpen kildekode-utvikling kunnskapsforvaltning gjennom nettbaserte felleskap ikke bare for de nåværende involverte, men også for nye deltagere i fremtiden [16].

Mens kunnskapsforvaltning i proprietær utvikling er eksplisitt og ofte etablert ovenfra gjennom egne systemer for å fange opp kunnskap, ser man at kunnskapsforvaltning i åpen kildekode-utvikling er implisitt og innfelt i sosiale og teknologiske konstruksjoner. Åpenheten i utviklingen legger også i større grad til rette for kunnskapsutveksling. Både proprietær og åpen kildekode-utvikling benytter seg i stor grad av kodegjenbruk. Åpen kildekode derimot tillater noe mer, nemlig kunnskapsgjenbruk via koden. Ved å granske selve kildekoden kan utviklere lære hvordan hvordan gitte problemer har blitt løst, og om denne løsningen er en generell løsning på problemer av samme type [8].

Hva bruker så utviklerne i åpen kildekode-utviklingsspiralen. Hvis man ser på denne i lys av I. Nonaka & H. Takechi sine forutsetninger for at kunnskapsspiralen skal fungere, ser man at disse i stor grad også reflekteres i utviklingsspiralen. Autonomitet i åpen kildekode-prosjekt er nærmest absolutt, utviklerne jobber på det de har lyst til å jobbe med. Videre ser

man stor grad av variasjon og "kreativt kaos" som fostrer problemløsning og refleksjon, selv om man kan si at det ikke er stor grad av organisatoriske barrierer som trenger å brytes ned i åpen kildekode-utvikling. Man finner også stor grad av redundans ved at utviklingen består av frivillig parallell bearbeidelse av et problem. Mangfold er også i stor grad tilstede ved at man har utviklere fra et stort antall ulike miljøer og med ulik bakgrunn og motivasjon.

Eksplisitte målsetninger finnes derimot ikke uttrykt på samme måten i åpen kildekode-utvikling, foruten på et veldig overordnet nivå hvor man ønsker å skape best mulig programvare, og i de målene hver enkelt setter seg. Det er derimot lite eksplisitte organisatoriske målsetninger som styrende faktor, men heller implisitte normer og regler. Når man så ser på flyten i utviklingen finner man at denne ligger relativt tett opp til kunnskapsspiralen til I. Nonaka & H. Takechi. Enkeltutviklere kommer individuelt opp med nye ideèr, som så blir delt med fellesskapet. Den blir så diskutert, argumentert rundt og eventuelle alternativer blir fremsatt. En eller flere av disse blir så implementert som igjen fører til mer diskusjon. Denne kommunikasjonen bygger opp en felles forståelse rundt problemstillingen og åpner for refleksjon gjennom "re-experience" [16].

Kapittel 5

Verktøy i programvareutvikling

I dette kapittelet vil jeg se nærmere på verktøy som er i utstrakt bruk i henholdsvis tradisjonell og åpen kildekode-programvareutvikling. Deretter vil jeg se nærmere på ulike typer verktøyintegrasjon, før jeg avslutter ved å se på de to paradigmene opp mot hverandre og diskutere hvilke typer integrasjon som vil være hensiktsmessig.

5.1 Verktøy i tradisjonell programvareutvikling

Bruken av verktøy i tradisjonell programvareutvikling blir ofte referert til som Computer Aided Software Engineering (CASE). Van Vliet [50] deler disse inn i en følgende kategorier: Verktøy (tool), arbeidsbenk (workbench), Miljø (Environment), en klassifisering med utgangspunkt i hvor godt de dekker de ulike fasene i utviklingsprosessen. Verktøy er programvare som tilbyr spesifikk funksjonalitet for å utføre en bestemt oppgave i utviklingsprosessen. Arbeidsbenk-programvare tilbyr funksjonalitet for en begrenset del av utviklingsprosessen, som f.eks analyse eller design. Et miljø støtter store deler av eller hele utviklingsprosessen. Arbeidsbenk-programvare og miljø omfatter et stort utvalg programvare som utmerker seg ved hvordan de ulike delene de består av er integrert med hverandre. Van Vliet [50] deler disse videre inn i kategoriene: "toolkit", språkspesifikke miljø, integrerte miljø/arbeidsbenker.

Toolkits består av en rekke verktøy for å utføre spesifikke veldefinerte oppgaver, og er generelt lite integrert med hverandre. Funksjonaliteten som tilbys er uavhengig programmeringsspråk og utviklingsparadigmer, det tilbyr

i stedet en rekke byggestener som kan kombineres. Et godt eksempel på *toolkits* er kommandolinjeverktøy for utvikling i systemer som UNIX og GNU/Linux, som tilbyr en rekke uavhengige verktøy som kan brukes separat eller i kombinasjon.

Språkspesifikke miljø inneholder verktøy som er tilpasset utvikling i et bestemt programmeringsspråk. Disse har ofte semantiske og syntaktiske hjelpemidler for det støttede språket. Systemer av denne typen fokuserer på å tilby hjelpemidler i implementasjon og testfasene for enkeltbrukere, og har lite støtte for deling av informasjon og funksjonalitet mellom flere brukere.

Integrerte systemer fokuserer på deling av informasjon innad mellom de ulike verktøyene som utgjør systemet. Disse sentrerer ofte rundt et datalager som inneholder data som omfatter alt fra selve kildekoden til kravspesifikasjon og dokumentasjon om selve prosessen.

Analyse-arbeidsbenker fokuserer på å støtte de tidlige fasene i utviklingsprosessen: kravspesifisering og analyse, og design. Har funksjonalitet for diagramgenerering, dataanalyse for konsistens, kompletthet og avhengigheter, og generering av rapporter og dokumentasjon. Videre kan de støtte prototyping, grensesnittbygging, og kodegenerering. Programmeringsarbeidsbenker tilbyr verktøy for implementasjons- og test-fasene: håndtering av kode, debugging, generering av testdata og simulering. Videre har mange en meget viktig funksjon for å støtte teamarbeid, nemlig et sentralt kodelager som støtter versjonskontroll. Forvaltnings-arbeidsbenker (Management Workbenches) fokuserer på verktøy for planlegging, kontroll og overordna styring av prosjekt. Dette omfatter blant annet kontroll med kravspesifikasjonsendringer, tildeling av arbeidsoppgaver, kostnadsestimering og kvalitetskontroll.

Integrerte prosjektmiljø forsøker å støtte alle faser i systemutvikling. Disse inneholder alle verktøyene diskutert over, og bruker å ha et fokus på enten analyse og design eller implementasjon og testing. Disse har en varierende grad av integrering mellom de ulike delene de består av. Denne typen verktøy bygger også rundt et sentralt versjonskontrollert kodelager for kode og dokumentasjon. En variant av integrerte prosjektmiljø fokuserer på prosessen. Disse er tett knyttet til en bestemt prosessmodell, og tilbyr funksjonalitet i tråd med en gitt formell modell.

5.2 “Software informalisms”

Scacchi fant i sin studie at funksjonelle og ikke-funksjonelle krav for åpen kildekode-programvare blir fremsatt, analysert, spesifisert, validert og håndtert gjennom en rekke nett-baserte beskrivelser som han kaller “software informalisms” [41, s. 19]. Disse nett-baserte verktøyene er også viktig for andre deler av utviklingsprosessen og jeg vil her kort beskrive de mest sentrale verktøyene.

5.2.1 “Community communications”

I mangel av et felles fysisk samlingsted samler åpen kildekode-fellesskap seg rundt en kommunikasjonsinfrastruktur. Denne kommunikasjonen foregår i følgende former: a) meldinger sendt til nettbaserte forum, b) epostlister, c) nyhetsgrupper, og d) Internett-basert lynmeldingstjenester (*instant messaging*) [41, s. 19].

Nettbaserte forum

Internettforum er en tjeneste på verdensveven (*World Wide Web*) for å holde diskusjoner. De tilbyr lignende funksjonalitet som oppringningstjenester som elektroniske oppslagstavler (*bulletin boards*) og nyhetsgrupper gjorde på 80- og 90-tallet. Internettforum er organisert ved ett eller flere forum hvor meldinger legges ut i tråder. En tråd er en samling meldinger hvor man svarer på hverandres innlegg.

I åpen kildekode-prosjekter er forum ofte i mer utstrakt bruk blant brukere og perifere bidragsytere enn aktive utviklere. Internettforum er lett tilgjengelig og er egnet for brukerstøtte og diskusjon rundt bruken av programvaren, mens utviklere i stor grad ser ut til å foretrekke epostlister for diskusjon rundt selve utviklingen. Internettforum har ofte stor trafikk og mye støy (uønskede innlegg) som gjør at det blir mindre velegnet enn abonnent epostlister, samt at epost er mer fleksibelt med tanke på filtrering, sortering, levering og lagring som er viktig for effektiv kommunikasjon og refleksjon.

Epost-lister

Epostlister en spesialisert benyttelse av epost som tillater utbredt og effektiv distribusjon til mange mottakere over Internettet. Programvare blir installert

på en server som prosesserer innkommende epost og avhengig av innhold enten behandler de internt eller distribuerer de videre til alle brukere som er medlemmer av epostlisten.

Epostlister er i stor grad den viktigste kommunikasjonskanalen i de aller fleste åpen kildekode-prosjekt. Hemetsberger og Reinhardt fant at KDE, som et av de største åpen kildekode-prosjekt med 800 utviklere, brukte en rekke epostlister som deres viktigste kommunikasjonsverktøy [16, s. 9].

Nyhetsgrupper

Nyhetsgrupper er et lager (repository) i Usenet-systemet [1] for meldinger sendt av mange brukere fra en rekke ulike stedet. Det kan ses på som en diskusjonsgruppe, hvor nyhetsgrupper er teknisk forskjellig, men funksjonelt tilnærmet ekvivalent med diskusjonsforum på verdensveven. Nyhetsgrupper krever egen programvare for å lese meldingene i de ulike gruppene.

Dagens format og sendingmetode av melding i Usenet har mye til felles med hvordan epost fungerer, og innen åpen kildekode-utvikling benyttes nyhetsgrupper i stor grad på samme måte som epostlister. Usenet er et en-til-mange medium og gjør dermed det samme som epostlister gjør for epost. Nyhetsgrupper er delt inn i kategorier som er organisert i hierarki.

Lynmeldingstjenester

Lynmeldinger (*Instant Messaging*) er synkron kommunikasjon mellom to eller flere personer over et nettverk som for eksempel Internett. Lynmeldinger krever at man benytter klientprogramvare som kobler seg til en lynmeldingstjeneste, hvor kommunikasjonen foregår i sanntid. De fleste tjenester tilbyr statusinformasjon som viser om brukere er tilgjengelig for kommunikasjon og en liste over kontakter som man kan kommunisere med.

Det finnes en rekke lynmeldingstjenester, blant annet MSN Messenger, AOL Instant Messenger, Yahoo! Messenger, Skype og Google Talk. Innen åpen kildekode-utvikling derimot er det den langt eldre, men fremdeles populære, tjenesten IRC (Internet Relay Chat) som har størst utbredelse. IRC er i hovedsak designet for mange-til-mange kommunikasjon i såkalte *kanaler*, men kan også benyttes for en-til-en kommunikasjon.

[1] Usenet er et distribuert Internett diskusjonssystem som involveret et generelt UUCP netverk av samme navn (http://en.wikipedia.org/wiki/Usenet)

5.2.2 Nettbasert informasjon

Åpen kildekode-prosjekt har en utstrakt bruk av nettbasert informasjon. Scacchi [41, s. 19] fant at utviklerne benytter nettbasert informasjon for å dele den han kaller "mentale konstruksjoner" for hvordan et gitt system skal fungere. Siden deltakerne er fysisk adskilt skjer dette gjennom en rekke artefakter som skjermbilder, guidede turer eller navigerbare lenkede nettsider. Formålet med disse er å formidle hva som er ønsket å oppnå med systemet og hvordan systemet forstås av vedkommende, og kan sies å fylle samme rolle som mer formelle metoder som *Use Cases* i tradisjonell utvikling.

Internett blir i stor grad brukt som en informasjoninfrastruktur for publisering og deling av beskrivelser av programvare i form av lenkede nettsider, wikier, forum og indeksert informasjon. Nettsider blir også brukt til å publisere how-to's og FAQ's (Frequently Asked Questions). How-to's fanger inn oppførsel og funksjonalitet ved et system, og tjener som en semistrukturert beskrivelse som fastsetter sluttbrukerkrav. FAQ's er mer uformelle, og inneholder spørsmål og svar til spesifikke individuelle problemstillinger.

5.2.3 Feilhåndteringssystemer

Nettbaserte systemer blir brukt for å lagre og håndtere feilrapporter, oppdateringer (*patches*) og andre innrapporteringer som kan gå på for eksempel ønsket om ny funksjonalitet. Dette kan være alt fra rene epostlister til dedikerte nettbaserte systemer som Bugzilla [2].

5.2.4 Dokumentasjon

Også åpen kildekode-programvare har tradisjonell systemdokumentasjon, både for utviklerne og sluttbrukere. Scacchi [41, s. 21] fant at denne ofte var utdatert og hadde mangler, men at leserne ved hjelp av de andre tilgjengelige informasjonskildene ofte var i stand til å identifisere dette og jobbe rundt det. Videre finnes det eksterne publikasjoner, da gjerne tekniske artikler, som identifiserer og beskriver funksjonelle og ikke-funksjonelle krav i åpen kildekode-prosjekter.

[2] http://www.bugzilla.org/

5.3 Verktøyintegrasjon

Verktøyintegrasjon går på relasjonene mellom verktøy som samhandler med hverandre. Yang og Han [58] definerer hovedformålet med verktøyintegrasjon ganske enkelt som økt produktiviteten for utviklerne. Integrasjonen kan foregå på flere nivåer i en omgivelse. Thomas og Nejmeh [48, s. 29] vektlegger at integrasjon ikke bare må defineres i forhold til et spesifikt verktøy, men at man også må fokusere verktøyets relasjoner med andre elementer i omgivelsen som omfatter andre verktøy, plattformen og selve prosessen, med spesiell vekt på relasjonene mellom ulike verktøy.

Både Yang og Han [58, s. 56] og Thomas og Nejmeh [48, s. 30] trekker frem at verktøyintegrasjon kan ses fra to ståsted. Tradisjonelt har det blitt fokuserer på mekanismene ved integrasjon sett fra en utvikler sitt ståsted som er velegnet for å finne de relevante aspektene ved integrasjon, men som har vist seg å være av begrenset nytte for å oppnå effektiv verktøyintegrasjon. Et alternativt ståsted er å se på verktøyintegrasjon fra en bruker sitt ståsted hvor fokuset er økt produktivitet for brukeren og hvordan grensesnittet oppfattes i bruk. De to ståstedene skiller seg ved at en utvikler ønsker lett integrerbare verktøy, mens en bruker ønsker seg godt integrerte verktøy.

Integrerte omgivelser må dermed anerkjenne to typer grensesnitt; grensesnittet verktøyomgivelsen presenteres til brukeren og grensesnittet mellom komponentene innad i omgivelsen. Førstnevnte er relatert til presentasjon av grensesnittet til brukeren, mens sistnevnte omfatter en rekke ulike typer integrasjon. Det varierer hvilke kategorier som blir brukt her. Yang og Han beskriver kontroll- og data-integrasjon [58, s. 56] mellom de ulike verktøyene omgivelsen består av, mens Thomas og Nejmeh [48, s. 30] baserer seg på Anthony Wasserman [53] sin inndeling som i tillegg til å omfatte presentasjon-, kontroll- og data-integrasjon også omfatter plattform- og prosess-integrasjon.

Presentasjonsintegrasjon går på å øke brukerens effektivitet i interaksjon med systemet ved å redusere den kognitive belastningen. Måter å oppnå dette på er blant annet å benytte et minimum av presentasjons- og interaksjonsparadigmer som i størst mulig grad stemmer overens med brukerens mentale modeller, innfri brukerens forventninger til responstid og kvalitetssikre informasjonen som blir gitt. Et viktig aspekt her er i hvor stor grad en kan overføre kjennskap fra et verktøy over til et annet både med tanke på utseende, oppførsel og interaksjonsparadigmer [48, s. 30-31].

Dataintegrasjon omfatter både deling av data mellom komponenter og administrasjon av relasjonene mellom data produsert av de ulike komponentene.

Det finnes et utall måter å behandle vedvarende og ikke-vedvarende data. Meyers [30] presenterer i sin taksonomi for verktøy følgende kategorier:

- Mellomlagring via filer
- Database eller objektbase
- Meldings-sending
- Kanonisk representasjon

For en høy grad av integrasjon er det flere aspekter som er viktig. Man ønsker å vedlikeholde informasjonen i en konsistent form uavhengig av hvordan de enkelte komponentene behandler og omformer den. I hvor stor grad de enkelte verktøyene må behandle dataene for å kunne utnytte den bestemmer graden av samspillsevne. Man ønsker også lav redundans og stor grad av synkronisering av data verktøyene deler [48, s. 31-32].

Kontrollintegrasjon går på at de individuelle komponentene må kunne kommunisere effektivt seg i mellom. Yang og Han [58, s. 56] beskriver fire metoder for kommunikasjon:

- Indirekte kontrollintegrasjon - aktivering av verktøy via fasiliteter som også er tilgjengelig direkte for brukere, som for eksempel systemkall.
- Triggere - aktivering av verktøy gjennom triggere implementert som events i en database eller objektbase.
- Meldingstjener - aktivering av verktøy via meldinger sendt ut av en tjener når den mottar gitte meldinger
- prosedyrekall - aktivering av verktøy via prosedyrekall.

En høy grad av kontrollintegrasjon oppnås når verktøy er i stand til å effektivt tilby sin funksjonalitet til alle andre verktøy i omgivelsen, noe som krever høy grad av verktøymodularitet [48, s. 33-34].

I prosessintegrasjon er det tre dimensjoner som må imøtekommes for at verktøy effektivt kan integreres for å støtte en prosess i programvareutviklingen [48, s. 34]:

- *Process Step* - et arbeidssteg som gir et resultat

- *Process Event* - en betingelse som oppstår i utførelsen av et Process Step som kan medføre utførelsen av en assosiert aksjon

- *Process Constraint* - rammevilkår for prosessen

Verktøy har et sett antagelser om prosessen de vil bli brukt i, mer bestemt om et gitt *process step*. Hvis disse antagelsene stemmer overens mellom verktøy vil de kunne oppnå en god prosessintegrasjon. Utførelsen av et *Process Step* kan brytes ned i kjøring av ulike verktøy, hvor det kreves at disse verktøyene må kunne utføre sine oppgaver som bidrar til prosess-steget som helhet, og bidra til at andre verktøy kan gjøre det samme. Videre må verktøyene konsistent kunne generere og behandle *process Events*. Verktøyene må også gjøre like antagelser om *process Constraints* ved at de støtter og respekterer de samme rammevilkårene [48, s. 34-35].

Plattformintegrasjon fokuserer på rammeverktjenester. Fokuset mitt ligger på relasjoner mellom verktøy så i likhet med Thomas og Nejmeh velger jeg å ikke se nærmere på denne typen integrasjon.

Grundy et al. [14] presenterer enda en kategori i tillegg til de beskrevet over, nemlig en komponentbasert arkitektur. Denne baserer seg på en kombinasjon av tilnærmingene: melding, database og kanonisk representasjon. Med en komponentbasert arkitektur kan man oppnå effektiv data- og kontroll-integrasjon, samtidig som den i motsetning til de andre tilnærminger også tilbyr god støtte for integrering av tredjepartsverktøy. Gjenbruk er også godt støttet ved designe komponentene til å ha et minimum av eksterne avhengigheter.

Yang og Han [58] presenterer en klassifisering av verktøy-*interfacing* basert på brukerperspektivet, i motsetning til Meyers [30] inndeling som ser på integrasjon fra et utviklerståsted. De kritiserer en inndeling basert på integrasjonsmekanismer. De argumenterer for at dette ofte blir kunstig måte å kategorisere på hvor problemstillinger rundt brukergrensesnitt-, kontroll- og data-integrasjon ikke alltid lar seg plassere entydig. I stedet foreslår de tre typer grensesnitt som har som gjenspeiler hovedformålet med integrasjon, nemlig økt produktivitet. *Class 1 interfacing* går på å redusere brukergenerert forsinkelse, ved å fjerne kravet til brukerinteraksjon mellom verktøyaktivering og ved å sette tilbakemeldinger til brukeren i en kontekst. *Class 2 interfacing* går på å fjerne verktøygenerert forsinkelse ved teknikker som inkrementell dataprosessering og samtidig kjøring av ulike verktøy. *Class 3 interfacing* går på å effektivisere konstruksjon og vedlikehold av verktøy ved fokus på gjenbruk av komponenter som tillater stor grad av gjensidig synkronisering.

På bakgrunn av dette presenterer Yang og Han [58] en konseptuell arkitektur som består av en *front-end* som gir brukeren et uniformt grensesnitt til verktøyene. Dette bidrar med brukergrensesnittintegrasjon ved å gi brukeren et konsistent rammeverk for aktivering og bruk av verktøyene. I tillegg har man en *back-end* som består av en rekke ulike verktøy som tilbyr den nødvendige funksjonaliteten. Disse blir aktivert av front-end'en gjennom et generisk grensesnitt, som er ansvarlig for kontroll- og data-integrasjon.

5.4 Diskusjon

Både Scacchi [41] og Hemetsberger og Reinhardt [16] konkluderer med at asynkrone kommunikasjonsmidler er viktigst når det gjelder skaping av ny kunnskap fordi den tillater analysering, refleksjon og "re-experience". Dette gjør asynkrone kommunikasjonsmidler best egnet til skapelse og design av ny funksjonalitet. Synkrone kommunikasjonsmidler kan også benyttes til å skape kunnskap, men da mest for umiddelbar problemløsning, brainstorming og diskusjon som kan danne grunnlaget for et mer utførlig forslag blir sendt til en epostliste for videre diskusjon og refleksjon.

Prosessene i åpen kildekode-utvikling er i stor grad implisitt og uformell, og det er dermed kritisk å kunne fange opp kunnskap på en effektiv måte. Dette gjøres ved at kunnskap blir innskrevet i teknologien (verktøyene) brukt [26, s. 5]. Dette setter verktøy og interaksjonene mellom de i fokus i åpen kildekode-utvikling. Man kan sette frem påstanden om at man på denne måten får en viss grad av prosessintegrasjon ved at prosessen er innskrevet i verktøyene som blir brukt.

Når vi ser på bruken av verktøy finner man også forskjellig bruksmønster i tradisjonell utvikling kontra åpen kildekode-utvikling som kan se ut til å ha rot i de ulike prosessene som ligger til grunn. I tradisjonell utvikling benyttes en rekke verktøy som kommer under betegnelsen CASE [50, s. 632]. CASE omfatter alt fra enkeltverktøy til integrerte omgivelser som er laget for å dekke hele utviklingsprosessen.

Disse omgivelsene benytter ulike tilnærmelser. En rekke verktøy benytter objekt-orienterte rammeverk, med data lagret som filer eller i en database. Disse er ofte kombinert med versjonskontroll og konfigurasjonshåndtering for støtte for teamarbeid. Smalltalk er et godt eksempel her. Disse tilbyr ofte veldig spesialiserte og rike grensesnitt, men er ofte lite egnet for utvidelse og integrasjon med eksterne verktøy [14].

Verktøy som benytter databaser for datahåndtering og datapresentasjon

tilbyr god dataintegrasjon, med meldingsstøtte for kontrollintegrasjon. Man har også prosess-sentrerte verktøy, som tilbyr funksjonalitet for prosesskodifisering og utførelse. En del verktøy tar dette enda et hakk lengre og bruker en kanonisk representasjon som deles av alle verktøyene. Dette medfører generelt sett en god data-, kontroll- og presentasjons-integrasjon, men har også de største utfordringene når det kommer til samhandling med eksterne verktøy [14].

Et problem ved verktøyene med sterk integrasjon er at de tilbyr liten fleksibilitet og interoperabilitet med tredjepartsverktøy. Dette kan forklares med at de er basert på formelle modeller som setter begrensninger og rammer for hvordan verktøyene kan benyttes [50, s. 652]. Tradisjonell utvikling ser dermed ut til å videreføre rigiditeten og mangelen på fleksibilitet fra prosessmodellene som ligger til grunn over til verktøyene brukt i utvikling.

I åpen kildekode utvikling er det mest vanlige Unix-baserte verktøy (*toolkits*) som har en løsere integrasjon med en filsystembasert tilnærming. Dette medfører at det er veldig lett å legge til og integrere nye verktøy. Det gir derimot liten grad av data-, kontroll- og presentasjons-integrasjon. En større grad av integrasjon kan oppnås ved å benytte meldings-sending for å integrere filbaserte systemer. Dette kan brukes til å øke presentasjonsintegrasjonen, mens data- og kontroll-integrasjon vil fremdeles mangle i stor grad [14]. En rekke integrerte miljøer tilbyr dette.

Et relevant spørsmål er da om større integrasjon alltid vil føre til mindre fleksibilitet, eller om dette kan motvirkes ved å basere seg på andre grunnleggende prosesser for programvareutvikling. Sterkt integrerte verktøy basert på formelle metoder har store utfordringer med fleksibilitet. Noen av dette kan nok løses ved tekniske nyvinninger og bruk av for eksempel komponentbasert arkitektur, men å overvinne rigiditeten til prosessene som ligger til grunn vil fremdeles være et stort problem. Hvis man derimot forsøker å integrere verktøy i et system som baserer seg på åpen kildekode-utviklingens prosesser, kan man da oppnå større fleksibilitet i tråd med at disse prosessene er mye mer åpne og fleksible, eller som Bollinger et.al påpeker [3] i stor grad er fraværende, i forhold til tradisjonell utvikling?

Utvikling i åpen kildekode-prosjekter er en "Community building process" [41] som må ligge som et grunnlag for prosjektet og dets "software informalisms". I stedet for formelle metoder finner man sosiotekniske prosesser som omfatter det å skape sosiale bånd, uformelle sosiale kontrakter, og vilje til å bidra til konstruksjonen av og innskrivingen av kunnskap i artefakter. Ved å underbygge og vedlikeholde et *community* skaper man et grunnlag for utviklingen av prosjektet uten noe form for sentral styring. Et åpen kildekode-

prosjekt sitt nettsted er et sentral samlingspunkt som samler informasjon om utviklingen, deltagerne og bidragsyterne, og "Community communications" [41, s. 22-23]. På denne måten blir en rekke av verktøyene brukt i Open Source-utvikling integrert rundt ett nettsted eller en portal med et viktig unntak, nemlig verktøyene for koding. Disse er sjelden koblet direkte sammen med dette samlingspunktet, og det kan dermed være fordeler å hente med å skape denne koblingen.

I evalueringen av arkitekturer som kan være hensiktsmessig for integrasjons av verktøy for jobbing direkte med koden vil jeg legge Yang og Han [58] sin arkitektur som jeg har beskrevet tidligere til grunn. Denne arkitekturen virker hensiktsmessig for Open Source-utvikling av flere grunner. Tradisjonen fra Unix med små spesialiserte verktøy som kan kombineres står også sterkt i "Open Source-verdenen" og systemer som GNU/Linux og de ulike BSD-variantene er i stor grad bygd opp på denne måten. En rekke av disse verktøyene, da i stor grad asynkrone kommunikasjonsverktøy, er vital for "re-experience" i Open Source-utvikling. "Re-experience" er grunnlaget for læring i Open Source-utvikling så jeg finner det lite aktuelt å erstatte disse verktøyene med et nytt rammeverk med et nytt sett med funksjonalitet og krav som erstatter disse verktøyene i stedet for å jobbe med de. Bollinger et.al [3, s. 9] beskriver noe de kaller *the Open Source razor* på følgende måte: "Can you justify adding a new control, menthod, or metric to the process when open-source methods already work fine without it?". Derimot så er det mulighet for at de eksisterende verktøyene danner en samling som faller naturlig inn i back-end delen av Yang og Han sitt rammeverk. Man kan dermed oppnå integrasjon gjennom *front-end* og et generisk grensesnitt uten å miste all fleksibiliteten i de underliggende verktøyene. Eventuelle nye verktøy må kunne tilby den samme funksjonaliteten som muliggjør "re-experience", eventuelt forbedre mulighetene for dette gjennom å lagre og presentere ny eller eksisterende kommunikasjon på en bedre måte.

Kapittel 6

Metode

I dette kapittelet vil jeg beskrive min metodiske tilnærmelse. Jeg velger å se på tre forskningstradisjoner innen systemutvikling, positivistiske, fortolkende og design research. Ved å se på hva som karakteriserer disse vil jeg benytte dette som et grunnlag for mitt valg av metodisk tilnærmelse. Deretter vil jeg beskrive forskningsprosessen, og til slutt vil jeg diskutere implikasjonene av mitt valg av metodetilnærmelse og hvordan dette har påvirket resultatene.

6.1 Forskningstradisjoner innen systemutvikling

All forskning er grunnlagt i visse underliggende antagelser om hva som krever for å utføre forskning av høy kvalitet. Den positivistiske tilnærmelsen har sitt ontologiske grunnlag i empirisk forskning, og kjennetegnes ved at det ser på virkeligheten som objektiv og at denne kan beskrives ved å måle egenskaper som er uavhengig av den observerende forskeren og hans instrumenter. Det som testes er ofte teorier, i et forsøk på å øke den prediktive forståelsen av et fenomen. Forskning som kommer under positivisme er formelle fremlegg, kvantifiserbare målinger, hypotesetesting og å dra slutninger om et fenomen basert på et utvalg av en gitt populasjon [32]. Fordelen med den positivistiske tilnærmelsen er at den kan påvise eksakte sammenhenger mellom de aktuelle variablene man ser på, og i tillegg holde kompleksiteten nede ved å kontrollere antallet variabler man betrakter. Tradisjonelt har forskning på informasjonssystem fokusert på teknologien. Senere arbeid innen informasjonssystem har derimot i større og større grad innsett at forskningen bør utvides til å også dekke adferd

og organisatoriske faktorer [12, s. 900]. Når man i større grad omfatter flere omliggende faktorer vil antall variabler som må vurderes med øke betraktelig, og dermed gjøre en positivistisk tilnærmelse mindre fruktbar. Tilhengere av positivismen argumenterer med: "the empirical-analytical method is the only valid approach to improve human knowledge. What cannot be investigated using this approach, cannot be be investigated at all scientifically" [12, s. 900]. Dette krever at alle fenomen som observeres kan gjentas, noe som ikke er mulig for sosiale situasjoner. Dette gjør at man må se etter alternative tilnærminger når man vil ta med sosiale aspekter i forskningen.

En fortolkende (interpretive) tilnærmelse antar at kunnskap og virkeligheten i stor grad tas opp gjennom sosiale konstruksjoner som språk, bevissthet, delt forståelse, dokumenter, verktøy, og andre artefakter. Dets ontologiske grunnlag ligger altså i at det som studeres er sosialt konstruert. Informasjonssystem må derfor ses på gjennom hvilken "mening" brukerne legger i systemet. Forskning innen den fortolkende tilnærmelsen kan hjelpe informasjonssystemforskere til å forstå hvordan mennesker tenker og handler i sosiale og organisatoriske kontekster. Dets styrke ligger i å kunne produsere dyp innsikt i fenomen rundt informasjonssystem og informasjonssystem-utvikling [24]. En fortolkende tilnærming til informasjonssystem forsøker å oppnå en forståelse for *konteksten* til systemet, og *prosessene* hvor systemet påvirker og blir påvirket av denne konteksten [52]. For at leseren i størst mulig grad skal kunne sette seg inn i situasjonen må teksten presentere den sosiale og historiske konteksten rundt. Dette skiller seg fra det positivistiske ståstedet som forfekter at en hendelse som har skjedd kan repeteres, noe som ikke tar hensyn til at mennesker aktivt skaper og påvirker den sosiale og fysiske virkeligheten de lever i. Fortolkende forskning på sin side argumenter for at relasjonene mellom mennesker, organisasjoner og teknologi er i stadig forandring og forsøker dermed å forstå et bevegelig mål [24].

Design research (går også under navnet Design Science) har sine røtter i ingeniørvitenskap og det H.A. Simon karakteriserer som "Science of the Artificial" [43]. Det er et paradigme som i all hovedsak fokuserer på problemløsning. Formålet er å skape nyvinninger som definerer idéer, metoder, tekniske muligheter og produkt som kan danne grunnlaget for effektiv gjennomføring av analyse, design, implementasjon og bruk av informasjonssystemer. Design Research omfatter å skaper og evaluere produktet (artifact) som har som formål å løse identifiserte organisatoriske problemstillinger. Gjennom en prosess hvor man konstruerer og bearbeider nyskapende produkt får forskeren en forståelse for problemstillingen gjennom selve produktet og evaluering av gjennomførbarheten til tilnærmingen [17]. Det ontologiske grunnlaget for Design Research kan beskrives som evolusjonær og komplementær,

henholdsvis ved å representere ontologiske antagelser som endrer seg gjennom forskningsprosessen og ved å ta hensyn til både fenomenet og produktet. Disse antagelsene kan sies å være kontekstspesifikke alternative "world-states". Det epistemologiske grunnlaget kan karakteriseres som "knowing through making" og kan beskrives som reflekterende og fortolkende (*hermeneutic*) [13].

Den positivistiske tilnærmelsen er ikke spesielt egnet til å se på åpen kildekode-utvikling siden den i sterk grad er basert på sosiotekniske prosesser. Den fortolkende tilnærmelsen derimot er mer gunstig siden den ser på virkeligheten som sosiale konstruksjoner, og dette ligger mye nærmere de sosiotekniske prosesser i åpen kildekode-utvikling som omfatter sosiale bånd, normer og andre uformelle og implisitte sosiale konstruksjoner som ikke lar seg etterprøve. Jeg ønsker derimot å utvikle og teste ny funksjonalitet i et forsøk på å kunne tilføre noe til utviklingsprosessen. Derfor har valget falt på Design Research som metodisk tilnærming. Et annet aspekt som gjør Design Research velegnet for å undersøke fenomenet åpen kildekode-utvikling er at dette er felt som ikke har den samme tyngden av forskningsarbeid rundt seg som de mer tradisjonelle utviklingsparadigmene.

6.2 Forskningsprosessen

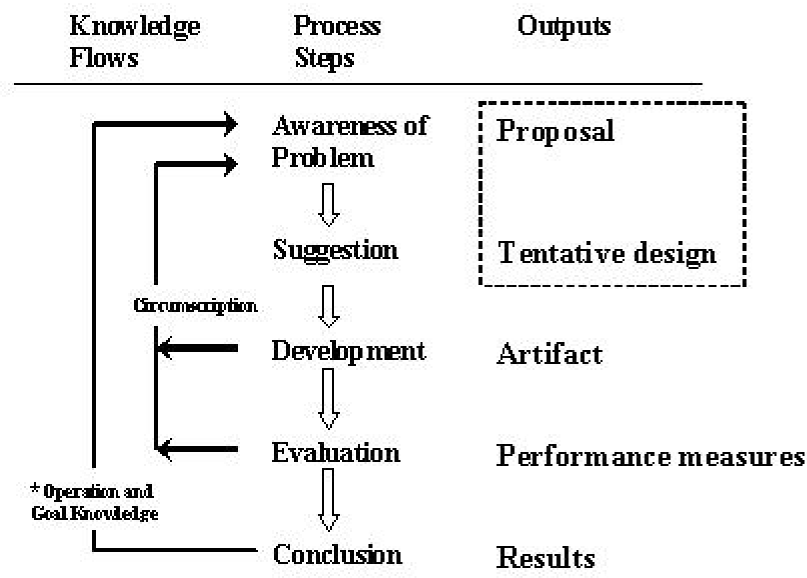
Metodelæren bak Design Research kan karakteriseres som en kreativ prosess som involverer å skape nye tanker og oppfinnsomhet rundt mulighetene som er til stede. S. Purao [13, s. 17] vektlegger i tillegg at Design Research kan sees på som en fortolkende (*hermeneutic*) prosess som eksplisitt betrakter de indre og ytre omgivelsene til produktet. Forskeren går inn i en fortolkende repeterende prosess med en idé om egenskapene produktet han skal skape vil inneha.

En typisk Design Research prosess kan i grove trekk gå frem på følgende måte (se figur 6.1[1]) [49].

Awareness of Problem: Man kan finne interessante problemstillinger fra en rekke kilder, for eksempel fra ny utvikling innen en disiplin eller inspirasjon fra andre fagfelt. Resultatet fra denne fasen er et formelt eller uformelt forslag for ny forskning.

Som jeg har sett på i foregående kapittel har åpen kildekode-metodikken vist seg å være en høyst levedyktig måte å utvikle programvare på som er i stand til å utvikle programvare av høy kvalitet. Denne måten å utvikle programvare

[1] Hentet fra www.isworld.com



Figur 6.1: The General Methodology of Design Research

på mange måter svært forskjellig fra tradisjonell utvikling, men har likevell en del fundamentale likhetstrekk. Jeg fant dette til å være et interessant område å se nærmere på i lys av verktøyintegrasjon som er en sterk trend i tradisjonell utvikling. I hvor stor grad kan sterkere integrasjon fungere i en åpen kildekode-omgivelse ble mitt hovedspørsmål.

Som grunnlag for mitt forslag har jeg i mitt litteraturstudie i hovedsak vært ute etter å finne hva som var fellestrekk for åpen kildekode-utvikling, ikke spesielle karakteristikker ved et unikt case. Dette involverer også flere nivåer: individ - gruppe - fellesskap (*community*), som tilsier at det ikke er hensiktsmessig å begrense seg til bare en case. Av mulige datakilder har jeg i hovedsak brukt artikler fra casestudier og deltakende observasjon. Jeg har brukt resultatene fra en rekke case-studier av spesifikke åpen kildekode-prosjekter for å forsøke å finne felles karakteristikker, samt annen kontekstskapende litteratur rundt emnet. Jeg vil trekke frem Scacchi [41], Mockus et. al [31], Hemetsberger og Reinhardt [16], og Lanzara, Morner [26] og Weber [54] som viktige kilder her. Jeg har i flere år deltatt i to aktive åpen kildekode-prosjekt[2]. Dette har latt meg studere på nært hold dynamikken i Open Source-prosjekt og hvordan de sosiale prosessene utspiller seg og hvilke problemstillinger og utfordringer som gjør seg gjeldende.

[2]Battle for Wesnoth (www.wesnoth.org) og Freeciv (www.freeciv.org)

I forslaget mitt identifiserer jeg de eksisterende infrastrukturene i utstrakt bruk i Open Source-utvikling som jeg mener er mest egnet for sterkere integrasjon med et utviklingsmiljø, samt beskriver en del elementer som jeg mener er hensiktsmessig i en slik integrasjon. Dette forslaget ble både grunnlag for valget av rammeverk og som grunnlag for designet.

Suggestion: I denne fasen utformer man et design på bakgrunn av forslaget i forrige fase. Dette er en høyst kreativ fase hvor man vurderer problemstillingen og ser om dette er interessant som et grunnlag for et foreløpig design hvor man ser for seg ny funksjonalitet eller nye måter å kombinere eksisterende funksjonalitet.

I designfasen går jeg i større detalj inn på funksjonaliteten jeg ønsker å implementere. Grensesnittet avhenger i stor grad av valget av Eclipse som rammeverk og jeg beskriver her hvilke rammebetingelser og muligheter dette valget medfører. Jeg benytter skjermbildeprototyper og scenario for i beskrive grensesnittet for funksjonaliteten som et programvaretillegg i Eclipse. Disse følger kronologisk, men jeg vil påpeke at det i stor grad har vært en iterativ prosess hvor arbeid med det ene har gitt nye tanker for de andre og ført til at jeg har revidert de andre delene i tråd med de nye funnene.

Development: I denne fasen blir det foreløpige designet implementert.

I designet har jeg beskrevet en rekke ulike elementer som potensielt kunne integreres i Eclipse. Det ville blitt for omfattende å implementere alle disse, så jeg var nødt til å begrense hvilke deler som skulle implementeres og testet. Jeg fant funksjonalitet for kontekstuell informasjon som det mest interessante og fokuserte på å implementere funksjonalitet rundt dette.

Implementasjonen ble gjennomført som en iterativ prosess hvor funksjonalitet ble implementert og testet i raske iterasjoner. Jeg hadde hyppig kontakt med Arne Mathisen hvor vi diskuterte funksjonalitet, problemstillinger rundt koden og testet programmet. Diskusjonene foregikk ofte ved å bruke selve programmet slik at vi fikk prøvd ut funksjonaliteten i utviklingssammenheng.

Evaluation: Etter implementeringen blir produktet evaluert i henhold til de kriteriene som implisitt eller eksplisitt ligger i forslaget. Avvik fra forventet resultat, både kvantitative og kvalitative blir her funnet og forsøkt forklart.

I tillegg til den hyppige uformelle testing mellom meg og Arne Mathisen som ble gjort underveis i utviklingen ble det utført brukbarhetstester på HCI-labben ved NTNU IDI. Disse testene avslørte en rekke problemstillinger ved både grensesnittet og den underliggende funksjonaliteten. En rekke grensesnittendringer ble gjort underveis for å rette på mangler som ble avdekket. Dette førte til at de største grensesnittrelaterte problemene ble

fjernet og fokuset kom mer på funksjonaliteten og hvor hensiktsmessig denne ble opplevd av testpersonen.

I tillegg til brukbarhetstesting av implementasjonen kontaktet jeg en rekke personer som er aktive deltagere i ett eller flere åpen kildekode-prosjekt med et sett spørsmål i et epost-intervju rundt åpen kildekode-utvikling og verktøyintegrasjon for å få tilbakemeldinger fra personer som var i målgruppen, samt for å få tilbakemelding på et høyere plan enn de mer konkrete resultatene fra testingen. Dette blir så brukt for å sette testresultatene inn i en større kontekst og for å validere eller invalidere antagelser gjort i designet og implementasjonen.

Conclusion: Siste fase i Design Research, hvor man konkluderer forskningsprosessen. Dette er typisk et resultat av at man har nådd et punkt hvor man finner resultatene til å være gode nok, selv om det fremdeles kan være avvik og problemstillinger til stede. Resultatene blir karakterisert som enten solide eller som mer usikre og med "loose ends", hvor sistnevnte ofte kan tjene som et grunnlag for videre forskning.

Min oppgave faller i kategorien med mer usikre resultater. Flere iterasjoner med implementasjon og testing, samt å få større grad av involvering fra personer fra hovedmålgruppen er nødvendig for å kunne oppnå solide resultater. Det er likevell muligheter for å bruke denne oppgaven som et utgangspunkt for videre forskning.

6.3 Diskusjon

Det epistemologiske grunnlaget for Design Research er som sagt refleksiv og fortolkende. Det er refleksivt fordi forskeren alternerer mellom å betrakte produktet og fenomenet. Forskeren genererer data og teorier gjennom en selektiv tilnærming av antagelser og analyse, som har som formål å redusere antall mulige "world-states" som må betraktes. Forskeren tar en fortolkende rolle i søken etter overensstemmelse mellom data og teori, og den virkelige verden. Designet av produktet og tolkningen av fenomenet blir dermed utført samtidig. Det fortolkende perspektivet involverer at forskeren har et forhold til både idéen om produktet og tolkningen av fenomenet som gir innsikt som åpner for videre tolkning. En forståelse av fenomenet og produktet blir dermed skapt samtidig i en fleksibel prosess [13].

For å forstå fenomenet har jeg forsøkt å sette åpen kildekode-utvikling inn i den nødvendige konteksten. I kapittel 3 beskriver jeg hvordan åpen kildekode oppstod, samt å vise dets tilknytning til hvordan programvareindustrien

ellers utviklet seg og dermed også proprietær utvikling. På samme måten forsøker jeg å plassere åpen kildekode-utvikling i lys av tradisjonell utvikling både for å forankre det til resten av bransjen, samt ha et grunnlag for sammenligning. En stor andel av utviklerne i åpen kildekode-prosjekt er også involvert i proprietær utvikling, samt at proprietær utvikling i større grad begynner å bruke åpen kildekode-biblioteker så å se disse i sammenheng er etter min mening viktig. Videre i kapittel 4 og 5 går jeg grundigere inn på åpen kildekode-utvikling satt opp mot proprietær utvikling i forhold til henholdsvis programvareutvikling og kunnskap for å se på hva som er likhetene og forskjellene, samt se på hvilken påvirkning som har foregått mellom de to. Kapittel 6 går spesifikt inn på verktøy og verktøyintegrasjon.

Undersøkelsen av fenomenet og arbeidet med produktet foregikk i iterasjoner. En foreløpig forståelse av fenomenet lå til grunn for første forslag som så ble grunnlag for et design, og til slutt en implementasjon. I hver fase gikk jeg flere ganger tilbake etter å ha møtt på problemstillinger som gjorde at jeg innså at jeg måtte få en større forståelse for fenomenet før jeg kunne fortsette. Dette medførte videre arbeid med bakgrunnskapitlene som igjen førte til revisjoner av forslag, design og implementasjon i hyppige iterasjoner.

Iterasjonene i Design Research blir ledet og avgrenset av interne og eksterne rammebetingelser. En iterasjon er en prosess som består av en rekke avgjørelser som blir bestemt av de nåværende representasjonene, forståelsen av fenomenet, og forskerens egne verdier. Purao understreker at det finnes ingen absolutte sannheter i design, noe som medfører at muligheten for at et annet design enn de man har funnet frem til kan tilfredsstille rammebetingelsene bedre alltid er til stede. Det er derfor kritisk at man i den fortolkende prosessen tar hensyn til det som finnes av eksisterende materiale og praksis og forsøker å ta høyde for dette i vurderingen av gjennomførbarheten til arbeidet [13].

Jeg har derfor i tillegg til å jobbe med å forstå fenomenet gjennom det arbeidet jeg har gjort i bakgrunnskapitlene samt arbeidet med design og implementasjon, gjennomført en undersøkelse blant åpen kildekode-utviklere for å potensielt kunne identifisere ytterligere faktorer som vil kunne ha innvirkning på gjennomførbarheten til arbeidet ut over det som har blitt funnet tidligere.

Resultatet av Design Research er mer enn bare produktet i seg selv. Produktet kan selv om det er det mest synlige resultatet ikke nødvendigvis være det viktigste. Både operasjonelle prinsipper illustrert gjennom produktet og forståelse for hvordan produktet støtter fenomenet er viktigere [13].

Jeg har forsøkt å evaluere produktet med grunnlag i fenomenet for å se

hvilken innsikt som kan hentes ut om verktøyintegrasjon i åpen kildekode-utvikling. Det har vist seg å være en rekke elementer som må tilfredsstilles for at dette skal kunne gjøres produktivt. Alt fra hvilken type integrasjon som er hensiktsmessig, til hvilke verktøy som er viktig å fokusere på, og hvilke egenskaper som man er nødt til å ta vare på for at fleksibiliteten som er et nøkkelmoment for åpen kildekode-utviklere ikke skal gå tapt.

Til slutt vil jeg se på noen av de problematiske aspektene ved Design Research som denne tilnærmingen har blitt kritisert for, og hvordan dette har reflektert seg i mitt arbeid. En relevant problemstilling er gyldigheten til evalueringen. Man kan ikke se bort ifra at det er store forskjeller mellom de prototypene man vanligvis tester og fullverdige IT-systemer som er satt i drift. Man er derfor avhengig av å støtte seg på en del antagelser som det ikke er gitt stemmer for reelle systemer. Denne problemstillingen er også relevant for mitt arbeid. Min prototype har blitt testet på enkeltpersoner som ikke er deltagere i et åpen kildekode-prosjekt, så de resultatene som kom fra brukbarhetstestene er i større grad nyttig for grensesnittevaluering enn for å evaluere nytten til funksjonaliteten i en reell setting. Testene bruker faktiske diskusjoner som er fra utviklingen for å gjøre testene mer reell, men man kan ikke se bort ifra at det er et stort gap mellom disse testene og et system satt i drift i et åpen kildekode-prosjekt.

Utvikling av programvare er et komplekst og tidkrevende arbeid, noe som også gjør Design Research til en krevende disiplin. Jeg hadde idéelt sett ønsket å gjennomføre en ny mer omfattende iterasjon hvor jeg kunne testet en ny implementasjon basert på de siste testresultatene i et faktisk åpen kildekode-prosjekt, men dette viste seg å bli for omfattende til å kunne inngå i denne oppgaven av flere grunner. For å være attraktivt for bruk i faktisk utvikling ville jeg være nødt til å implementere alle elementene jeg la frem i designet, samt ta høyde for alle problemstillingene som kom frem i undersøkelsen blant åpen kildekode-utviklerne i en ny implementasjon. Videre ville det være lite realistisk å forvente at deltagerne i et eksisterende åpen kildekode-prosjekt ville være villig til å bytte ut de verktøyene de var vant til og fortrolig med for å benytte en sannsynligvis ukjent omgivelse og teste et system som ikke enda var stabilt og modent.

Kapittel 7

Forslag til verktøyintegrasjon

Som jeg beskrev i kapittel 3 kan man se en rekke likheter med åpen kildekode-utvikling og nyere prosessmodeller innen tradisjonell utvikling som *Extreme Programming*. Dette danner dermed grunnlag for at det kan være hensiktsmessig å se på verktøy brukt i tradisjonell utvikling som interessant også for åpen kildekode-utvikling. Det er også tilfellet at de i stor grad bruker de samme typer verktøy for mange av oppgavene i programvareutvikling. Åpen kildekode-utviklere sin måte å arbeide på blir i større grad definert av hvordan de velger å samarbeide enn hva de faktisk gjør. Det er ingen overordnet styring og individuelle bidrag er grunnlaget for prosjektets kollektive fremdrift. I kapittel 5 ser jeg nærmere på verktøyintegrasjon og det som der utpekte seg var Yang og Han [58] arkitektur fordi denne kunne bygge videre på toppen av eksisterende verktøy. Hvorfor er dette så viktig? Jeg fremsetter at det er hvordan kunnskap blir delt og reflektert rundt, samt måten dette ligger innbakt i verktøyene og de sosiale konstruksjonene i åpen kildekode-utvikling som et av de viktigste aspektene ved denne måten å utvikle programvare på. Å ivareta dette og muligheten for å forsterke denne effekten blir dermed viktig.

Åpen kildekode-prosjekter består av en heterogen gruppe mennesker spredt over hele verden. Målgruppa for dette arbeidet er utviklere i i disse prosjektene som i all hovedsak jobber med kildekode, og som jobber fysisk separat fra hverandre og må ty til elektroniske kommunikasjonsverktøy for kontakt seg i mellom. I denne typen utvikling er medlemmene ofte flyktige. Eksisterende bidragsytere forsvinner og nye kommer til hele tiden, og *utviklingsteamet* blir dermed veldig dynamisk og i stadig endring. Krav er også i stor grad knyttet til enkeltpersoner og hva de ønsker å implementere så det aspekter blir også i stadig endring.

En mer nøyaktig definisjon av et “Open Source Community” vil være hensiktsmessig før en går videre. En vid måte å definere det på er å inkludere alle som bruker åpen kildekode-programvare, men termen er oftest brukt om de personene som på en eller annen måte bidrar til et åpen kildekode-prosjekt. Disse kan så deles inn i kjernebidragsytere og mer perifere bidragsytere [55].

Kjernen består av de deltakerne som har jobber direkte med kode og annen data som prosjektet består av. Periferibidragsyterne består i stor grad av brukere av programvaren som rapporterer feil, foreslår forbedringer og bidrar med fikser.

Deltakerne kan videre inndeles på følgende måte:

- Prosjektledere som er ansvarlig for prosjektet (kjernebidragsytere). Disse kontrollerer hvilken retning prosjektet skal ta og hvilke endringer som blir godtatt fra andre bidragsytere.
- Frivillige bidragsytere (kjerne og periferi) som bidrar med kode eller annen data, disse inkluderer:
  - Seniormedlemmer som har fått ansvarsområder og rettigheter
  - Perifere bidragsytere som bidrar med kodefikser og annet data
  - Tilfeldige bidragsytere
  - Bidragsytere som vedlikeholder ulike aspekter ved prosjektet
- Vanlige brukere som utfører testing, identifiserer feil, registrerer feilrapporter, etc (perifer)
- Brukere som deltar i nyhetsgrupper, forum og lignende, men som ikke bidrar med kode eller data

De aller fleste aktive åpen kildekode-prosjekter har en eller flere nettbaserte infrastrukturer som er samlingspunkt (hubs) for utviklingen. Disse er ikke av absolutt nødvendighet, men de bidrar til at åpen kildekode-miljø, prosesser og praksis kan utfolde seg effektivt [41]. Jeg vil trekke frem tre elementer her som er de jeg mener er de viktigste i denne sammenhengen. Første er nettsider som fungerer som en informasjonsinfrastruktur for publisering og deling av beskrivelser av programvaren som fungerer som prosjektets *organizational memory* ved å lagre, håndtere og presentere informasjon globalt og fritt tilgjengelig [41]. Det andre er systemer for håndtering av kompleksitet. Viktige verktøy her er versjonskontrollsystem for kode og feilhåndteringsystemer som skal håndtere rapportering og organisering

av feil og ønsket ny funksjonalitet i programvaren. Siste er *community communications* som omfatter en rekke kommunikasjonsverktøy som blir brukt aktivt for å fasilitere effektiv kommunikasjon og deling av kunnskap mellom deltagerne. De viktigste her ser ut til å være asynkron kommunikasjon av typen epostlister og nettbaserte forum, og synkron kommunikasjon gjennom IRC og lynmeldinger.

Det som er verdt å merke seg her er at selve kodingen og dermed utviklingsomgivelsene som blir brukt i liten grad er integrert i disse og er dermed ikke del av fokuspunktet for et prosjekt. Hovedutfordringen blir dermed å utforske muligheten for integrasjon mellom disse, og hvordan dette er hensiktsmessig å utføre. For å kunne oppnå dette er det en rekke faktorer som må tilfredsstilles og som vil danne grunnlaget for kravene til forslaget.

Jeg har identifisert tre eksisterende infrastrukturer i åpen kildekode-utvikling som jeg mener er de viktigste. Integrasjon med disse vil arte seg ulikt. *community communications* er det jeg tenker kan ha størst effekt å direkte integrere, samt kobling mellom eksterne verktøy for reduksjon av kompleksitet slik at de kan styres fra et utviklingsmiljø. Informasjonsinfrastruktur som nettsider er mindre relevant å ha en direkte kobling til.

Effektivisering er som sagt et opplagt mål ved integrering ved å automatisere oppgaver, samle funksjonalitet, etc. Jeg vil derimot sette økt grad av deling av kunnskap og læring som det viktigste målet for mitt integrasjonsforslag. Dette vil i hovedsak gå ut på å sette synkron kommunikasjon inn i en kontekst som så kan lagres og gjenfinnes i denne konteksten. På denne måten fanger man bedre opp kunnskap hos enkeltpersoner og knytter det til relevante objekter som bidrar til læring for de som studerer de. Videre er økt bruk av meta-informasjon rundt objektene i et utviklingsmiljø også med på å inskribere skjult kunnskap og gjøre den eksplisitt og tilgjengelig for refleksjon. Kodeintensiv jobbing foregår ofte i samråd med problemløsning på synkrone kommunikasjonsverktøy. Disse diskusjonene er umiddelbare og får ikke noe kontekst rundt seg bortsett fra selve diskusjonen. Det er dermed vanskelig selv med loggførte samtaler og skjønne konteksten rundt det og kunne reflektere over den. Asynkrone kommunikasjonsverktøy på sin side bidrar allerede i stor grad til "re-experience". Jeg vil derfor se på om i hovedsak synkrone verktøy satt i en kontekst og lagret i forbindelse med et objekt i en utviklingsomgivelse kan gi kommunikasjonen større verdi og muligens være mer velegnet for refleksjon og læring.

7.1 *community communications*

Kommunikasjonsverktøyene som det her er snakk om er alle basert på åpne kommunikasjonsprotokoller og er tilgjengelig via en rekke ulike program. Et viktig aspekt er at all kommunikasjon i stor grad er tilgjengelig utenfor sitt eget medium også. Dette materialiserer seg i blant annet logger fra IRC-kanaler og arkiv over epostlister som gjøres tilgjengelig på nett. Denne globale tilgjengeligheten av kommunikasjon som er vedvarende, søkbar og åpen er et viktig aspekt ved åpen kildekode-utvikling og må tas hensyn til ved integrasjon i et utviklingsmiljø.

Ved å integrere kommunikasjon i et utviklingsmiljø vil man forsøke å bygge en bro mellom koden og kommunikasjon om koden. For effektiv kommunikasjon er det kritisk at deltagerne har et klart bilde av hva som diskuterer. Ved å sette diskusjonen sterkere inn i en kontekst vil man kunne oppnå større effektivitet og mindre misforståelser om det som diskuteres. Det er derfor ønskelig at alle får en god forståelse for konteksten rundt kommunikasjon så raskt som mulig. Kontekst kan skapes på mange måter, f.eks gjennom diskusjon og håndgesturer, lyd, bilde og tekst. I åpen kildekode-utvikling har man som oftest et begrenset utvalg av disse hjelpemidlene tilgjengelig og det er derfor ønskelig å maksimere nytten av de man har tilgjengelig.

Både synkron og asynkron kommunikasjon bør støttes i en integrert kommunikasjonsløsning. Synkron kommunikasjon er hensiktsmessig for direkte diskusjon om implementasjonsdetaljer og i problemsøk i koden, mens asynkron er viktig for refleksjon og *re-experience*. Kommunikasjonsløsninger som integreres bør dermed støtte begge to og ikke nødvendigvis som adskilte verktøy. Ved å benytte en løsning som støtter begge kan man få en direkte kommunikasjon mellom de deltagerne som er logget på for øyeblikket og i tillegg kunne sende beskjeder og/eller annen informasjon til deltagere som ikke er pålogget, men som vil få denne informasjonen når de logger seg på. Dette vil kunne fasilitere begge typer og virke naturlig for brukeren. I tillegg bør all kommunikasjon som er av interesse for alle i prosjektet arkiveres og gjøres tilgjengelig på nettet for senere lesning og refleksjon. Denne informasjonen vil det også kunne være hensiktsmessig å knytte til bestemte objekter i prosjektet slik at man i utviklingsmiljøet skal kunne få opp relevante diskusjoner om det man ser på. For å oppnå dette kan man gi muligheten for å definere en kontekst for en diskusjon. Denne konteksten kan omfatte informasjon om hvem som er aktiv på de aktuelle delene av koden, hvilke objekter og ressurser som er involvert og tidsrom for aktivitet.

7.2 Kontekstuell informasjon

Ved integrering av kommunikasjonsverktøy i et utviklingsmiljø vil man kunne bidra til en sterkere kontekstuell forankring rundt jobbing med koden ved å tilby metainformasjon rundt både de involverte personene og filene og andre ressurser prosjektet består av, samt større kobling mellom disse. Med denne metainformasjonen har man en økt mulighet til å skape større bevissthet ("awareness") for deltagerne. Bevissthet involverer å vite hvem som er tilgjengelig, hva som skjer og hvem som er i kontakt med hvem. Mennesker som jobber fysisk sammen har en rekke måter å bli bevisst på andre og måter for å tilrettelegge sin egen arbeidsdag for at andre igjen skal bli bevisst. Innen systemutvikling var det tidlig systemer for overvåkning som var mest utstrakt, hvor bilder og lyd ble brukt til å vise hva man jobbet med og hvem som var tilgjengelig. Senere systemer har skiftet fokus over på å legge til rette for at folk skal kunne gjøre tilgjengelig informasjon av hva de jobber med, og i tillegg gi statusinformasjon om delte ressurser og fremgang i oppgaver det samarbeides om [38, s. 124].

For økt bevissthet kan man benytte både sanntidsinformasjon og persistent informasjon. Sanntidsinformasjon er informasjon som bare er gyldig der og da, mens persistent informasjon er informasjon som er gyldig over en lengre tidsperiode.

Sanntidsinformasjon kan brukes for både personer og filer og andre dataressurser. Dette går i stor grad på statusinformasjon om de ulike objektene man har i utviklingsmiljøet. For personer kan interessant informasjon være:

- Informasjon om vedkommende er logget på eller ikke
- Informasjon om hva vedkommende jobber med og eventuelt om vedkommende er tilgjengelig for kommunikasjon

For filer og andre ressurser kan interessant informasjon være:

- Informasjon om status for objektet. Dette kan være markering av objekter som har blitt endret av andre i forhold til den versjonen du har sjekket ut lokalt
- Vise hvem som jobber med en gitt fil

Sanntidsinformasjon er viktig for å vite hva hvem gjør på ett gitt tidspunkt. Dette gir oversikt over utvikling i et prosjekt og kan hjelpe å forhindre

overlappende arbeid, kodekonflikter og andre relaterte problemer. Denne informasjonen kan med fordel presenteres med enn viss redundans ved at man i tillegg til å kunne se status på en bestemt person som kan jobbe på gitte objekter, også skal kunne få samme informasjon ved å se på objektene det blir jobbet på. Hvis sanntidsinformasjonen også innbefatter "shared workspace" (som innebærer at man i sanntid ser hva andre jobber med) kan man vise objekter som blir oppdatert også før de blir sjekka inn i revisjonssystemet. Vanligvis får man ikke innblikk i hva som er i endring før endringen er sjekket inn i revisjonssystemet, med mindre det har vært en diskusjon i forkant om implementeringen.

I tillegg til sanntidsinformasjon om objekter er det også en stor mulighet i å legge til kontekstuell varig informasjon. Kodefiler og andre ressurser i et prosjekt kan merkes med forskjellig informasjon som kan være av interesse for andre som deltar.

For personer kan interessant informasjon her være:

- kontaktinformasjon for vedkommende
- Informasjon om ansvarsområder i prosjektet

Ansvarsområder er ofte uklar. Utviklerfellesskap er dynamiske, noen forsvinner, nye kommer til så å ha en oppdatert oversikt over hvem som har ansvar for hva kan være fordelaktig. Dette innebærer også at det bør være lettvint å sette denne informasjon. Dette kan også benyttes til å sette at en kodefil ikke er vedlikeholdt av noen for øyeblikket og trenger noen til å trå til. Dette kan nye utviklere benytte for å se hvor det trengs at man hjelper til.

For filer kan man blant annet ha informasjon om:

- Statusinformasjon om en gitt fil eller ressurs
- Metainformasjon som andre har satt på filer for å markere spesielle omstendigheter, som for eksempel at denne filen vil bli fjernet etter at de har fullført *refactoring* av denne delen av koden
- Kommunikasjon knyttet til en bestemt ressurs
- Informasjon knyttet til en fil lagt inn av en utvikler

I tillegg til å lagre kommunikasjon mellom brukere kan det også være interessant å lagre informasjon som blir skrevet inn av enkeltutviklere For

eksempel kan en utvikler ønske å legge inn en begrunnelse for hvorfor han valgte en gitt implementasjon som kan hjelpe andre å forstå hvorfor koden er som den er. Kode som i utgangspunktet ser lite effektiv ut eller tilsynelatende har andre problemer kan ha en rekke årsaker som ikke umiddelbart er opplagt når man bare ser på koden uten å vite hvorfor det ble implementert på denne måten. Dette er informasjon som blir for omfattende å legge i selve kildekodefilen, og vil kunne egne seg som metainformasjon sammen med diskusjoner.

På denne måten kan man kodifisere mer skjult kunnskap som utviklerne sitter inne med i selve utviklingsomgivelsen. I tillegg kan mal få samlet eksisterende informasjon som kan være tidkrevende og tungvint å finne frem til, spesielt for nye deltagere. Å ha koblinger med metainformasjon til filene i revisjonssystemet vil kunne være et naturlig utgangspunkt for å nøste seg videre i tilgjengelig informasjon når man er ute etter informasjon som er spesifikk for en gitt del av koden.

7.3 Reduksjon av kompleksitet

For reduksjon av kompleksitet er feilhåndteringssystemer og versjonskontroll to av de viktigste verktøyene i bruk. Felles for disse er at det er liten grad av integrasjon mellom kommunikasjonsverktøyene og utviklingsmiljøet brukt av deltagerne.

I tillegg til den kontekstuelle informasjonen rundt “awareness” er det mye metainformasjon som også kan hentes fra infrastruktur som allerede benyttes. Dette er informasjon som kan brukes til å forstå hvilken posisjon en person har i prosjektet og til å forstå sammenhenger og konteksten til en fil eller ressurs. Her finnes det allerede en stor mengde informasjon tilgjengelig i blant annet revisjonssystem og feilhåndterings-systemer som kan hentes ut og benyttes i et utviklingsmiljø. I tillegg til å kunne være effektivt ved å være mer umiddelbart tilgjengelig kan dette også bidra til at informasjonen holdes mer oppdatert. Med eksterne systemer kan man fort ende opp med å ikke holde ting oppdatert og det vil da fort hope seg opp og bli en stor arbeidsoppgave å rydde opp i, som for eksempel med feilrapporter. Hvis man i et utviklingsmiljø får opp de rapportene som er tildelt en selv så kan en bli mer motivert til å bli “kvitt de”, det vil si fikse feilene.

Det er mye informasjon som kan være interessant å hente ut. Dette omfatter blant annet hvem som har skrevet hvilken del av koden, uthenting av revisjoner av koden for å se endringer som har blitt gjort og feilrapporter

og hvilke deler av koden disse er knyttet til. Dette er informasjon som er i aktiv bruk, men ofte gjennom eksterne verktøy. Dette er informasjon som er nyttig å kunne få tilgang til direkte i utviklingsomgivelsen.

7.4 Nettbasert informasjon

Nettbasert informasjon er den infrastrukturen jeg ser minst nytte av en mer direkte integrasjon med et utviklingsrammeverk. Dette er informasjon som er tilpasset og egnet for å aksessere i en nettleser og ikke nødvendigvis ha direkte tilgjengelig i en utviklingsomgivelse. Så lenge man ikke har en interaksjon ut over det å bare innhente og vise informasjon så ser jeg ikke den store nytten akkurat her. Det kan være mer nyttig andre veien med generering av dokumentasjon som så legges ut på nettet, men dette krever ikke nødvendigvis noe tilleggsverktøy i et utviklingsrammeverk av den typen funksjonalitet jeg ser på. Jeg velger å ikke se nærmere på mulig integrasjon av denne typen informasjon siden jeg ser betraktelig mindre nytteverdi her enn for de andre infrastrukturene jeg har identifisert.

Kapittel 8

Valg av utviklingsrammeverk

Med kravet om å vedlikeholde fleksibiliteten i de eksisterende verktøyene som allerede er i utstrakt bruk i åpen kildekode-utvikling, setter man visse føringer for hvilke løsninger som kan benyttes for å oppnå større integrasjon. Jeg har valgt å ta utgangspunkt i Yang og Han [58] arkitektur fordi denne i stor grad tilfredsstiller de kravene som er fremsatt. Det man da står igjen med som mest hensiktsmessig er grensesnitt-integrasjon, og til en viss grad muligheter for kontroll- og data-integrasjon gjennom et generisk grensesnitt mellom front-end og back-end. Systemer som baseres seg på en kanonisk data-integrasjon vil dermed ikke være spesielt egnet på grunn av de typiske problemene med fleksibilitet og samhandling med tredjepartsverktøy.

Videre er det en rekke andre krav som det vil være naturlig å sette frem. Rammeverket bør være distribuert under en åpen kildekode-lisens for å tillate den nødvendige åpenheten for deltagelse og muligheten for tilpasning. Dette er også viktig for distribusjon av rammeverket, både for at det skal kunne inkluderes på ulike plattformer og for at det skal kunne oversettes (*portes* til nye systemer. I åpen kildekode-utvikling benyttes det en rekke programmeringsspråk, og det vil være naturlig å sette et krav og at rammeverket bør være språk-agnostisk. Det er også ønskelig at det er i aktiv utvikling og har et levende åpen kildekode-fellesskap rundt seg. Rammeverket må også ha koding som fokus, og dermed blir en rekke systemer av typen portaler (sourceforge, gna, savannah) mindre aktuelle. Rammeverket må heller ha muligheter for kommunikasjon/integrasjon med disse portalene.

Det er som sagt utbredt i "Unix-verdenen" at det benyttes en rekke uavhengige verktøy for å få utført spesifikke oppgaver, og som lett kan samhandle med hverandre. Det jeg vil se på her er de rammeverkene som kan fungere som en paraply over disse verktøyene og tilby et integrert

og konsistent grensesnitt for effektivt arbeid i implementasjons- og kodefasen av utviklingsprosessen. Rammeverket bør også kunne utvides for å oppnå større integrasjon mellom verktøyene som kommer inn under "software informalisms" og rammeverket brukt for kode og testing. Det som da faller seg mest naturlig her er å se på *Integrated Development Environment* (IDE) som er en samling verktøy som skal assistere programutviklere med å utvikle programvare. Et IDE består vanligvis av en kildekode-editor, en kompilator/interpreter, debugger og kompilerings-automatiseringsverktøy. I tillegg kan den ha verktøy for versjonskontroll og grensesnitt-bygging. Disse verktøyene blir presentert i et felles grensesnitt hvor verktøyene kan virke sammen.

8.1 Eclipse

Valget av rammeverk falt på Eclipse [11]. Eclipse er velegnet til presentasjonsintegrasjon og samtidig vedlikeholde den nødvendige fleksibiliteten og samhandlingsevnen. Det er programvaretilleggs-arkitekturen (*plugins* til Eclipse som vil bli benyttet i denne oppgaven for å prototype og teste integrasjonen av verktøy i Eclipse.

Eclipse er et programvarerammeverk som er laget for å levere en utvidbar utviklingsplattform og applikasjonsrammeverk for bygging av programvare. Eclipse er lisensiert som åpen kildekode under Common Public License (CPL) og er plattformuavhengig. Eclipse ble originalt utviklet av IBM, men i 2001 ble Eclipse Foundation opprettet for å ta over utviklingen og styringen med prosjektet. Eclipse Foundation er en ikkeprofitt-konsortium som består av en rekke aktører fra programvareindustrien.

Eclipse har til nå i hovedsak blitt benyttet til å utvikle IDE'er, hvor Java Development Toolkit (JDT) med tilhørende kompilator kommer som en del av Eclipse, og som også blir brukt til å utvikle selve Eclipse. Eclipse består av 3 prosjekter: Eclipse Project, Eclipse Tools Project og Eclipse Technology Project. Eclipse Project består igjen av tre underprosjekter: Platform, JDT - Java Development Tools og PDE - Plug-in development environment.

8.1.1 Programvaretilleggsarkitektur

Et programvaretillegg (*plugin*) er den minste enhet som kan utvikle og levere Eclipse Platform funksjonalitet som en separat entitet. Verktøy blir levert som et eller flere programvaretillegg. Med unntak av "Platform Runtime" er

all funksjonalitet i Eclipse gitt ved disse programvaretilleggene. Disse er kodet i programmeringsspråket Java, og blir distribuert som en Java Archive (JAR) fil sammen med andre nødvendige filer for det gitte tillegget. Konfigurasjonen til et tillegg er gitt i to filer. En manifestfil "manifest.mf" som inneholder essensiell informasjon om tillegget, som inkluderer blant annet navn, versjon og avhengigheter til andre programvaretillegg. I tillegg kan man benytte "plugin.xml" som beskriver hvordan programvaretillegget er sammenkoblet med andre programvaretillegg. Dette gjøres ved å deklarere *extension points* og et gitt antall *extensions* til *extension points* i andre tillegg. Et *extension point* er altså et gitt grensesnitt til et programvaretillegg som kan bli utvidet av andre tillegg [19].

8.1.2 Platform Runtime

Platform Runtime er en liten kjerne som har som ansvar å initiere og starte opp Eclipse og blir kjørt av en enkelt Java Virtual Machine-instans. All annen funksjonalitet er gitt ved programvaretillegg. Ved oppstart finner Platform Runtime alle tilgjengelige programvaretillegg, leser de tilhørende manifest-filene og bygger et register over alle programvaretilleggene som er tilgjengelig. Dette registerer er så tilgjengelig via Platform API'en [19].

8.1.3 Workspaces

Verktøyene i Eclipse arbeider mot filer på filsystemet via brukerens *workspace* som inneholder ett eller flere prosjekt som korresponderer til en tilhørende katalog på brukerens filsystem. Alle filer i et *workspace* er direkte tilgjengelig for verktøy i det underliggende operativsystemet. Prosjekter, filer og foldere er tilgjengelig som *resources* for verktøy (programvaretillegg) i Eclipse via *The Platform API* [19].

8.1.4 Workbench og UI Toolkits

Eclipse Platform tilbyr et grensesnitt bygd opp rundt en *workbench* som tilbyr en overordnet struktur, og som er implementert gjennom to *toolkits*: SWT og Jface. SWT er et sett av grafiske brukergrensesnitt-elementer (*widget set*) og et grafikk-bibliotek som er integrert med de ulike plattformene og som støtter dets grafiske grensesnitt, men med en plattformuavhengig API. JFace er et UI toolkit som er implementert ved hjelp av SWT for å tilby høynivå grafikk-komponenter som dialog-, innstillings- og "wizard"-rammeverk [19].

Mens SWT og JFace er generelle UI toolkits så tilbyr Eclipse workbench det spesifikke grensesnittet i Eclipse-plattformen sammen med strukturer for verktøyinteraksjon med brukeren. Grensesnittet i Eclipse-plattformen er bygd opp av *editors*, *views* og *perspectives*. *Editors* tilbyr funksjonalitet for å åpne, editere og lagre objekter. *Views* tilbyr kontekstuell informasjon om objekter som brukeren arbeider på. En *workbench* kan tilby flere ulike *perspectives* hvor bare ett av de kan være synlig til enhver tid. Et *perspective* inneholder sine egne *Editors* og *views*. Eclipse-plattformen tilbyr standard perspektiver for ressursnavigasjon, online hjelpfunksjonalitet og team-funksjonalitet. Ytterligere perspektiver er tilgjengelig i andre programvaretillegg [19].

Verktøy integrerer inn i dette *Editors-Views-Perspectives*-paradigmet gjennom *extension points* for hver av disse grensesnittkomponentene. I tillegg kan verktøy utvide eller endre eksisterende komponenter. Integrasjon av verktøy kan foregå på flere nivåer:

- Verktøy skrevet i Java som benytter plattform API'en kan oppnå full integrasjon
- En rekke plattformspesifikke verktøy kan integreres som OLE og ActiveX under MS Windows
- Verktøy som bruker Java AWT eller Java Swing kan åpnes i separate vinduer som gir løs grensesnitt-integrasjon, men kan ha tett underliggende integrasjon
- Eksterne verktøy kan startes fra Eclipse og kjøre som en egen instans

8.1.5 The Eclipse Communication Framework (ECF)

The Eclipse Communication Framework er et prosjekt som har som mål å tilby API'er som skal forenkle bygging av distribuert programvare som skal tilby klient-tjener og peer-to-peer meldings- og kommunikasjons-tjenester. ECF tilbyr en rekke komponenter for dette formålet, blant annet for tilgjengelighet, lynmeldinger og fildeling. ECF er også lagt til rette for å utvides for å benytte en rekke kommunikasjonsprotokoller som Jabber og IRC [10]. I skrivende stund er ECF under tung utvikling og er vurdert til å være et for ustabilt mål til å benyttes, men er absolutt relevant for det denne oppgaven ser på.

8.2 Andre aktuelle rammeverk

Eclipse sin oppbygning som en modulær, fleksibel og utbyggbar plattform gjør den velegnet til å utvides med ny funksjonalitet. Disse egenskapene er dog ikke unik for Eclipse og det finnes en rekke andre rammeverk som også tilbyr tilsvarende løsninger, av disse vil jeg trekke frem Emacs [47] og netBeans [5].

8.2.1 Emacs

Emacs beskrives på følgende måte av Stallman [47]: "Emacs is the extensible, customizable, self-documenting real-time display editor.". I bunn er Emacs en ren teksteditor, men den har store muligheter for utvidelser og tilpasninger gjennom en innebygd Lisp interpreter (Elisp) som er en dialekt av Lisp spesielt tilpasset for utvidelser som støtter tekstediteringen. Emacs er åpen kildekode og er lisensiert under GPL-lisensen.

Emacs er et av de mest oversatte (*ported*) ikke-trivielle program i verden. Det kjører på en lang rekke operativsystemer, blant annet de aller fleste Unix-type systemer, MS-DOS, MS Windows og OpenVMS. Emacs kjører både i tekstterminaler og i grafiske omgivelser hvor det benyttet X Window System som grunnlag sitt grafiske grensesnitt enten direkte eller via *widget toolkits* som Motif, LessTif eller GTK. På Mac OS X og MS Windows benytter Emacs de innebygde grafiske systemene disse operativsystemene tilbyr [56].

Editeringsmodus

Emacs tilpasser seg til hvilken type tekst man editerer. Dette kalles "Major Modes" i Emacs, og det finnes slike for vanlige rene tekstfiler, kildekode for et utall programmeringsspråk, HTML og andre nettrelaterte filtyper, Tex og LaTeX, og en rekke andre typer tekstfiler. Hvert modus tilpasser Emacs til teksttypen, blant annet ved å tilby "syntax highlighting" og tilgang til spesielt tilpassede editeringskommandoer. Emacs støtter også "Minor Modes" som gir ytterligere tilpasninger ut over det "Major Modes" gir [56].

Utvidelser

Elisp har blitt brukt til å utvikle en stor mengde utvidelser til Emacs som spenner over alt fra "Major Modes" for de fleste programmeringsspråk, til

innebygd epost-leser, nettleser og ulike spill. Dette oppnås ved å benytte essensielt tre ulike metoder for tilpasning og utvidelse:

Customize extension Ved hjelp av "customize extension" kan brukere sette en rekke variabler som tilpasser miljøet. Dette kan gjøres direkte i Emacs-grensesnittet og brukes typisk til å tilpasse egenskaper som fargetema [56].

Makroer Inntasting av kommandoer og tekst kan samles i makroer som kan spilles av for å automatiseres komplekse repeterende oppgaver. Makroer blir ofte benyttet i en gitt situasjon og forkastet etterpå, men de kan også lagres og gjenbrukes [56].

Emacs Lisp Emacs har en modulær oppbygning bestående av en rekke separate og uavhengige funksjoner, hvor en bruker kan utvide og tilpasse Emacs ved å lage nye eller endre eksisterende funksjoner og variabler. Dette kan også gjøres under kjøring som gjør det mulig å tilpasse Emacs mens man editerer filer. Et sett funksjoner kan samles i bibliotek og distribueres [45].

Emacs som rammeverk

Emacs er et meget fleksibel og tilpasningsdyktig editor, men i dette ligger også en begrensing som gjør den mindre egnet som rammeverk i forhold til Eclipse. Emacs er fokusert på editering av tekst og dermed ikke i så stor grad en egnet plattform som kan utvides med funksjonalitet som krever grafiske grensesnittelementer eller som fokuserer på egenskaper ut over det som er relatert til selve editoren.

8.2.2 NetBeans

NetBeans omfatter i hovedsak to produkter: The NetBeans Platform som er en rammeverk for utvikling av Java desktop-applikasjoner og NetBeans IDE som er en Java utviklingsomgivelse utviklet på The NetBeans Platform. Begge er åpen kildekode gitt ut under Sun Public License.

The NetBeans Platform består av to lag, en *NetBeans Core Runtime* som tilbyr generisk funksjonalitet for desktop-applikasjoner og en "NetBeans Open API" som er et *public* dokumentert grensesnitt til "NetBeans Core" [5]. The NetBeans Platform muliggjør utvikling av applikasjoner ved hjelp av et sett

selvstendige programvarekomponenter kalt “modules”. En “module” er en Java Archive (JAR) fil som inneholder selve Java-kildekoden som samhandler med the NetBeans Open API og en manifest-fil som identifiserer JAR-filen som en “module”. Uavhengig utviklede moduler kan utvide hverandre og muliggjør tredjepartsutvidelser av programmer utviklet med The NetBeans Platform [57].

NetBeans IDE består av NetBeans Core og en rekke moduler som tilbyr funksjonalitet som til sammen utgjør et fullverdig IDE, blant annet finner man Java-spesifikk funksjonalitet som en egen module, kode-editoren som en egen modul, etc [5].

NetBeans som rammeverk

Når man sammenligner NetBeans og Eclipse så ser man fort at de har en rekke likheter. Begge baserer seg på en liten kjerne som bygges ut med en rekke uavhengige moduler og begge tilbyr både et rammeverk for å bygge programvare og en IDE. Dette gjorde valget mellom de to i hovedsak et pragmatisk valg. Veilederen min var interessert i oppgaver rundt Eclipse, og via han kom jeg i kontakt med en annen student som jobber med et programvaretillegg til Eclipse som delvis omfatter den funksjonaliteten jeg ønsker å se på. Det falt seg dermed naturlig å samarbeide med vedkommende om dette programvaretillegget. Videre så ser Eclipse ut til å ha større “mind share” for tiden som gjør at det er lettere å finne god litteratur på denne plattformen, og samtidig gjør den til en mer populær plattform å utvikle ulike miljøer for en rekke programmeringsspråk i.

8.3 Kommunikasjonsverktøy

Tidligere har jeg sett på ulike typer kommunikasjonsverktøy og hvilke som er det er hensiktsmessig å integrere. En rekke faktorer har vist seg å være viktig for hvordan åpen kildekode-utviklere kommuniserer og tilegner seg kunnskap. Av disse vil jeg understreke følgende som viktige egenskaper for et integrert kommunikasjonsverktøy i Eclipse for bruk i åpen kildekode-utvikling:

- Synkron kommunikasjon for direkte problemløsning og diskusjon
- Asynkron kommunikasjon for refleksjon og “re-experience”

- Arkivering av kommunikasjon som er åpent tilgjengelig, legger også tilrette for “re-experience”
- Støtte for både en-til-en og mange-til-mange kommunikasjon
- Muligheter for å dele kontekstspesifikk informasjon i forhold til det man jobber på
- Mulighet for å sette statusinformasjon

Jabber peker seg her ut som en velegnet kandidat. Jabber er distribuert under en åpen kildekode-lisens og er en sikker, fleksibel og utvidbar alternativ for lynmeldinger (*instant messaging*). Dette betyr at det kan tilpasses til å både håndtere synkrone og asynkron kommunikasjon.

8.3.1 Jabber - teknisk oversikt

Jabber er et sett av XML-protokoller og teknologi for “streaming” som lar personer dele meldinger, statusinformasjon og annet strukturert informasjon over Internett. Disse protokollene er fritt tilgjengelig og det finnes en rekke tilgjengelige implementasjoner av klienter, servere, komponenter og kode-bibliotek. XML-protokollene er standardisert gjennom Internet Engineering Task Force (IETF) under navnet *XMPP*. Spesifikasjonene til XMPP er publisert som RFC 3920 og RFC 3921.

Jabber har en desentralisert arkitektur som kan ligne på epost. Servere kan settes opp individuelt eller de kan kommunisere med hverandre. Dette sammen med støtte for SASL[1] og TLS[2] gir god sikkerhet. Videre er Jabber utvidbart gjennom mulighetene i XML navnerom. Utvidelser kan bygges på toppen av kjerneprotokollene for å imøtekomme spesielle behov. Dette gjør Jabber til et fleksibelt rammeverk for lynmeldinger og kan benyttes til å utvikle funksjonalitet som omfatter blant annet nettverkadministrasjon, innholdsaggregering, samarbeidsverktøy og fildeling [22].

Jabber kan på mange måter minne om epost. Jabber har en klient-server arkitektur, i motsetning til *peer-to-peer* arkitektur som er vanlig blant mange andre meldingssystemer. Kommunikasjon med jabber fungerer som med epost i at den går over en rekke distribuerte servere som benytter en felles

[1]Simple Authentication and Security Layer (SASL): rammeverk for autentifisering og datasikkerhet i Internett-protokoller

[2]Transport Layer Security (TLS): kryptografisk protokoll for å sikre kommunikasjon over Internett

protokoll. Klienter kobler seg til en server for å motta og sende meldinger til andre brukere på samme server eller en som er koblet på samme nettverk. Jabber benytter også som er basert på DNS og godkjente URI-format. Dette gjør at de er på samme format som epost-adresser (bruker@tjener).

Derimot hvor epost fungerer ved å lagre og videresende leverer jabber-servere meldinger i sanntid ved å holde styr på når du er pålogget. Statusinformasjon om du er logget på eller ikke kan også deles med andre, som kan ha deg i vennelista si og se når du logger på og av samt lese om du er opptatt fra statusinformasjonen du har satt.

Kapittel 9

Design - kommunikasjonsstøtte i et utviklingsrammeverk

Jeg vil her spesifisere kravene nærmere ut over de som ble beskrevet i kapittel 7 og 8, gitt valget av Eclipse og Jabbber som rammeverk. Eclipse sitt grensesnittparadigme er basert på *editors*, *views* og *perspectives*. *Editors* tilbyr funksjonalitet for å åpne, editere og lagre objekter. De følger en åpne-lagre-lukke syklus på samme måte som filsystemverktøy, mer er tettere integrert. Aktive editorer kan bidra med *actions* til menyer og verktøylinjer.

Views tilbyr kontekstuell informasjon om objekter som brukeren arbeider på. Et view kan assistere en editor ved å vise kontekstuell informasjon om dokumentet som blir editert. For eksempel så bruker *standard content view* en JFace trestruktur for å presentere en strukturert oversikt over innholdet i den aktive editoren hvis dette er tilgjengelig for filtypen. Et view kan endre på andre views ved å gi informasjon om et aktivt objekt. Views har en enklere livssyklus enn editors, endringer lagres og vises til brukeren umiddelbart [7].

En *workbench* kan tilby flere ulike *perspectives* hvor bare ett av de kan være synlig til enhver tid. Et *perspective* inneholder sine egne *Editors* og *views*. Eclipse-plattformen tilbyr standard perspektiver for ressursnavigasjon, online hjelpfunksjonalitet og team-funksjonalitet. Et perspektiv styrer synlighet og utseende til de ulike elementene i grensesnittet. Man kan raskt skifte mellom ulike perspektiv for å jobbe på ulike oppgaver. Perspectives kan også endres og tilpasses for å bedre imøtekomme en gitt oppgave [19].

JFace har to egenskaper som det er interessant i se nærmere på: *actions* og *Viewers*. *Actions* muliggjør brukerhandlinger som kan være uavhengig av hvor de tilhører i grensesnittet. En *actions* kan representere en kommando

som kan aktiveres gjennom en knapp, menyelement, eller en verktøylinje-knapp. På denne måten kan man gjenbruke samme *actions* på flere steder i grensesnittet. *Viewers* er modell-baserte adapter for visse SWT grafiske grensesnittelementer. *Viewers* bidrar med elementer for typisk oppførsel og er semantisk på et høyere nivå enn SWT. En rekke *Viewers* finnes, blant annet for lister, trestrukturer og tabeller. Flere *Viewers* kan være åpne for samme modell eller dokument og oppdateres automatisk ved endringer.

Nye verktøy integreres inn i dette paradigmet. Her er det ønskelig med tett presentasjonintegrasjon så man vil benytte de grensesnittkomponentene som eksisterer. Verktøy kan utvide eksisterende editors, views og perspectives ved å legge til actions til eksisterende menyer, verktøylinjer uavhengig eller avhengig av aktiv editor. Videre kan actions legges til i menyer og kontekstmenyer, eller legges til som nye views, action sets og snarveier til et eksisterende perspectives. Disse elementene vil i stor grad bli brukt for å få tettest mulig integrasjon.

Eclipse er meget fleksibelt når det gjelder hvilke elementer som kan plasseres i en workbench og hvor de kan plasseres. Valget av Eclipse gir dermed stor frihet for brukerne til å tilpasse sitt eget grensesnitt, noe som også vil gjelde de komponentene jeg vil jobbe på siden de bruker samme grensesnitt-byggeklosser.

Med utgangspunkt i beskrivelsen av den ønskede funksjonaliteten i kapittel 7 vil jeg nå konkretisere dette til et design for et programvaretillegg i Eclipse. Dette vil bestå av tre hovedelementer. Et kommunikasjonsverktøy som består av et *chat view* og en kontaktliste view som viser personer og prosjektgrupper. En personlig *issue tracker* som viser ulike oppgaver som er blitt tildelt din bruker. Det siste er funksjonalitet for å vise kontekstuell informasjon for en fil som er gitt enten via chat eller direkte lagt inn i en fil.

9.1 Kommunikasjonsverktøy - *chat*

Kommunikasjonsverktøyet er jabber-basert og vil støtte både direkte chat mellom to personer og chatrooms hvor flere kan delta og alle ser hva alle skriver. Begge disse vil bli implementert som *views*. En kontaktliste som også vil bli implementert som et *view* som inneholder:

- En liste med navn på andre deltagere. Du legger selv til de du vil ha i listen. Disse vil ligge under 'Contacts'.

- En liste med chatrooms. Disse representerer chatrooms. En gruppe hører til et prosjekt og du vil automatisk få opp alle chatrooms som er opprettet for et gitt prosjekt. Disse vil ligge under 'Chatrooms'.

Kontaktlisten vil vise et ikon og et navn for hver kontakt. Ikonet vil indikere status for vedkommende. Rødt ikon for ikke pålogget og blått ikon for pålogget. Listen vil ha muligheter for å vise tilleggsinformasjon ved å holde musepekeren over en kontakt. Dette vil omfatte informasjon om operativsystem og rolle. På denne måten kan man lettere finne frem til personer på samme plattform som seg selv som kan hjelpe i feilsøk, og å finne ansvarsområde for deltagerne.

Samtaler kan knyttes til ulike kontekster. Disse representeres via "context" og "location". Context setter en beskrivelse for en samtale som denne blir gruppert under. Location peker til en bestemt fil i prosjektet. Et chatroom får automatisk satt context til samme navn som det selv har.

Programvaretillegget vil ha en egen konfigurasjonsdialog hvor man setter de nødvendige feltene for å koble seg til en jabber-server: tjenernavn, portnummer, brukernavn og passord. I tillegg er det et ekstra felt hvor man kan fylle inn sin rolle i prosjektet. Hensikten med rollefeltet er for at andre lettere skal kunne finne ut hvem de bør henvende seg til hvis de har spørsmål om bestemte deler av koden. Hvis man ikke har opprettet en jabber-bruker før vil denne bli opprettet basert på informasjonen gitt i denne dialogen ved første innlogging hvis serveren som er spesifisert har åpnet for fri registrering.

Det vil være mulig å sende med informasjon som vedlegg til meldinger for å gi grundigere informasjon i feilsøk i samarbeid med andre. Dette vil inkludere detaljert informasjon om hvilken plattform man bruker og informasjon om kompilator og bibliotek brukt i kompilering. Dette vil kunne hjelpe ved å lettere kunne spore eventuelle forskjeller mellom plattform og bibliotek brukt av de ulike partene i feilsøk. Disse vil bli listet i grensesnittet som 'attachments'. Man kan sende ved informasjon ved å høyreklikke på ulike elementer i grensesnittet som for eksempel en kompileringsfeil og velge 'attach to discussion' menyvalget. Prosjektnoden i "Package Explorer" vil også ha ett nytt menyvalg 'Attach build environment'.

9.2 Personlig "issue tracker"

Hovedformålet med dette view'et er å samle oppgaver som er blitt tildelt en bruker for et prosjekt. View'et vil inneholde informasjon om feilrapporter,

kodefiler (*patches*) og funksjonsønsker som er tildelt vedkommende. I tillegg vil den vise en oversikt over filer som brukeren har satt at han vil følge. Dette er en oppdeling som er vanlig i eksterne feilrapporteringssystem og bør falle naturlig for åpen kildekode-utviklere. Disse vil bli presentert i en liste hvor man kan klikke på den enkelte for å åpne en dialog som viser detaljert informasjon om denne sammen med følgende funksjoner (dette er for feilrapporter, det vil være lignende funksjonalitet for de andre typene):

- 'Close bug' knapp
- 'set status' knapp i samsvar med en rullegardinmeny for statusvalg
- 'reasign bug' knapp i samsvar med en rullegardinmeny for andre deltagere i prosjektet
- 'category' knapp i samsvar med en rullegardinmeny for kategorivalgene

Listen vil ha filtreringsmuligheter for hva man vil vise i listen som samsvarer med status- og kategori-valgene i dialogen for detaljert informasjon. Dette grensesnittet vil kommunisere med eksterne feilrapporteringssystemer hvor det henter informasjonen og sender statusendringer. I tillegg til å hente informasjon fra eksterne kilder og vise disse vil brukeren kunne merke filer han vil følge. Han vil så få opp hvis filene har blitt endra, og om noen har endra metadata for en fil.

9.3 Kontekstuell informasjon

Kontekstuell informasjon er ønskelig for å hjelpe personer med å få økt forståelse for koden. Økt bruk av kontekstuell informasjon rundt koden vil kunne hjelpe å kodifisere mer av den skjulte kunnskapen som prosjektmedlemmene sitter inne med. Kontekstuell informasjon vil her bestå av to typer. *Metadata* om en gitt kodefil vil i hovedsak kunne brukes for å markere hvem som jobber på hva, og status for en fil. I tillegg vil *diskusjoner* kunne bli lagret for en gitt kodefil og fremhentes i etterkant for refleksjon. Dette vil bli implementert som ulike elementer:

Metadata vil bli lagrer i selve kodefilen syntaktisk som en kommentar. Spesielle merkelapper vil benyttes slik at de kan bli analysert programmatisk, som tillater at man kan søke i filene etter gitte merkelapper og vise disse som resultat i ulike deler av grensesnittet. Denne informasjonen kan dermed leses

og redigeres direkte i kodefilen, jeg finner det ikke nødvendig å ha et eget grensesnitt for manipulering og visning av denne informasjonen.

Diskusjoner knyttet til en fil vil bli åpnet som en diskusjon i chat-vinduet under en egen arkfane som har filnavnet som overskrift. Denne arkfanen vil i tillegg ha noe ekstra informasjon. Den vil vise når filen sist ble endret og av hvem, samt hvem som har ansvar for filen hvis det er satt. Du skal kunne klikke på navnet til den som sist har endret eller som har ansvar for å åpne en chat med vedkommende. Det vil være to metoder for å åpne diskusjonsdialogen:

- Ikon som viser om en fil har en diskusjon knyttet til seg
- Menyvalg i kontekstmeny for en gitt fil

Når en 'location' er satt for diskusjonen har man mulighet for å lagre diskusjonen i en egen fil som blir knyttet til kodefilen som er satt i 'location'. I grensesnittet vil dette kunne gjøres via en knapp for lagring til fil for den aktive konteksten. Diskusjoner som er knytta til enkeltfiler vil lagres i versjonskontrollsystemet sammen med filene de hører til slik at alle kan få tilgang til disse når de oppdaterer sin lokale kopi. De vil bli lagt til prosjektet slik at de blir sjekket inn sammen med resten. Denne filen vil bli lagret på formatet: '.filnavn.chat'. Filnavn vil bli satt til samme navn som Java-filen diskusjonen er knytta til. Bruker "dotfiles" (filer som begynner med et punktum) for at de skal være skjult på Unix-systemer slik at de ikke fyller opp med "rot" for de som benytter andre verktøy enn Eclipse for å aksessere kodelageret. Filen er i XML-format som åpner opp for mange muligheter for prosessering for presentasjon. Ulempen her er at filen ikke er så lettlest som ren tekst hvis noen vil se på den direkte. Det er derimot rimelig trivielt å skrive et skript som analyserer XML-filen og gir ut ren tekst.

Andre alternativer hadde vært å lagre automatisk alle samtaler som er har satt en 'location' eller å kunne velge når man skriver om man bare vil sende eller sende og lagre. Førstnevnte av disse alternativene er det mest gjennomsiktige for brukeren, men det medfører at brukerne ikke har noe kontroll med hva som lagres for en fil og man kan dermed ende opp med veldig mye støy i form av informasjon som ikke er relevant. Velge for hver gang man skriver på sin side virker litt påtrengende på brukeren ved at man har enda mer å forholde seg til å grensesnittet og man må hele tiden velge mellom de to midt i en diskusjon. Å velge å kunne lagre en diskusjon på et vilkårlig tidspunkt føler jeg er et greit kompromiss her.

Hvis en fil har en diskusjon knyttet til seg vil dette bli vist i grensesnittet på to måter. I filliste-view'et vil man se et ikon ved siden av filnavnet. Ved å klikke på dette ikonet får man opp diskusjonen i en chat-arkfane. I tillegg kan man benytte høyreklikk-menyen for en gitt fil. Når filen har en diskusjon knyttet til seg vil menyvalget 'Discussions' være aktivert. Klikker man på denne vil den åpne diskusjonen i en chat-arkfane.

En aktuell problemstilling er om det skal være mulig å fjerne diskusjoner som er knyttet til en fil. Dette kunne være aktuelt blant annet i tilfeller hvor diskusjonen omhandler kode som senere har blitt endret og derfor ikke lenger er aktuell. Jeg har valgt å ikke tilby å kunne slette diskusjoner fra grensesnittet av flere grunner. En diskusjon vil ha følgende informasjon knyttet til seg:

- Dato
- Hvilket revisjonsnummer filen den er knyttet til hadde når diskusjonen ble lagret
- Kontekst

Med denne informasjonen vil man kunne se hvilken versjon av kodefilen en diskusjon hører til og man kan lett finne frem til hvilke endringer som diskusjonen omhandler. Selv om dette ikke nødvendigvis er relevant for hvordan koden er utformet senere vil det fremdeles ligge et potensiale for læring i forhold til de problemstillingene som var relevante da diskusjonen ble lagret og disse diskusjonene har dermed potensielt en verdi langt fremover i tid.

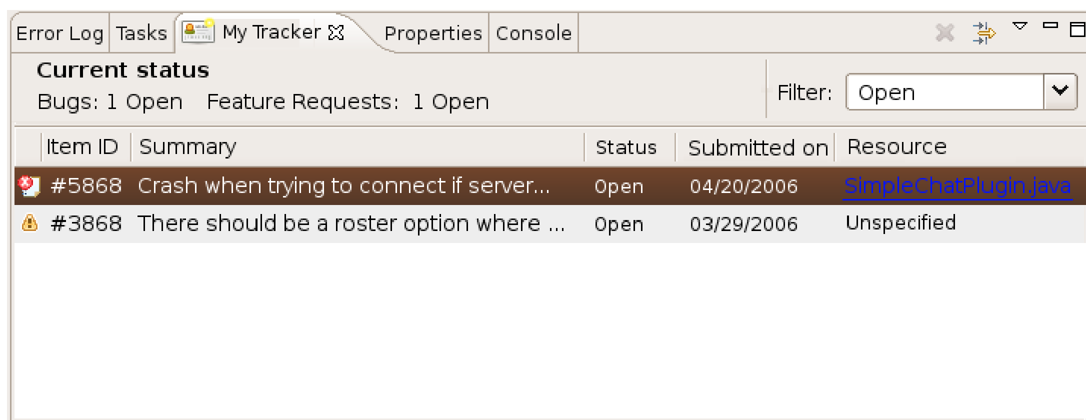
Revisjonsnummer og hvem som har sjekket inn endringen hentes fra merkelapper som revisjonssystemet støtter. Ved å legge inn strengen "*Id*" i kodefilene vil revisjonssystemet ekspandere dette til en streng som viser dato, revisjonsnummer og hvem som sjekket inn endringen. Dette støttes av revisjonssystemer som CVS og Subversion.

Jeg vil se på bruken av denne funksjonaliteten ved å beskrive tenkte situasjoner i tre ulike scenario. I scenariene vil jeg også vise prototyper av grensesnittene som viser hvordan det vil kunne se ut. Disse grensesnittprototypene er modifiserte skjermbilder fra Eclipse sitt grensesnitt. Resten er laget i grafikkprogrammet The Gimp foruten noen ikoner som er hentet fra Tango-prosjektet[1].

[1] http://tango-project.org/Tango_Icon_Gallery

9.4 Scenario I - Bruk av personlig "issue tracker"

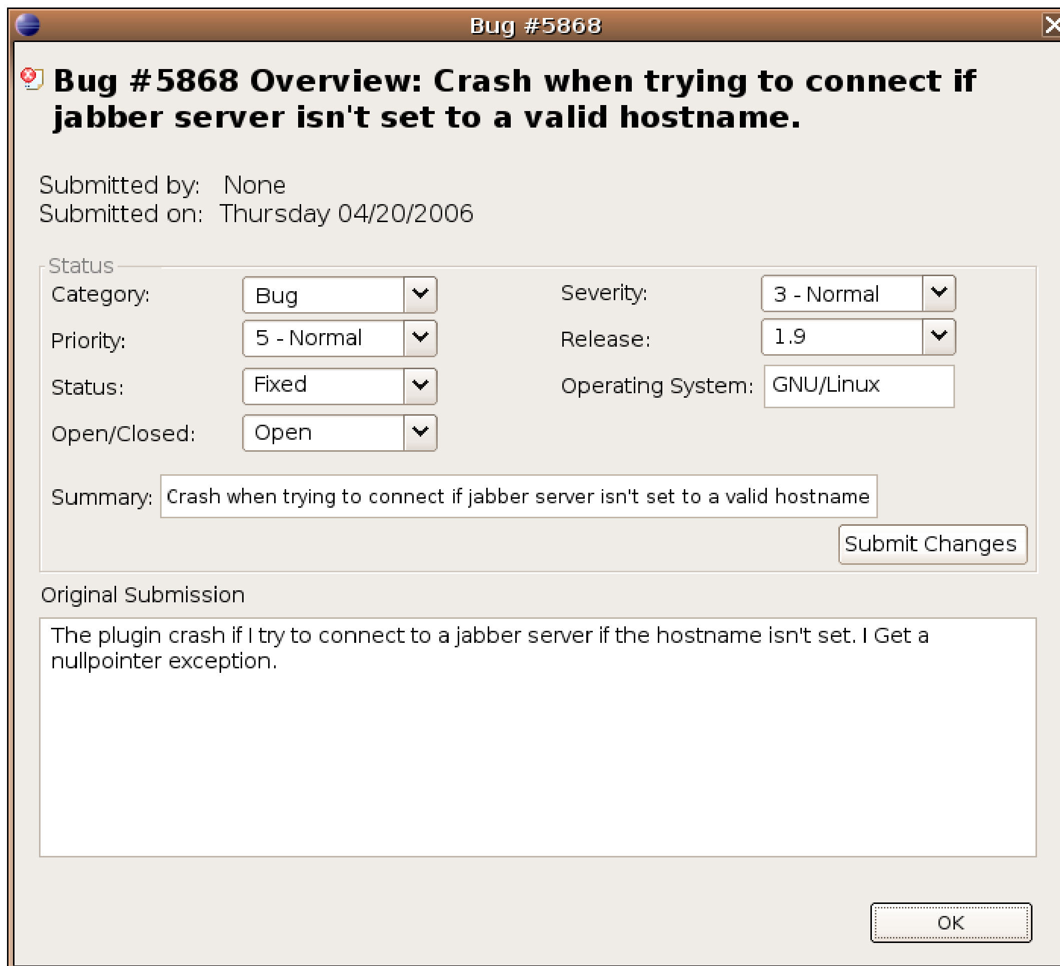
I åpen kildekode-utvikling skjer utviklingen av mennesker fra hele verden til alle døgnets timer. Det vil dermed stadig skje endringer til neste gang du logger på og skal jobbe med prosjektet. Derfor er det viktig å raskt og effektivt kunne bli oppdatert på hva som skjer.



Figur 9.1: "Issue tracker"-arkfanen

Deb starter opp Eclipse og oppdaterer kildekoden fra Subversion datalageret, og logger seg på Jabber-tjenesten. Hun klikker på "My Tracker" arkfanen (se figur 9.1) for å se om det er noe nytt som er lagt til for henne. Hun ser at hun blitt tildelt 1 ny feilrapport. Hun klikker på lenken til feilrapporten og får opp feilrapporten i et eget vindu, hvor informasjonen er hentet fra en ekstern feildatabase. Hun ser at feilen er i kode hun kjenner til og åpner den aktuelle kodefilen. Hun forsøker å reprodusere feilen og verifiserer at feilen er reell. Det viser seg å være en enkel fiks som hun utfører og sjekker inn en ny versjon av kodefilen hvor feilen er rettet. Deretter klikker hun på "My Tracker" arkfanen igjen, velger feilrapporten og skifter status til "fixed" (se figur 9.2). Den forsvinner så fra listen som er satt til å filtrere ut alt annet enn de som har 'Open' som status.

Etter å ha fikset feilen ser Deb at en fil hun har valgt å overvåke har fått endret metadata. Hun klikker på endringen og filen åpnes og hun leser over metadata-informasjonen. Hun ser at "Maintainer"-feltet har blitt endret fra "None" til "Sid". Hun kikker ned på kontaktlisten og ser at Sid er pålogget og starten en chat med vedkommende. Hun forteller at hun har overvåket



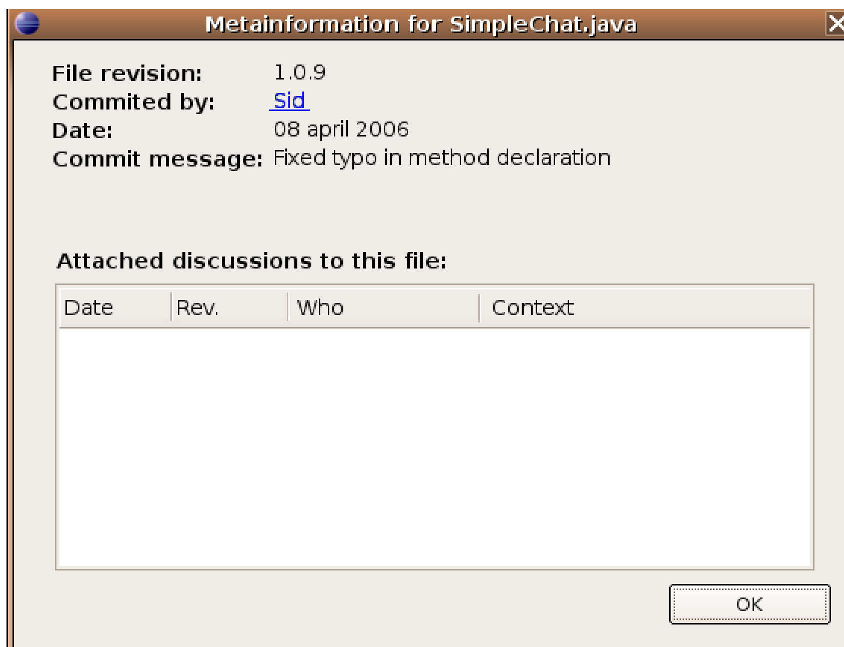
Figur 9.2: Dialog som viser detaljer for en valgt feilrapport

filen fordi hun tidligere hadde forespurt både på IRC og epostlisten om en ønsket endring i denne koden, men ikke fått noen respons. Hun legger så frem problemstillingen og Sid sier han kan ta en kikk på det og forsøke å få inn endringen samtidig med det han jobber med i samme fil.

9.5 Scenario II - Få hjelp og yte hjelp

Å sette seg inn i ukjent kode kan være tungt i alle former for programvareutvikling. I åpen kildekode-utvikling er koden fritt tilgjengelig, men det betyr ikke nødvendigvis at den er godt dokumentert. Det kan derfor være nødvendig

å spørre om hjelp eller ty til annen informasjon ut over selve kodefilene.

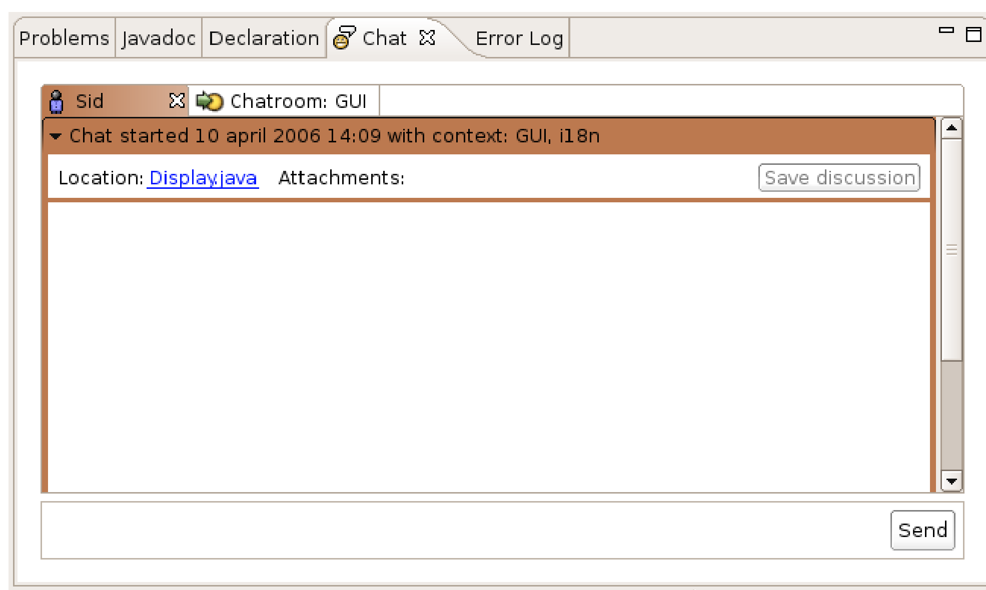


Figur 9.3: Dialog som viser kontekstuell informasjon for en valgt fil

Ian har tatt interesse i et nytt prosjekt hvor han ønsker å legge til støtte for internasjonalisering slik at han kan oversette programmet til sitt eget språk. Han har problemer med få dette til å fungere korrekt på grunn av hvordan flere metoder i grensesnittkoden for dialoger fungerer. Han sjekker metainformasjonen (se figur 9.3) for kildekodefilen han ser på og ser at siste innsjekking av endringer på denne filen ble gjort av Sid. Ingen diskusjoner er knyttet til filen, så han klikker på navnet til Sid som er en lenke som vil åpne et samtalevindu med vedkommende. 'location' for denne samtalen vil bli satt til kildekodefilen som Ian satt og jobbet med. Han setter 'context' til å være 'GUI, i18n' (se figur 9.4).

Sid er også pålogget og Ian forklarer problemene han har med dialogkoden. Sid ser på koden det er snakk om og forstår problemstillingen. Han forklarer at koden er sånn på grunn av hensyn til ytelse. De diskuterer litt frem og tilbake og kommer frem til at det må en større *refactoring* til for å kunne løse dette problemet på en tilfredsstillende måte. Sid foreslår også å sende en epost til epostlisten for prosjektet for å diskutere behovet for en større kodeendring.

Sid har tidligere sett at flere andre også hadde spurt spørsmål rundt samme



Figur 9.4: Chatvindu

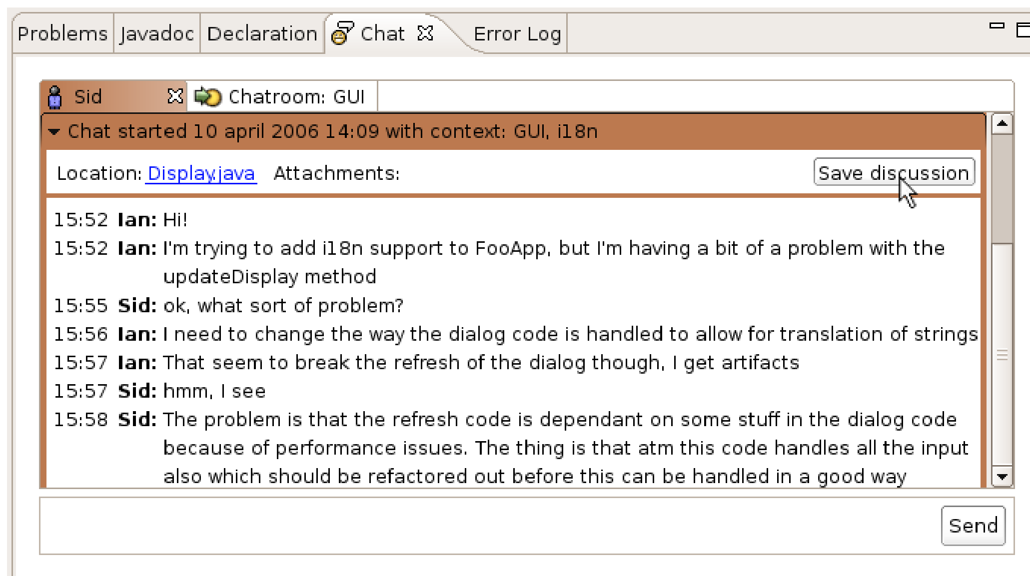
koden, så han velger å lagre diskusjonen slik at den blir knyttet til filen, som vil la andre kunne lese den i kontekst av kodefilen (se figur 9.5).

Senere i uka sitter Woody, en annen koder som akkurat har begynt å sette seg inn i koden til samme prosjektet og ser på samme fil som Ian og Sid diskuterte tidligere. Han ser også at denne delen av koden har behov for *refactoring*. Før han begynner å jobbe på oppgaven sjekker han metainformasjonen for filen og ser at den har en samtale knyttet til seg (se figur 9.6). Ved hjelp av denne får han større forståelse for problemstillingen og følger henvisningen til epostlisten som han dermed melder seg på og deltar i diskusjonen rundt hvordan dette bør håndteres.

9.6 Scenario III - Feilsøk

I åpen kildekode-utvikling er det ofte en rekke utviklere og et stort antall plattformer med sine egne særegenheter som skal støttes. Dette kan ofte føre til problemer hvor noe som fungerer for en utvikler på en plattform gir uforutsette problemer på en annen. Å spore og løse slike plattformspesifikke problemer kan være en omstendelig og lang prosess.

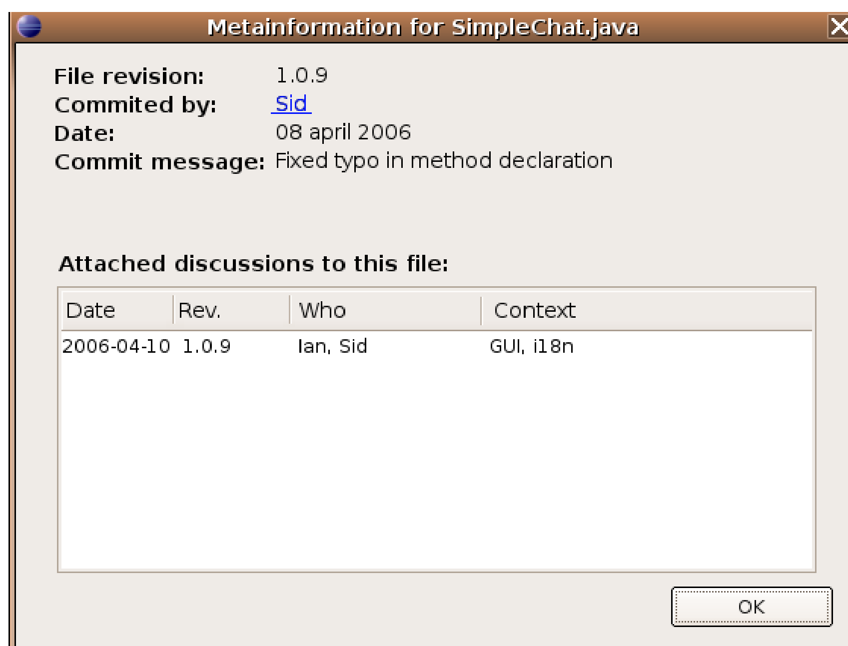
Etter å ha kompilert sin lokale kodekopi fra Subversion datalageret så får



Figur 9.5: Diskusjon om koden

Sid en kompileringsfeil. Sid ser nærmere på feilen uten å se hva som kan forårsake feilen, men mistenker at det kan være en plattformspesifikk feil. Sid åpner filen feilen stammer fra og sjekker hvem som har sjekket inn endringen som førte til problemer og finner at det er Woody. Sid ser så på listen over kodere på prosjektet som er online (se figur 9.7). Til venstre for Woody viser kontaktlisten et rødt ikon som indikerer at han ikke online akkurat nå. Sid holder musepekeren over navnene til de som er pålogget og sjekker mouseover-popup for å se hvilke operativsystem de kjører, og finner en person som kjører samme plattform som seg selv (se figur 9.8). Tar kontakt med spørsmål om vedkommende også har problemer etter gitte innsjekking, noe som bekreftes. Etter en del feilsøk kommer de frem til at en funksjon ikke gir samme resultat ut på Mac OS X som på GNU/Linux som er det vedkommende som sjekka inn endringen kjører (se figur 9.9).

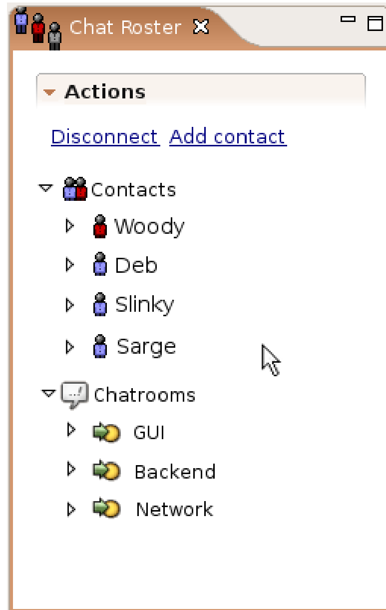
Sid sender så en melding til Woody hvor han beskriver hvilke problemer han har og hva han og Sarge hadde kommet frem til som sannsynlig årsak. Han velger også å sende med ekstra informasjon som tillegg til tekstmeldingen. Han klikker på 'Error Log'-arkfanen hvor han finner kompileringsfeilen. Han høyreklikker på den og velger 'attach to discussion'. Han høyreklikker så på prosjektnoden og velger 'Attach build environment' for å sende med informasjon om hvilke biblioteker som ble brukt under kompilering samt informasjon om Java-installasjonen som benyttes.



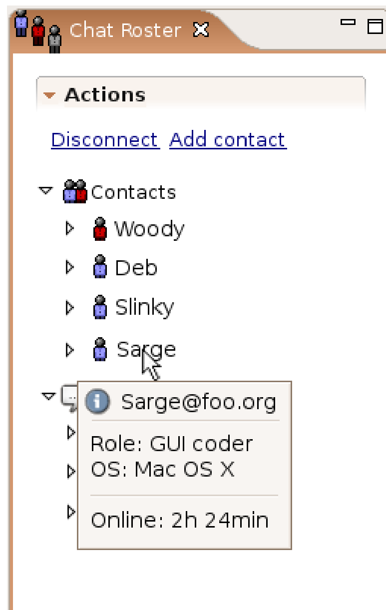
Figur 9.6: Dialog som viser kontekstuell informasjon for en valgt fil

Woody logger på senere den kvelden og mottar meldingen fra Sid. Ved hjelp av informasjonen har har mottatt finner han en måte å endre koden på slik at den vil fungere på Mac OS X også. Han sjekker inn en ny versjon og sender en melding til Sid hvor han skriver at feilen skal være fikset og at han ønsker tilbakemelding på om det fungerer.

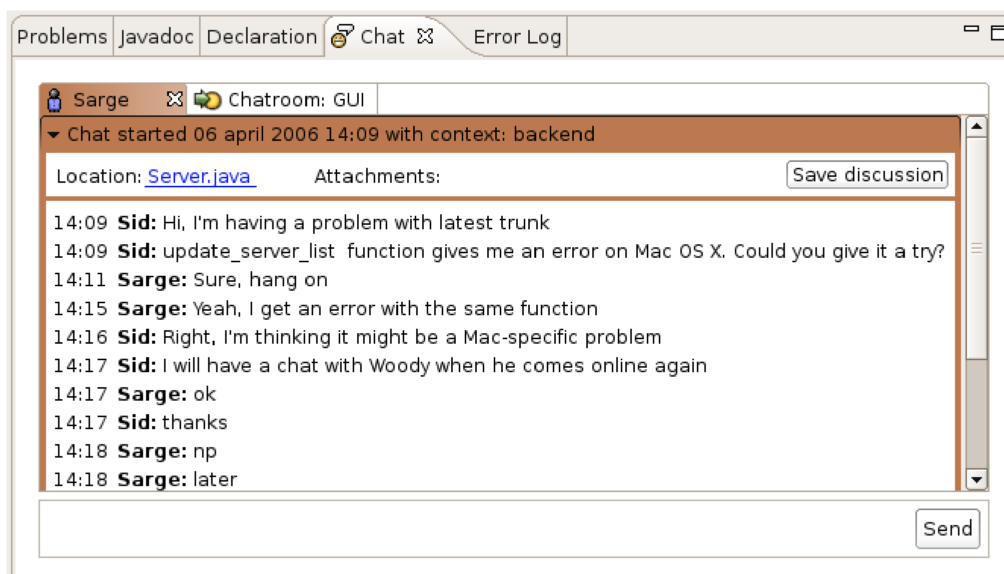
Neste dag starter Sid opp Eclipse igjen, logger på jabber og oppdaterer fra Subversion. Han leser meldingen fra Woody og prøver å kompilere på nytt. Denne gangen fungerer det og han sender en melding tilbake til Woody om at problemet er løst.



Figur 9.7: Dialog som viser kontaktliste og chatroom-liste



Figur 9.8: Dialog som viser mouseover for en kontakt



Figur 9.9: Dialog som en typisk samtale

Kapittel 10

Implementasjon

I dette kapittelet vil jeg beskrive implementasjonen av Eclipse programtillegget (*plugin*). I stedet for å finne opp hjulet på nytt valgte jeg i ekte åpen kildekode-ånd å basere meg på eksisterende åpen programvare gjennom en Jabber-klient kalt SimpleChat [28]. Denne ble utviklet av Arne Mathisen og Frode Angell-Petersen som en prosjektoppgave ved IDI, og videreutviklet av Arne Mathisen for hans diplomoppgave. Jeg brukte denne som et utgangspunkt for mitt arbeid og utvidet med funksjonalitet for kontekstuell informasjon rettet mot de egenskapene jeg ønsker å teste.

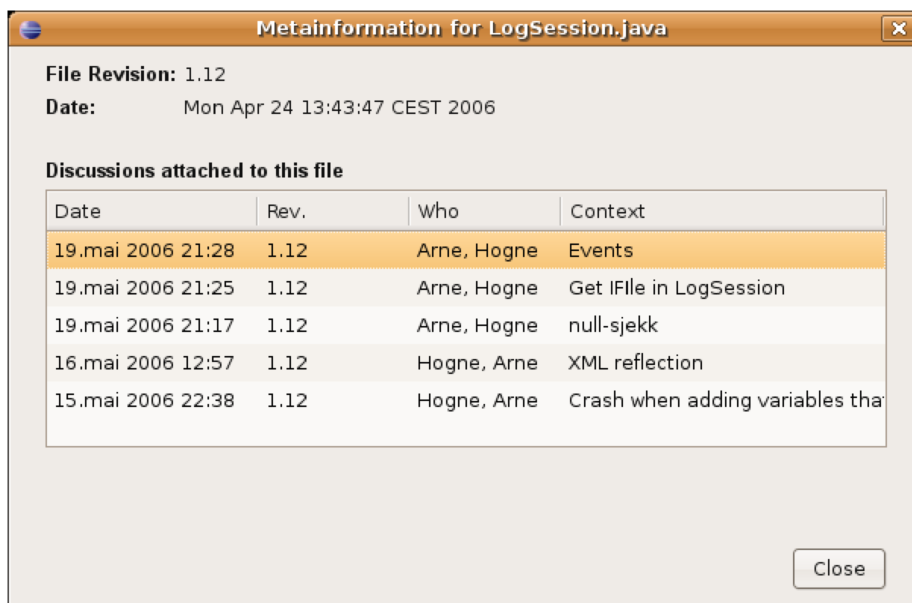
I SimpleChat finner man de vanlige elementene man kjenner igjen fra nær sagt alle jabber-klienter: kontaktliste og Chattevindu hvor man skriver og mottar meldinger. I tillegg til dette har SimpleChat en del ekstra funksjonalitet for kommunisere om og å dele informasjon om koden man jobber på:

- “Context”: man kan sette en kontekst for en samtale
- Eclipse-link: sende lenker til resurser i Eclipse over chat
- “Awareness”-informasjon: sender informasjon til de andre som er logget på om hvilken fil du jobber med
- “location”: kobler en chat til en ressurs i Eclipse

SimpleChat er implementert som et Eclipse-programvaretillegg. SimpleChat lagrer samtalene og *awareness*-informasjon som XML, og benytter biblioteket XStream for dette. XStream tilbyr funksjonalitet for å serialisere fra objekt til XML og deserialisere fra XML til objekt. Kommunikasjonen foregår over

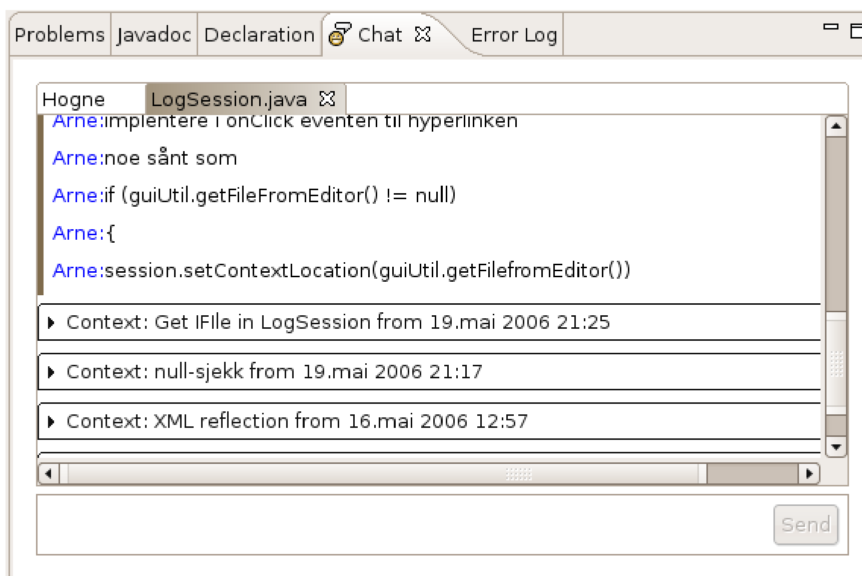
XMPP (Jabber). For dette benytter SimpleChat Smack, som er et åpen kildekode XMPP-bibliotek skrevet i Java.

I SimpleChat blir samtaler lagret når man avslutter Eclipse. Jeg behøver funksjonalitet for å lagre diskusjoner knytta til bestemte filer mens programmet kjører. Dette krevde endringer både i grensesnittet og i funksjonaliteten for serialisering og deserialisering av logger. I grensesnittet ble det lagt til en knapp for lagring av aktiv kontekst når denne er knyttet til en fil i prosjektet. Samtalen blir lagret i en separat XML-fil. Det ble lagt til en session-id som ble brukt til å sjekke om en session allerede er blitt lagret slik at man unngår lagring av duplikate sessions. I stedet blir en eksisterende session erstattet hvis ID er identisk til den som blir lagret.



Figur 10.1: Dialog som viser lagrede diskusjoner for den aktuelle filen

En ny dialog (se figur 10.1) ble implementert for å vise lagrede chats for en gitt fil. Denne er tilgjengelig via høyreklikkmenyen til et element i Package Explorer. Den viser CVS-informasjon for en gitt fil samt at den lister chat-kontekster som er lagret. Disse kan dobbeltklikkes på for å åpnes i en vanlig chat-arkfane hvor arkfanen får overskriften satt til filnavnet konteksten er knytta til. Denne arkfanen (se figur 10.2) inneholder alle kontekstene kobla til denne fila. Den valgte konteksten er ekspandert, mens de andre er minimert. Denne arkfanen har ikke mulighet for å koble til fil og lagre igjen, og sending av meldinger er deaktivert. Den er *read-only*.



Figur 10.2: Arkfane som viser lagrede diskusjoner for 'LogSession.java'

I tillegg ble det gjort en del mindre endringer. Dette omfatter muligheten for å sette hvilket navn som vises på seg selv (alias) i chatten i SimpleChat preferences. I den originale versjonen av SimpleChat er du identifisert ved strengen '$me'. Ved å sette ditt eget alias blir dette brukt i stedet slik at du får sett navnene på de som deltar i samtalen. Dette ble i hovedsak gjort fordi dette blir lagret i loggene, og det er mer hensiktsmessig å kunne se hvem som har deltatt i en samtale i dialogen som lister disse i stedet for at det står '$me' som ikke er spesielt informativt i denne sammenhengen. Dette ble gjort ved å legge til en ny XML-merkelapp (*tag*) for dette. Det ble også lagt til en XML-merkelapp for filrevisjon som benyttes til å merke en samtale med hvilken revisjon av filen som var gjeldende da samtalen fant sted.

Videre ble det gjort en rekke endringer av strenger og menyvalg for å legge mer til rette for testingen av den funksjonaliteten jeg ville fokusere på. Dette ble gjort fordi kobling av ressurser og diskusjoner ble noe endret. I SimpleChat kunne man sette en 'location' som var en kobling til en ressurs. Dette ble endret til å brukes til å koble chats til filer og kalt "attach file" i stedet. Underveis i testingen ble det også gjort en del endringer i grensesnittet. Disse er beskrevet nærmere i neste kapittel.

Kapittel 11

Prototype - testing og evaluering

Det finnes en rekke evalueringsparadigmer som kan legges til grunn for testing [38, s. 340-345]. Som grunnlag for forslaget brukte jeg litteraturstudie og min egen erfaring fra åpen kildekode-prosjekter for å vurdere de ulike forslagene jeg kom frem til. For å teste selve designet velger jeg å benytte meg av HCI-labben ved IDI og kjøre brukbarhetstester av grensesnittet. Testene vil bestå av to deler: observasjon av brukerne som jobber med oppgavene i testen, og en samtale etterpå for å avdekke hvordan brukerne opplevde bruken av programmet samt å få tilbakemelding på eventuelle forbedringer de ser for seg.

Hovedformålet med testene vil være å avdekke problemer med grensesnittet og designet, samt å finne ut om informasjon knyttet til kodefiler kan hjelpe på forståelsen av koden og i hvilken grad testpersonene følte nytten av disse. Testpersonene vil i så stor grad det er mulig bli valgt ut basert på følgende kriterier:

- Har kjennskap til Eclipse
- Har kjennskap til åpen kildekode-utvikling
- Har kjennskap til Java-utvikling

Disse kriteriene er satt fordi jeg ønsker å teste på personer som kan sette seg inn i situasjonen til målgruppen for funksjonaliteten. Dette betyr at de må ha kjennskap til utvikling i omgivelsen og programmeringsspråket for at de skal kunne ta i bruk og potensielt ha nytte av den ekstra funksjonaliteten.

11.1 Testoppgaver

Testen vil bestå av to oppgavetyper. Den første vil gå på forståelse av grensesnittet og vil innebære at testpersonen blir kjent med funksjonaliteten. Den andre typen vil fokusere på kodeforståelse gjennom en konkret implementasjonsoppgave. Koden som vil bli brukt er kildekoden til den originale SimpleChatkodebasen før mine endringer. Det viktigste her blir å se i hvilken grad testpersonene bruker de tilgjengelige hjelpemidlene, ikke om de klarer selve implementeringen.

Et utvalg diskusjoner vil bli knyttet til filene. Disse diskusjonene vil være faktiske samtaler mellom meg og Arne fra implementeringen av mine tillegg til SimpleChat slik at testene blir basert på reelle data. Noen av kodefilene vil ha diskusjoner knyttet til seg, mens de fleste vil ikke ha det. Disse vil ha ulike tema hvor noen kan være til nytte for selve implementasjonsoppgaven hvis testpersonen lokaliserer de.

11.1.1 Oppgavetekst

Innledende spørsmål

Hvor kjent er du med Java-utvikling i Eclipse?

Hvor kjent er du med åpen kildekode utvikling?

Oppgave 1

Utforsk grensesnittet, prøv ut funksjonaliteten som er en del av SimpleChat. Fortell høyt hva du gjør og tenker.

Oppgave 2

Hvordan ville du ha implementert en ny XML-merkelapp *sessionID* i LogSession.java. Denne skal kunne unikt identifisere en session (samtale). Sett deg inn i koden, bruk tilgjengelige hjelpemidler og forsøk å implementer. Debug eventuelle feil under kjøring.

11.1.2 Testprosedyre

Før de begynner testen får testpersonene en kort forklaring på hva SimpleChat innebærer av funksjonalitet. Dette omfatter bare hva det er i grove trekk, uten noen direkte forklaring på elementene i grensesnittet og spesifikk detaljering av funksjonaliteten.

Under testen vil testpersonene bli bedt om å snakke høyt om hva han gjør, tenker og forventer for å avdekke hvordan vedkommende opplever og forstår grensesnittet og funksjonaliteten. Etter oppgavene vil det bli en samtale hvor testeren vil få forklart hva som er tanken bak funksjonaliteten og designet og ha mulighet for ytterligere tilbakemelding i lys av dette. Testingen vil bruke en kombinasjon av videoopptak og direkte observasjon og "talk-aloud".

11.2 Testutførelsen

Fire personer med bakgrunn fra informatikk/datateknikk ved IDI testet implementasjonen ved fire ulike anledninger. I første test ble det i første oppgave brukt direkte observasjon og "talk-aloud", og video for resten, mens i de tre påfølgende testene ble det brukt direkte observasjon og "talk-aloud" gjennom hele testen fordi kvaliteten på videoen fra første test viste seg å være av utilfredsstillende kvalitet.

Før test 3 ble det gjort noen mindre endringen i grensesnittet basert på tilbakemeldinger i tidligere tester.

- Decorator for filer ble endret fra [Chat] til [Chatlog] i et forsøk på å bedre beskrive hva den markerer.
- Tooltip-teksten for lenken for attached files ble også gjort mer spesifikk ved å presisere at man finner SimpleChat-menyen for attaching av filer i view'et Package Explorer .

Videre ble det før test 4 gjort en rekke endringer i grensesnittet. Basert på tidligere tester ble det gjort en del endringer for å i større grad få fokusert på funksjonaliteten og få fjernet de mest opplagte problemene med grensesnittet som lett kunne rettes opp i. Dette ble også gjort for å øke sjansen for å få tilbakemeldinger som ikke gikk på problemstillinger som allerede var klarlagt.

- Fjernet $me dekoratøren. Denne skapte forvirring og var ikke funksjonalitet som skulle testes.

- Fjernet “(keyword)” CVS decorator, denne tok stor plass og skapte ekstra støy i Package Explorer som gjorde at man vanskelig så [Chatlog].

- Lagret flere diskusjoner under LogSession.java og endret navn på de eksisterende slik at de ikke skulle være så innlysende og opplagt for oppgave 2, og dermed forhåpentligvis få oppgaven litt mer realistisk.

- Endret navn på meny-elementet fra “Open Metainformation” til “View Saved Chats” for å få en mer konsekvent og presis terminologi i grensesnittet.

- Ga filnavn med [Chatlog] farge (mørkeblå).

- Endre “View Saved Chats” meny-elementet til å bare være aktiv for filer med chatlogger knytta til seg.

- Fjernet “Set selection as additional info” fra editor høyreklikkmeny siden dette feltet ikke er vist i chat-grensesnittet lenger.

- Fjernet “Set line number as location” fra editor høyreklikkmeny siden jeg bare vil bruke “location” for attached files.

11.3 Resultater

Jeg vil her presentere resultatene fra testene, samt å foreslå ulike endringer basert på disse. Testloggene er å finne som vedlegg i kapittel 14. Det kom fram flere interessante problemstillinger under testingen samt en rekke mindre grensesnittrelaterte problemer.

Testingen avslørte en rekke problemstillinger relatert til grensesnittet, blant annet i forhold til tilbakemelding for handlinger og terminologien brukt. Disse ble forsøkt utbedret basert på tilbakemeldinger fra tidligere tester, før test to og test fire for at disse ikke skulle ta fokus bort fra eventuelt andre problemstillinger i de påfølgende testene.

Det er derimot problemstillinger rundt mer grunnleggende designvalg som er de mest interessante tilbakemeldingene fra testingen. Dette gikk i hovedsak på hvordan koblingen mellom chats og filer ble gjort og det viste seg at denne ikke var så intuitiv og fleksibel som testpersonene forventet og ønsket. Jeg vil nå gå nærmere inn på de problemstillingene som kom frem, samt foreslå endringer basert på disse.

11.3.1 Grensesnittendringer

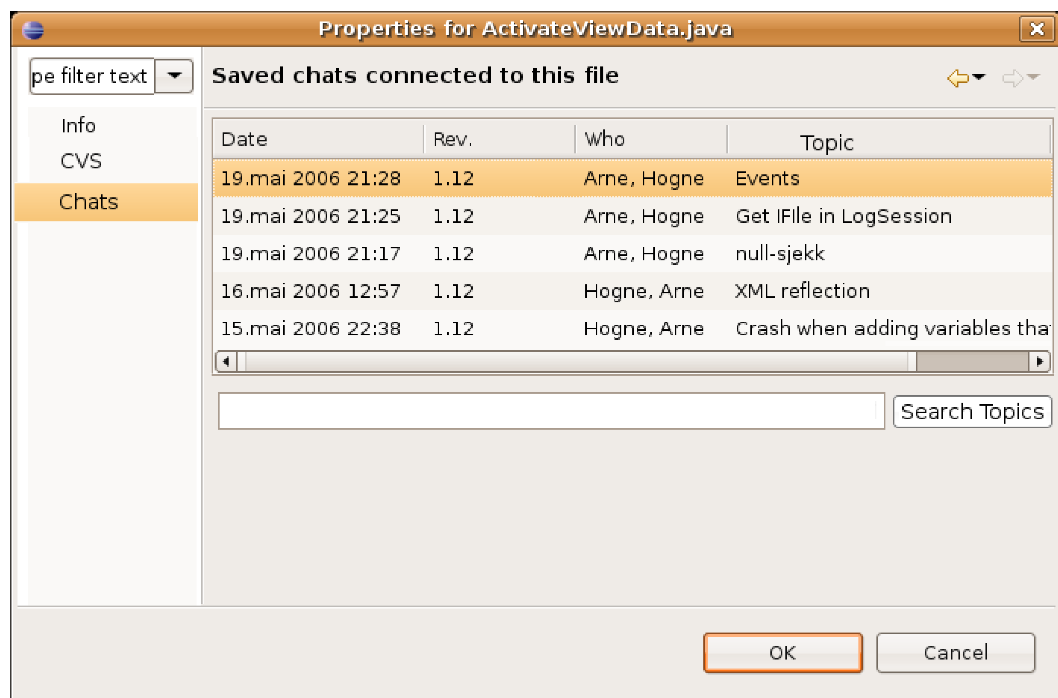
Testingen viste at grensesnittet led under uklar bruk av terminologi, hvor ulike termer ble brukt om de samme tingene. Dette gjorde koblingen mellom de mindre åpenbar enn ønskelig. Et godt eksempel her er "Open Metainformation" menyvalget. Ordet "metainformasjon" ble ikke brukt andre steder og gjorde derfor at det ble uklart hva dette gjaldt. Dette ble endret før siste test med gode resultater, testpersonen fant mye lettere koblingen mellom denne og resten av funksjonaliteten etter en endring til "Open Saved Chats".

Et annet problem med grensesnittet var lite tilbakemelding på de ulike handlingene. Du får ingen umiddelbar tilbakemelding på "Save Chat" og "attach chat", og de tre første forsøkspersonene hadde i starten problemer med å se hva disse gjorde. Siste testperson fant dette mer naturlig som kan tyde på at de endringene som ble gjort for å bedre terminologiene og synliggjøre elementer som [Chatlog] hadde en positiv virkning også i denne sammenhengen. For å ytterligere bedre på denne kan det benyttes informasjonsdialoger som beskriver hva som skjer, og som man kan velge å ikke vise ved en senere anledning.

Kontaktlisten var en grensesnittkomponent som testpersonene hadde en del utfordringer med. Både prosjekt-merkelappen og kontekstene var uklar for flere av testpersonene, spesielt i starten. Funksjonaliteten rundt prosjekt-merkelappen var før testene i stor grad blitt fjernet på grunn av at den ga problemer for attach-funksjonaliteten. Dette gjorde den mindre hensiktsmessig enn den i utgangspunktet var ment å være. Jeg hadde selv testen den fjernede funksjonaliteten tidligere og fant den heller lite intuitiv og tungvint å bruke for å koble prosjektnavn fra faktiske prosjekt i Eclipse til denne prosjekt-merkelappen. I tillegg ble det gitt tilbakemeldinger om at det føltes unaturlig at den lå under hver kontakt i stedet for at man kunne sortere kontakter under ulike prosjekt. Sett samlet vil jeg derfor si at det nok hadde vært bedre å fjerne denne fullstendig og heller benytte en mer vanlig inndeling sett i andre lynmelding-klienter, hvor man kan dele kontakter inn i grupper. Man kan i Simple Chat se på toppnoden "Contacts" som en av disse gruppene.

"Default" og "Context" var som nevnt også gjengangere blant de termene som spesielt i starten var uklar for testpersonene. Også her kan det være nyttig å forsøke å i større grad benytte termer som gir assosiasjoner til lignende funksjonalitet i andre kommunikasjonsverktøy. "Topic" kan for eksempel være et enklere ord å forholde seg til enn "Context" i og med at dette er i utstrakt

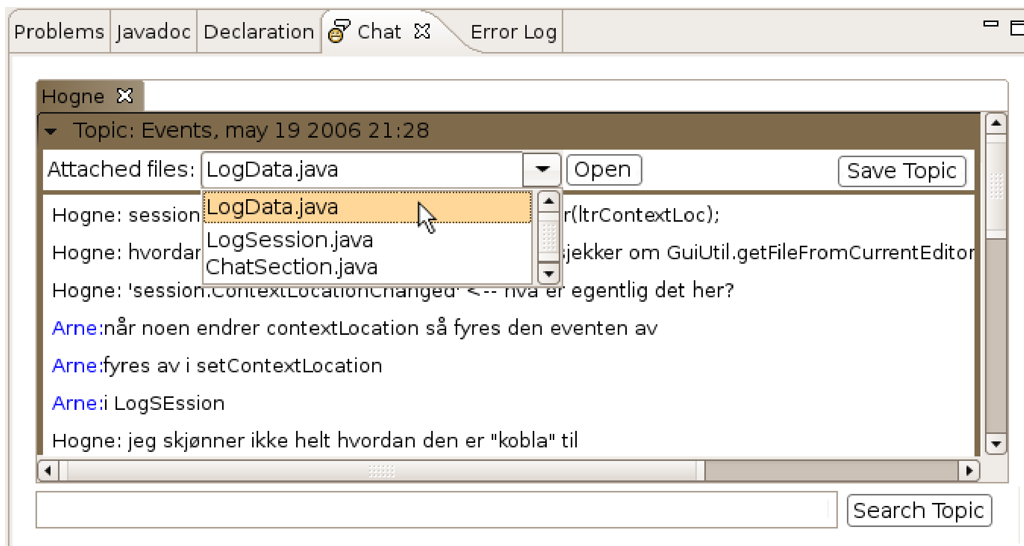
bruk i lignende elektroniske kommunikasjonsverktøy. I tillegg kan standard-konteksten “Default” få et verktøystips som forklarer hva dette er.



Figur 11.1: Ny dialog for lagrede chats under properties

Diskusjonsdialogen i den nåværende implementasjonen ligger tilgjengelig via et eget menyvalg. En mer naturlig plassering hvis man skal følge måten filinformasjon blir vist i Eclipse ville vært å bruke “Properties”-dialogen for en fil (se figur 11.1). Dette medfører også at man kan unngå å duplisere CVS-informasjon om filen siden dette allerede er tilgjengelig under “properties” for en fil.

Samtlige testpersoner etterlyste muligheten for å koble flere filer til samme kontekst. Flere av testpersonene påpekte at de syntes kontekstene tok litt stor plass, så å legge til denne funksjonaliteten må gjøres uten at kontekstoverskriften vil ta betydelig mer plass. For å oppnå dette vil filer som er koblet til bli vist i en rullegardinmeny (se figur 11.2). Ved å klikke ’open’ vil man åpne den valgte filen i editoren. Jeg vurderte også å ha en knapp for å attache en åpnet fil her, men fant det mer hensiktsmessig å legge dette til i selve editoren fordi dette er konsistent med hvordan det gjøres i Package Explorer og enda flere elementer i en kontekst i chat-vinduet ikke er ønskelig.



Figur 11.2: Nytt design av chatvindu

Søkefunksjonalitet ble etterlyst av en av testpersonene, noe som vil bli hensiktsmessig når informasjonsmengden vokser. I grensesnittet er det spesielt to steder som kan tjene på dette. Første er for *Chats* under *properties* for en fil, og andre er i chat-arkfanen man får opp når man har åpnet en av de lagrede diskusjonene. Det vil her være naturlig å la den erstatte innskrivningsfeltet og *Send* knappen siden denne funksjonaliteten ikke er aktiv i denne sammenhengen (se figur 11.1 og figur 11.2).

Flere av testpersonene ga uttrykk for at de syntes det var noe tungvint å hele tiden måtte passe på hvilken kontekst man var i for å ikke ende opp med å skrive urelaterte ting i en diskusjon man ønsket å lagre. De følte de ønsket å holde støynivået nede på de samtalene som ble lagret. Det er flere måter dette kan utbedres på, men de fleste av disse vil føre til et relativt mye mer komplisert grensesnitt så det er en avveining om funksjonaliteten er viktig nok til å rettferdiggjøre dette. Jeg ser for meg at man kunne lagt til tastesnarveier til å skifte mellom *topics* slik at man effektivt kan skifte mellom disse. I tillegg til disse grensesnittendringene kreves det en rekke logiske endringer i programmet for å støtte disse. Disse vil bli beskrevet nærmere i neste delkapittel.

11.3.2 Logiske endringer

Det viste seg at samtlige testpersoner forventet at man kunne koble en eller flere filer til samme kontekst, og flere mente også at det var mer naturlig å tenkte på filer koblet til chats i stedet for chats koblet til filer. Dette er forståelig fordi dette er mer nærliggende hvordan en rekke andre utbredte kommunikasjonsverktøy gjør det. Man er vant til å kunne koble (*attach*) filer til for eksempel epost, men ikke å koble kommunikasjon til en fil andre veien. Det er derfor mer hensiktsmessig å snu denne koblingen for å få utnyttet erfaringen folk har med andre utbredte verktøy. Dette vil tillate en mer intuitiv brukeropplevelse.

I tillegg til at å snu koblingen gjør det mer i samsvar med hva man kan forvente brukerne er vant til vil dette medføre en økt fleksibilitet ved å tillate at man kan koble flere filer til samme kontekst. Dette er ønskelig fordi man ofte diskuterer tema som kan omfatte flere filer. Det er nærliggende her å også se på hvordan chat-filene blir lagret. I den nåværende implementasjonen blir disse lagret sammen med kodefilene. Disse vil dermed kunne komme i veien for de som sjekker ut prosjektet, men som ikke bruker Eclipse hvor disse filene kan filtreres bort i view'ene. Med en omstrukturering på hvordan chats kobles vil det også være mer naturlig å lagre disse på en annen måte. Alle chats kan lagres i en egen katalog, som også kan bli navngitt for å skjules. På Unix-plattformer gjøres dette ved å benytte "dotfiles" som beskrevet tidligere. På denne måten unngår man at disse filene kommer i veien for de som ikke brukes Eclipse som er viktig for at dette skal være en akseptabel løsning for et åpen kildekode-prosjekt hvor deltagerne vil bruke et stort utvalg av ulike verktøy.

En annen problemstilling er potensielle konflikter ved innsjekking av chat-filene. Det er ønskelig at filene skal være mest mulig transparent for brukeren slik at han slipper å få ekstra administrasjonsoppgaver med å håndtere disse. Det vil derfor være hensiktsmessig å i størst mulig grad forsøke å løse mulige konflikter automatisk. Endringer i disse filene medfører å legge til nye sessions eller erstatning av en eksisterende session. Sessions har en ID som blir generert når en session blir laget, dette medfører at bare vedkommende som opprettet konteksten vil ha mulighet til å erstatte den. Det medfører at det med mindre flere personer skulle få generert samme ID (som er teoretisk mulig, men lite sannsynlig), så skal det være mulig å unngå konflikter ved innsjekking.

Ved å snu koblingen kreves det en endring i hvordan man henter ut chats som referer til en fil. Oppslaget må endres til en "reverse look-up" hvor man sjekker

hvilke chats som referer til filen og henter ut disse. Dette kan gjøres ved å legge til merkelapper for filer som blir referert til under session i XML'en:

```
<files>
    <file>LogData.java</file>
    <file>LogSession.java</file>
    <file>ChatSection.java</file>
</files>
```

Et slik oppslag kan bli tungt ved mange logger og filer involvert, men dette kan løses ved å bygge en tabell som mapper filer til logger slik at man unngår å analysere XML-filene om og om igjen.

I større prosjekt hvor mengden lagrede samtaler vil bli stor er det også ønskelig med søkefunksjonalitet. I grensesnittet ble søkefunksjonalitet lagt til på to plasser. Under *properties* er de lagrede diskusjonene enda ikke laster fra XML-filene og man er dermed nødt til å foreta et filsøk. For å vedlikeholde en akseptabel ytelse må filene indekseres og søket vil dermed begrenses til å treffe på de ordene som er i denne indeksen. For søk i åpnede diskusjoner kan man foreta et uttømmende søk i den aktive *topic* ved å iterere gjennom arraylisten. Hvis mer sofistikert søkefunksjonalitet blir ønskelig vil det være naturlig å se etter eksterne bibliotek som tilbyr dette.

11.4 Diskusjon

Testingen viste at så lenge de stegene man måtte utføre for å lagre og hente ut diskusjoner ikke ble for kompliserte og tidkrevende var det ingen av testpersonene som reagerte på den ekstra mengden arbeid dette involverte. Det var heller at flere av testpersonene forventet mer tilbakemelding og dermed flere steg i disse operasjonene. Det er farlig å generalisere dette for mye, men jeg tror man kan si at om barrièren er lav nok vil mange godta noen ekstra steg som de ikke nødvendigvis vil dra nytte av umiddelbart selv. Dette er viktig for å få bygd opp en kunnskapsbase rundt koden. Å søke etter informasjon når man selv lurer på noe er det de færreste som har problemer med siden de får et potensielt umiddelbart utbytte av det. Det som derimot er en større problemstilling her er at terskelen for å lagre informasjonen må være lav nok. Dette har vært hovedgrunnen for å ha minst mulig som kommer i veien når man skal utføre denne operasjonen. Lagring av en logg innebærer at man setter en *topic* og kobler til en eller flere filer og trykker på *save topic*-knappen. Dette er ikke uforholdsmessig mange operasjoner,

men det hadde vært ønskelig å kunne gjøre dette enda enklere som var tanken bak en del funksjonalitet som ikke ble ferdig i tide før testingen. Dette involverte blant annet at man kunne koble til den filen man hadde som aktiv i editoren både via kontekstmeny og tastatursnarvei. Selve kommunikasjon vil bli gjennomført siden den er en faktisk problemløsningssituasjon mellom to eller flere personer i en gitt situasjon. Det er dermed disse ekstra stegene som må være enkle nok til å bli naturlig å gjøre i denne sammenhengen. Et poeng her har også vært å forsøke å gjøre de koblingene som er nødvendig for å ta vare på informasjonen nyttig også i selve samtalen ved at den er med på å sette konteksten rundt en samtale slik at man er sikrere på at man har en felles forståelse av problemet, samt som en organisatorisk hjelp om man er involvert i mange samtidige diskusjoner.

Når det gjelder lagring av diskusjoner kom det frem av testene at det kunne vært ønskelig med mer styring over hvilke deler av en samtale under et emne som blir lagret. Som sagt tidligere er det ønskelig å holde dette enklest mulig for at grensesnittet ikke skal bli unødvendig komplisert, men det er også en annen problemstilling her som jeg mener er viktig å påpeke. Som beskrevet i kapittel 4 om *re-experience* så er det et poeng å lagre kommunikasjon i den formen den utviklet seg fordi dette lar leseren følge hele diskusjonen og bedre kunne reflektere over denne [16, s. 13]. Informasjon som ikke trenger å virke relevant for de som deltar i diskusjonen som skal lagres kan være til hjelp for andre som skal sette seg inn i diskusjonen i ettertid. Dette er også viktig for at man skal kunne sette seg inn i problemstillingen og ikke bare løsningen. Hvis bare løsningen ble lagret ville man få lagret informasjonen mer kompakt, men man ville vanskeliggjøre effektiv refleksjon og læring.

Kapittel 12

Undersøkelse om verktøyintegrasjon

I tillegg til å implementere deler av funksjonaliteten og teste denne kontaktet jeg en rekke åpen kildekodeutviklere i de to prosjektene jeg er aktiv deltager i en forespørsel om å svare på noen spørsmål over epost. Spørsmålene var generelle spørsmål rundt hvilke verktøy de brukte når de utviklet programvare og spørsmål rundt verktøyintegrasjon (Se vedlegg, kap. 14). I dette kapittelet vil jeg bruke dette som en bakgrunn for å evaluere designet som en helhet og forsøke å samle trådene rundt det jeg har sett på frem til nå. Det skal nevnes at denne undersøkelsen også kunne vært gjennomført i forkant av forslaget og designet, men jeg valgte å ta det i etterkant for å evaluere resultatene med fokus på gjennomførbarhet og videre arbeid.

12.1 Diskusjon

En gjenganger blant svarene er at fleksibilitet og åpenhet er kritisk for verktøyene. Verktøy må ikke "komme i veien" og flere hadde opplevd at IDE'er ofte gjorde dette ved at de ble påtvunget bestemte arbeidsmåter (*workflow*). Fleksibilitet her innebærer en rekke aspekter. IDE'er avhenger ofte av prosjektfiler som benyttes for kompilering av prosjektet. Dette bør ikke være påkrevd for å et åpen kildekode-prosjekt, man må kunne bygge prosjektet ved hjelp av standard kommandolinjeverktøy (./configure;make;make install). Må derfor legge til rette for samarbeid mellom personer som bruker ulike verktøy og kommandolinjeverktøy virker her som det mest fornuftige som en fellesnevner. Eclipse kan etter min mening i stor grad imøtekomme dette

fordi den fokusere på presentasjonintegrasjon, samt gir muligheten til en viss grad av valgfri dataintegrasjon uten å kompromittere de underliggende verktøyene med tanke på fleksibilitet og uavhengighet. Flere av de spurte utviklerne fant også presentasjonsintegrasjon som den mest hensiktsmessige integrasjonstypen i denne sammenhengen. Dette er i samsvar med det jeg kom frem til når jeg evaluerte hvilke typer integrasjon som var hensiktsmessig. I denne sammenhengen er det også interessant å se at svarene i større grad ser ut til å samsvare med en utvikler sitt ståsted i forhold til integrasjon enn en bruker sitt ståsted, gjennom at de er mest opptatt av at verktøy ikke bindes fast i for sterk data- eller kontroll-integrasjon enn at de fokuserer på hvilket grensesnitt som vil bli presentert for brukeren.

Når det gjelder funksjonaliteten jeg har implementert som et Eclipse programvaretillegg er den opplagt begrenset til å bare fungere i Eclipse og ikke som et uavhengig kommandolinjeverktøy. Dataene er imidlertid lagret som XML som er et utbredt klartekst format. Dette gjør gjør at det er velegnet til å prosesseres av eksterne verktøy som kan ha sitt eget grensesnitt opp mot disse filene som passer inn i det verktøyet. Videre er det mulig å lage programvaretillegg som fungerer som kommandolinjeverktøy med Eclipse, og det vil nok være hensiktsmessig å dele programvaretillegget i to deler: brukergrensesnitt og *backend*, hvor *backend* blir endret til å også kunne kjøre som et kommandolinjeverktøy og dermed kan metodene for prosessering av XML-filene bli gjort tilgjengelig også her i tråd med denne funksjonaliteten som en fellesnevner for verktøykommunikasjon.

Nesten samtlige av de spurte så liten nytte i å integrere synkrone kommunikasjonsverktøy i en IDE. Dette kan umiddelbart virke som en invalidering av nytteverdien til det jeg har forsøkt å implementere, men det er ikke nødvendigvis slik når man ser nærmere på det. Spørsmålet var av en generell type og det er naturlig å anta at de la eksisterende kommunikasjonsverktøy de selv kjenner til, til grunn når de vurderte dette spørsmålet. Å bare integrere ett av disse, som for eksempel IRC i en IDE uten noen videre utnyttelse av koblingen gir helt klart ikke stor gevinst. Derimot om man utnytter denne koblingen til å skape nye muligheter vil man kunne finne det mer hensiktsmessig. Dette kan gå på ting som SimpleChat hadde fra før som kobling til elementer i koden samt funksjonalitet som jeg beskrev i designet mitt med blant annet deling av informasjon for miljøet man kompilerte for feilsøk. Det viktigste poenget her er likevell knyttet til den funksjonaliteten jeg implementerte. Formålet er å lagre kunnskap om koden på en måte som gjør den lett tilgjengelig i den konteksten den er mest nyttig, nemlig knyttet til selve kodefilene. På denne måten transformerer man synkron kommunikasjon som i seg selv ikke nødvendigvis er så hensiktsmessig å

knytte til et IDE over til lagret asynkron informasjon som utvikleren kan lese og reflektere over. Mange av de spurte svarte at de ofte leste logger fra asynkrone kommunikasjonsmidler som IRC, så dette er allerede tilstede i en viss grad. Disse loggene er derimot ikke knyttet til noe spesielt kontekst og er en blanding av en mengde samtidige samtaler og mye "støy" som gjør den tidkrevende å søke igjennom for å finne relevant informasjon.

Eclipse kommer med en utmerket Java-omgivelse, som tilbyr en kraftig editor, debugger, refactoring-verktøy, Ant-integrasjon ,m.m. Det er derimot en rekke andre populære språk som også må støttes på samme nivå for å være interessant for åpen kildekode-utvikling. Jeg har sett på C/C++-omgivelsen som er tilgjengelig til Eclipse og det viste seg at denne ikke var på samme nivå kvalitetsmessig i skrivende stund.

Flere av de spurte hadde som ankepunkt mot IDE'er at de har grensesnitt som ofte inneholder en rekke elementer som stjeler plass fra selve editoren. Dette er ugunstig siden man ønsker å ha mest mulig plass til selve koden, og minst mulig til andre forstyrrende elementer. For Eclipse sin del kunne man fasilitert dette med å å ha et et eget "perspective" for SimpleChat sine komponenter. Man kan imidlertid enkelt maksimere editorvinduet i Eclipse som i stor grad bør kunne imøtekomme dette. Videre kom det frem at funksjonaliteten editoren tilbyr er kritisk, de fleste av de spurte som ikke benyttet en IDE foretrakk enten Emacs eller Vim og det var uaktuelt for disse å bytte med mindre de kunne få en like kraftig editor i IDE'en også. Det er derfor kritisk å kunne tilby samme funksjonaliteten for editordelen av IDE som disse kjente og kraftige editorene tilbyr for åpen kildekode-utviklere. Eclipse imøtekommer ikke dette i skrivende stund, men den har via programvaretilleggfunksjonaliteten potensialet til å kunne tilby en meget kraftig og funksjonsrik editor for flere språk en det den gjør i dag.

Dette viser at videre arbeid også i stor grad avhenger av andre faktorer i Eclipse som må tilfredsstilles før bruk av Eclipse programvaretillegget blir aktuell. Utviklingen av en mengde programvaretillegg for Eclipse er under arbeid så selv om mange ikke er helt på høyde enda for alle aktuelle programmeringsspråk vil jeg likevell si at Eclipse har potensialet til å være høyst aktuell som vist gjennom omgivelsens støtte for Java-utvikling som er av meget høy kvalitet, samt dens fokus på presentasjonsintegrasjon som i stor grad legger til rette for den nødvendige fleksibiliteten.

Kapittel 13

Konklusjon

I denne oppgaven har jeg sett på åpen kildekode-utvikling og verktøyintegrasjon. Ved å ta utgangspunkt i integrasjonstyper identifisert for tradisjonell utvikling har jeg sett på i hvilken grad disse er hensiktsmessig for åpen kildekode-utvikling.

Jeg har sett på hvordan åpen kildekode-utvikling på flere grunnleggende områder skiller seg fra tradisjonell utvikling. Dette gir føringer for hvilke typer integrasjon som er hensiktsmessig. Verktøyene som er i utstrakt bruk tilfredsstiller en rekke kritiske krav for denne utviklingsprosessen. Disse kravene kommer både fra utviklerne samt ligger implisitt i egenskapene til verktøyene ved at de legger til rette for at kunnskap blir fanget opp og distribuert på en effektiv måte.

Åpen kildekode-utviklere setter fleksibilitet, plattformuavhengighet og åpenhet som meget viktige krav for verktøy de benytter i utviklingen. For å oppnå dette har kommandolinjeverktøy blitt en felles grunnlinje for verktøysamvirke. Dette tilfredsstiller også kravet om at man lett skal kunne bytte ut et verktøy med et annet. Med disse kravene var det presentasjonsintegrasjon som var den typen integrasjon som pekte seg ut som mest hensiktsmessig. De andre typene integrasjon ville vært adskillig vanskeligere å forene med disse kravene og jeg valgte derfor å ikke forfølge disse videre i samme grad, selv om det ikke er utelukket at man også her kan kunne finne interessante problemstillinger som kan forfølges. Med dette utgangspunktet evaluerte jeg et utvalg rammeverk og fant Eclipse til å være det mest hensiktsmessige av disse. Eclipse fokuserer på presentasjonsintegrasjon, og kan fungere som en paraply over en rekke verktøy som har en ulik grad av integrasjon med Eclipse-rammeverket. Dette omfatter alt fra kommandolinjeverktøy til Eclipse programvaretillegg (*plugins*). Dette gjør Eclipse i stand til å bygge på

den eksisterende basen av verktøy som er i utstrakt bruk i åpen kildekode-utvikling. Videre er Eclipse bygd opp av programvaretillegg som kan byttes ut og er dermed i stand til å tilfredsstille den påkrevde fleksibiliteten.

Det er et utall tilgjengelige verktøy som kan være interessant å integrere i et rammeverk som Eclipse, og det finnes allerede et stort antall programvaretillegg av alle typer til Eclipse. Det viser seg at åpen kildekode-utviklingsprosessen ved hjelp av asynkrone verktøy er i stand til å kodifisere store mengder informasjon som blir til kunnskap for deltagerne gjennom "re-experience". Asynkrone kommunikasjonsverktøy legger i utstrakt grad til rette for dette og blir i stor grad brukt til refleksjon og læring av den tilgjengelige kunnskapen, mens de synkrone verktøyene blir sett på som mindre egnet. Et fellestrekk for de eksisterende kommunikasjonsverktøyene i utstrakt bruk er at de i liten grad integrerer med selve utviklingsomgivelsen. Jeg har sett på hvilke typer som kan ha størst nytte av dette og funnet frem til at synkrone verktøy virker til å være mest hensiktsmessige her. Synkrone verktøy er ofte brukt i direkte problemløsning som er tett knyttet til koden og kan trekke fordeler av ett felles grensesnitt som var fokuset for SimpleChat som jeg har basert min prototype på. Jeg bygde videre på dette ved å se på om en tettere kobling mellom synkrone verktøy og utviklingsmiljøet. Ved å lagre den synkrone kommunikasjonen direkte koblet til koden vil den kunne bidra med kontekstuell informasjon som kan utdype aktuelle problemstillinger. Mye av den synkrone kommunikasjon blir også i dag lagret i logger som er tilgjengelig på nettet, men det er betraktelig mer tidkrevende og vanskelig å lokalisere nyttig informasjon for et gitt problem i disse både fordi man mangler den direkte koblingen og fordi volumet er adskillig større siden alt lagres i stedet for bare det som utviklerne finner interessant å ta vare på i en gitt sammenheng. Testing av prototypen viste at en slik kobling, bare den er fleksibel nok og ved å bruke begreper og mekanismer som er kjent fra andre nærliggende kommunikasjonsverktøy, kan være til hjelp for å sette seg inn i ny kode. Dette gjelder ikke minst bare det å forstå den rent teknisk, men også få en bedre forståelse for valgene som ligger bak en gitt implementasjon.

Et utviklingsrammeverk har altså en stor utfordring med å tilrettelegge for de kravene som åpen kildekode-utvikling stiller. Eclipse har vist seg å ha potensialet til å kunne tilfredsstille disse, selv om ikke alle nødvendigvis er tilfredsstilt i skrivende stund. En mye brukt epost-signatur er "Real programmers don't comment their code. It was hard to write, it should be hard to understand". Dette understreket at både implementering av koden og å forstå den er vanskelige oppgaver hvor kodekommentarer i stor grad er den viktigste kilden til "re-experience" [16]. Integrasjon av kommunikasjonsverktøy kan bidra til å lettere legge til rette for at utviklere

kan finne og utnytte kunnskap rundt kodeforståelse, både gjennom at den er lettere å lokalisere og i at den kan bidra til å i enda større grad kodifisere det som ellers i praksis ville vært skjult kunnskap. Hvis man i tillegg har et grensesnitt som krever liten innsats for å bidra med denne informasjonen vil man kunne bygge opp en kunnskapsbase koblet direkte til koden i et prosjekt.

13.1 Videre arbeid

I denne oppgaven har jeg forsøkt å danne et grunnlag for verktøyintegrasjon i åpen kildekode-utvikling ved å identifisere hvilke typer integrasjon som er hensiktsmessig og ved å designe, implementere og teste funksjonalitet som potensielt kan forbedre prosessene rundt læring, samt identifisert viktige problemstillinger som er utfordringer ved denne typen integrasjon. Dette kan benyttes som et grunnlag både for å videreutvikle funksjonaliteten som er beskrevet i designet og videre i prototypen. Den kan også brukes som et grunnlag for relaterte implementasjonsoppgaver innen andre rammeverk og andre typer verktøy for å se hvordan de lar seg koble sammen.

Når det gjelder prototypen som ble implementert i denne oppgaven er den av begrenset omfang, og det er en rekke oppgaver som man kan ta tak i her. Av videre arbeid som kan være interessant er videre arbeid på de delene av designet som ikke ble implementert, samt videre utvikling av prototypen basert på resultatene fra brukbarhetstestene og undersøkelsen. Av mer konkrete ting her er muligheten for å splitte programvaretillegget i et kommandolinjeverktøy og et grensesnitt for å legge seg på linje med andre verktøy som er i utstrakt bruk. Dette kunne bidratt til å gjøre funksjonaliteten uavhengig av Eclipse og man kunne sett på muligheten for å integrere det i andre verktøy også.

Videre testing er også nødvendig hvis man vil komme videre i evalueringen av funksjonaliteten til prototypen. Det ville også være hensiktsmessig å forsøke å få verktøyet i bruk i et faktisk prosjekt. På denne måten ville man fått direkte tilbakemeldinger fra hovedmålgruppen. Det vil nok innebære en stor utfordring å få folk til å benytte programvare under utvikling i et pågående prosjekt, men man kunne ta sikte på å gjennomføre mindre omfattende tester innenfor et prosjekt. Til slutt kan et hett tips være å holde et øye med *The Eclipse Communication Framework* og se om man kan dra fordeler av å ved en senere anledning benytte dette for videre utvikling av funksjonaliteten denne oppgaven ser på.

Bibliografi

[1] Aybüke Aurum, Ross Jeffery, Claes Wohlin, and Meliha Handzic. *Managing Software, Engineering Knowledge*. Springer, 2003.

[2] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21:61–72, 1988.

[3] T. Bollinger, R. Nelson, K.M. Self, and S.J. Turnbull. Open-Source Methods: Peering Through the Clutter. *IEEE Software*, July/August:8–11, 1999.

[4] Clayton M. Christensen. *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*. Harvard Business School Press, 1997.

[5] NetBeans Community. Welcome to NetBeans. Lokalisert den 13 Mar 2006, 2006.

[6] M. Conway. How Do Committees Invent. *Datamation*, 14, 2000.

[7] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43:2:371–383, 2004.

[8] Chris DiBona. Open Source and Proprietary Software Development. In *Open Sources 2.0*, pages 21–36. O'Reilly, 2005.

[9] IEEE Standards Collection: Software Engineering. *IEEE Standard 610.12-1990*. IEEE, 1993.

[10] The Eclipse Foundation. Eclipse communication framework. Lokalisert den 30 Mar 2006, 2006.

[11] The Eclipse Foundation. Eclipse.org home. Lokalisert den 13 Mar 2006, 2006.

[12] R.D. Galliers and Frank F. Land. Choosing appropriate Information Systems Research Methodologies. *Communications of the ACM*, 30:11:900–902, 1987.

[13] Georgia State University. *Design Research in the Technology of Information Systems: Truth or Dare*, 2002.

[14] John Grundy, Warwick Mugridge, and John Hosking. Constructing Component-based Software Engineering Environments: Issues and Experiences. *Journal of Information and Software Technology*, 42:117–128, 2000.

[15] Morten T. Hansen, Nitin Nohria, and Thomas Tierney. What's Your Strategy for Managing Knowledge? *Harvard Business Review*, 1999.

[16] Andrea Hemetsberger and Christian Reinhardt. Sharing and Creating Knowledge in Open-Source Communities - The case of KDE. In *Proceedings of the Fifth European Conference on Organizational Knowledge, Learning and Capabilities*, 2004.

[17] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information System Research. *MIS Quarterly*, 28:75–106, 2004.

[18] E. Von Hippel. *The Sources of Information*. Oxford University Press, 1988.

[19] IBM. *Eclipse Platform Technical Overview*, december 2005 (updated for 3.|; originally published july 2001) edition, 2005.

[20] Open Source Initiative. History of the OSI. Lokalisert den 14 Okt 2005, 2005.

[21] Open Source Initiative. Open Source Initiative OSI - Licensing. Lokalisert den 30 Mar 2006, 2006.

[22] Jabber. Jabber Technical Overview. Lokalisert den 25 Apr 2006, 2006.

[23] K. Kautz, K. Thaysen, and M. T. Vendelø. Knowledge creation and IT systems in a small software firm. *OR Insight*, 15:11–17, 2002.

[24] H.K. Klein and M.D. Myers. A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems. *MIS Quarterly*, 23:1:67–93, 1999.

[25] B. Kogut and A. Metiu. Open-Source Software Development and Distributed Innovation. *Oxford Review of Economic Policy*, 17 (2):248–264, 2001.

[26] Giovan Francesco Lanzara and Michèle Morner. Making and sharing knowledge at electronic crossroads: the evolutionary ecology of open source. In *Proceedings of the Fifth European Conference on Organizational Knowledge, Learning and Capabilities*, 2004.

[27] Josh Lerner and Jean Tirole. The simple economics of open source. *NBER WORKING PAPER SERIES*, WP 7600, 2000.

[28] Arne Mathisen and Frode Angell-Petersen. Integration of Instant Messaging in an Integrated Development Environment. Lokalisert den 25 Apr 2006, 2005.

[29] Steve McConnel. Open Source Methodology: Ready for Prime Time? *IEEE Software*, July/August:8–11, 1999.

[30] S. Meyers. Difficulties in Integrating Multiview Development Systems. *IEEE Software*, 8:49–57, 1991.

[31] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11:309–346, 2002.

[32] M. D. Myers. Qualitative Research in Information Systems. Lokalisert den 30 Mar 2006. Orignalt publisert i MISQ Discovery, Juni 1997, June 1997.

[33] James Newkirk. Introduction to Agile Processes and Extreme Programming. In *Proceedings of the 24th International Conference on Software Engineering, International Conference on Software Engineering*, pages 695–696, 2002.

[34] Nathan Newman. The Origins and Future of Open Source Software. *A NetAction White Paper*, 1999.

[35] Ikujiro Nonaka, P. Reinmoeller, and D. Senoo. *Integrated IT Systems to Capitalize on Marked Knowledge*. I: T. Nishiguchi (red): Knowledge Creation: A Source of Value. Macmillian Press, 2000.

[36] Ikujiro Nonaka and Hirotaka Takeuchi. *A Theory of the Firm's Knowledge-Creation Dynamics.* I: A.D. Chandler & P. Hagstrøm & Ø. Sølvell (red.): The Dymanic Firm. The role of technology, strategy, organization and regions. Oxford Univ. Press, 1998.

[37] Tim O'Reilly. The Open Source Paradigm Shift. In *Open Sources 2.0*, pages 253–271. O'Reilly, 2005.

[38] J. Preece, Y. Rogers, and H. Sharp. *Interaction design: beyond human-computer interaction.* John Wiley & Sons, 2002.

[39] Roger S. Pressman. *Software Engineering. A Practitioner's Approach.* The McGraw-Hill Companies, Inc., 1997.

[40] E.S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open-source from an Accidental Revolutionary.* O'Reilly and Associates, 2001.

[41] Walt Scacchi. Understanding the Requirements for Developing Open Source Software Systems. *IEE Proceedings–Software*, 149:24–39, 2002.

[42] George Siemens. Free and Open Source Movements: Part 1 - History and Philosophies. Lokalisert den 14 Okt 2005, 2003.

[43] H.A. Simon. *The Sciences of the Artificial.* MIT Press, 1996.

[44] Ian Sommerville. *Software Engineering.* Addison-Wesley, 6th edition edition, 2001.

[45] Richard Stallman. EMACS: The Extensible, Customizable Display Editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–160, 1981. Lokalisert den 20 Jan 2006.

[46] Richard Stallman. FSF - The Free Software Definition. Lokalisert den 30 Mar 2006, December 2005.

[47] Richard M. Stallman. GNU Emacs. Lokalisert den 13 Mar 2006, 2006.

[48] Ian Thomas and Brian A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, 09:2:29–35, 1992.

[49] Vijay Vaishnavi and Bill Kuechler. Design Research in Information Systems. Lokalisert den 25 Apr 2006, 2006.

[50] Hans Van Vliet. *Software Engineering - Principles and Practice.* John Wiley & Sons, LTD, second edition edition, 2000.

[51] Eric von Hippel and Georg von Krogh. Open Source Software and the 'Private-Collective' Innovation Model: Issues for Organization Science. *Organization Science*, 14:209–223, 2003.

[52] G. Walsham. *Interpreting Information Systems in Organizations*. Wiley, 1993.

[53] Anthony. I. Wasserman. Tool Integration in Software Engineering Environments. In *Proceedings of the international workshop on environments on Software engineering environments*, pages 137–149. Springer-Verlag New York, Inc., 1990.

[54] Steven Weber. *The Success of Open Source*. Harvard University Press, 2004.

[55] Wikipedia. Open-source software: Participants in OSS development projects. Lokalisert den 14 Okt 2005, 2005.

[56] Wikipedia. Emacs. Lokalisert den 15 Mar 2006, 2006.

[57] Wikipedia. NetBeans. Lokalisert den 15 Mar 2006, 2006.

[58] Yun Yang and Jun Han. Classification of an Experimentation on Tool Interfacing in Software Development Environments. In *Proceedings of 1996 Asia-Pacific Software Engineering Conference*, pages 56–65, 1996.

Kapittel 14

Vedlegg

Testlogger

Testpersonene ble bedt om å snakke høyt om hva han gjorde, tenkte og forventet underveis. Tilbakemeldingene ble nedskrevet og gjengitt her.

Test 1, 16 mai 2006

1 Person ble testa.

Ble forklart at Eclipse plugin som skulle testes var en jabber-klient med en del ekstra funksjonalitet for samarbeid. Ingen spesifikk forklaring ble gitt på de ulike elementene SimpleChat består av.

Innledende spørsmål

Kjennskap til Eclipse og Java: meget god

Kjennskap til Open Source-utvikling: meget god

Oppgave 1

Kjenner igjen kontaktliste fra MSN.

Skjønner ikke hva “context” er for noe.

Prøver å skrive i chat-vinduet og den fungerer som forventet: en “vanlig chat”.

Ser “Testprosjekt”-merkelappen i Chat Roster. Forventer at dette har med delt utvikling å gjøre, delt kode. Forventer at det skal korrespondere med et “Testprosjekt” i Navigator-vinduet, noe han ikke finner. Tror dette er et faktisk “Eclipse project”.

Åpner en javafil, forventer funksjonalitet for felles redigering.

La merke til $me. Tror det betyr at han redigerer fila og mulig synkronisering med CVS.

Ble oppmerksom på [Chat]. Åpner java-fila og ser i den uten å finne noe som virker tilknytta tag’en.

Usikker på “Context”. Tror den har med synkron redigering å gjøre.

Brukte litt tid før han han så tooltip. Mente at dette ikke var noe man forventer på en link. Forventa heller en dialog.

Syntes tooltip hadde uklar tekst. Skjønte hvor det var snakk om. Prøvde først å åpne en fil og se der før han prøvde i se i Navigator som var hvor det var ment å henvise til.

Fant “SimpleChat”-meny i filen åpnet i editoren. Fikk kobla til en fil til en chat, uten å helt skjønne hva som skjedde. Hva ble kobla til hva?

Prøvde a lagre en chat. Forstod ikke med en gang hvordan det fungerte. Lette litt rundt og fant og åpna MEtaInformasjonsdialogen. Forstod da hvordan koblinga fungerte: kobler spesifikke chats (kontekster) til ressurser.

Lagde en ny context, fant høyreklikkmeny med en gang. Åpnet den ved å dobbeltklikke på den. Klikka litt rundt mellom kontekstene. Forventet at chatten skulle hatt filer koblet til seg, ikke omvendt. Man burde kunne legge til (flere) ressurser i chatten i stedet.

Oppgave 2

Åpnet fila LogSession.java

Så at fila hadde diskusjoner knyttet til seg. Klikka seg inn på metainformasjonsdialogen og åpnet diskusjonene som var knyttet til filen og leste disse.

Navigerer litt rundt i LogSession og legger til sessionID variabel og get- og set-metoder. Kjente igjen XML-refleksjon som nevnt i ene diskusjonen og implementerte det uten problemer.

Tilbakemeldinger

Ga en rask forklaring av de ulike elementene og hvordan de var tiltenkt i bruk. Avsluttet så med en runde med tilbakemeldinger fra testpersonen.

Bør støtte mange-til-mange kommunikasjon.

Chat har et tema, bør kunne koble flere filer til samme tema. Altså andre veien enn hvordan de kobles nå. Attached gir også assosiasjoner til dette. Intuitivt forventer man at man "attacher" filer til feks en epost, ikke andre veien.

Legg metainformasjonsdialogen under "properties" for en fil, hører mer hjemme der.

Omvendt kobling: en eller flere filer kan refereres fra en chat. Filene kan ha en reverse look-up for hvilke chat'er som refererer en gitt fil.

Test 2, 18 mai 2006

1 Person ble testa.

Ble forklart at Eclipse plugin som skulle testes var en jabber-klient med en del ekstra funksjonalitet for samarbeid. Fikk litt mer detaljert informasjon en personen i test 1: at "chats" kunne kobles til ressurser uten at dette ble spesifisert nærmere. Ingen spesifikk forklaring ble gitt på de ulike elementene SimpleChat består av.

Innledende spørsmål

Kjennskap til Eclipse og Java: noe kjent

Kjennskap til Open Source-utvikling: litt kjent

Oppgave 1

Kjenner igjen "disconnect", "add contact" og chattefeltet fra andre instant message-klienter.

Usikker på hva konteksten "default" er: session?, chat-vindu?, kanal?

Leste pop-up for attach discussion-lenke. Fant SimpleChat-menyen i høyreklikkmenyen for java-filer i Package Explorer. Prøvde å koble til en fil. Lette

litt etter hva dette gjorde og fant etter en stund at dette hadde endret linken til å vise at filen var "attached".

Prøvde å attache en ny fil, fant at den erstatta den gamle i stedet for å legge til som var forventa.

Lukka filen som var oppe og åpnet den igjen ved å klikke på linken i konteksten som var knytta til samme fil.

Prøvde "save discussion", forventet mer feedback. Skjønnte ikke først hva som hadde skjedd. Forventet at det skulle komme en dialog for hvor man skulle lagre loggen. Ser ikke koblingen mellom "attached file" og "save discussion".

Ser etter en mulighet for å fjerne attached file, finner ingen (funksjonaliteten er ikke implementert).

Usikker på hva "attached file" er:
Gir den tilgang for andre til å se på?
Er attached file ei fil for hele chatloggen?

Testprosjekt: er dette en kanal?

Høyreklikker i etter å ha utforsket den en stund. Brukte litt tid på å finne ut at det fantes kontekstmenyer. Lager en ny kontekst. Tenker på kontekster som filer.

Synes kontekstene (header) tar litt stor plass, blir litt rotete.

Kan man markere og sende til chat?

Merka seg at dato for kontekst ikke dukker opp med en gang for kontekster (ser ut til å være en bug i SimpleChat).

La aldri merke til [Chat] som skal indikere at en fil har en chat knyttet til seg. Måtte gi et hint. Trodde først at [Chat] betydde at den file er satt i en kontekst i chatten.

Legger merke til $me, usikker på hva den viser til.

Fant metainformasjonsdialogen, åpnet den for en fil som har [Chat]. Prøvde å åpne en chat i denne dialogen. Forstår at det er chats som er lagra til gitte filer. Skjønner nå hva attach file er i denne sammenhengen og hva "save discussion" gjør.

Oppgave 2

Usikker på kodingen, har ingen kjennskap til koding med XML. Lette en del rundt i koden. Kikka etter en stund i diskusjonene knytta til filen og fant en relatert til XML. Leste denne og kom frem til at man antageligvis kunne lage en variabel og denne da ville komme i XML'en ved serialisering.

Tilbakemeldinger

Ga en rask forklaring av de ulike elementene og hvordan de var tiltenkt i bruk. Avsluttet så med en runde med tilbakemeldinger fra testpersonen.

[Chat] virker som en indikator på noe som blir utført akkurat nå. For eksempel [Chatlog] hadde vært et bedre ord å benytte. Bør også vises bedre, for eksempel ved å benytte farge. Mange andre elementer (decorators) skaper mye støy.

Koblingen mellom "attached file" og "save discussion" kan være noe uklar, lite feedback på utførte actions i grensesnittet knyttet til disse.

Bør kunne lukke kontekster.

Litt tungvint å måtte passe på å starte en ny kontekst når man vil skifte tema og unngå at man lagrer uønskede deler av en samtale til en fil. Kan også løses ved å ha for eksempel clear for en kontekst eller lignende.

Test 3, 18 mai 2006

1 Person ble testa.

Ble forklart at Eclipse plugin som skulle testes var en jabber-klient med en del ekstra funksjonalitet for samarbeid. Fikk litt mer detaljert informasjon en personen i test 1: at "chats" kunne kobles til ressurser uten at dette ble spesifisert nærmere. Ingen spesifikk forklaring ble gitt på de ulike elementene SimpleChat består av.

Innledende spørsmål

Kjennskap til Eclipse og Java: lite kjent

Kjennskap til Open Source-utvikling: lite kjent

Oppgave 1

Forventet mer funksjonalitet i kontekstmeny i SimpleChat Roster. Forventet at man kunne definere nye prosjekter (ikke implementert) samt at noe skulle skje om man klikker på et - det er bare et element for organisering av kontekster for øyeblikket.

Ble oppmerksom på tooltip for attach file-lenken. Lokaliserte Package Explorer og høyreklikket på en java-fil. Brukte tid på å finne SimpleChat-menyen, kontekstmenyen i Eclipse har veldig mange elementer.

Forventer at "attach" bare kobler til filer, så den burde ikke vises for andre elementer enn filer (bug). Prøvde også "link", forventa at denne også koble til filen på et eller annet vis.

Observerte at hvis man klikker på "not attached"-lenken får man ikke opp tooltip etterpå når man holder muspekeren over. Dette kan medføre at hvis en bruker klikker umiddelbart får vedkommende aldri sett tooltip med mindre han flytter muspekeren bort og så over igjen uten å klikke.

Usikker på hva kontekst er. Bare et navn man setter?

Klikket på metainformasjonsdialogen for en fil som ikke hadde noen chats tilkobla, fikk opp en dialog som sa at ingen chats kobla til. Hva gjør den? Prøvde enda en fil med samme resultat.

Likte ikke at chat'en ikke knekker linjer som er for stor for tekstfeltet.

Prøvde "attach chat" - skjønnte ikke hva som skjedde. Forventet en dialog. Ble chat lagra til en fil?

Fant attach-linken hadde endra seg til "Open attached file" og skjønnte dermed hva "attach file" gjorde. Forventet mer feedback når man attacher.

Forventet også mer feedback på "save discussion". Har den en default location? Hvorfor vil man attache en fil? bare til eget bruk? Ser ikke helt vitsen med å attache, forventer at den blir sendt til den man snakker med. Finner ikke helt hva den gjør.

Ble oppmerksom på $me, skjønner ikke hva den gjør. Markeren den noe man benytter? Strenger gir lite informasjon.

Ble også oppmerksom på [Chatlog], er dette attached file? Leter i selve java-filen uten å finne noe relatert.

Legger merke til at når man lagrer en chat får man opp [Chatlog]. Tror dette indikerer en åpnet kontekst ikke lagret chat. Forventer en dialog for lagring av chat til fil.

Trodde først at context header hørte til tab'en, ikke conteksten. Når han ble oppmerksom på at den hører til en spesifikk kontekst så regner han med at attached file er noe den man kommuniserer med også kan se som attached.

Fant ut at metadialog ga informasjon når filen var merket med [Chatlog]. Måtte ha litt hint om dette fordi hadde ikke brydd seg med å kikke i metainformasjondialogen på flere filer fordi den ikke ga noe nyttig da han testet den tidligere på filer uten [Chatlog]. Mener "metainformasjon" er et dårlig navn, burde hett noe sånt som "Open Saved Chats".

Hvordan kan man legge til ulike filer uten å få med det man allerede har diskutert? Ser ikke sammenhengen mellom det å lage nye kontekster og det å lagre de i forhold til å dele opp en samtale slik at bare det man vil ha med blir lagret.

Laget en ny kontekst, prøvde ikke å åpne den, tenkte ikke på den som et nytt element i chatten. Fikk tips om å dobbeltklikke på den for å åpne. Skjønnte så hvordan kontekst hang sammen og hvordan man kunne dele en chat inn i disse.

Oppgave 2

Åpnet javafilen og kikket rundt i den. Usikker på hva som skulle gjøres. Så at det fantes chats for fila og leste disse. Fant at man sannsynligvis bare kunne legge til en variabel så ble den automatisk en tag.

Tilbakemeldinger

Ga en rask forklaring av de ulike elementene og hvordan de var tiltenkt i bruk. Avsluttet så med en runde med tilbakemeldinger fra testpersonen.

Chat bør ha line breaks

"Context" uklart, bør gjøres mer forståelig.

Bruk farge på [Chatlog] for å gjøre mer synlig.

Mer feedback på handlinger: pop-up ved handlinger som lagre og attach som kan ha en checkbox for å huke av "do not show again". På denne måten forstår man bedre hva som skjer og kan slå den av når man er blitt kjent med funksjonaliteten.

Test 4, 22 mai 2006

1 Person ble testa.

Ble forklart at Eclipse plugin som skulle testes var en jabber-klient med en del ekstra funksjonalitet for samarbeid. Fikk litt mer detaljert informasjon en personen i test 1: at "chats" kunne kobles til ressurser uten at dette ble spesifisert nærmere. Ingen spesifikk forklaring ble gitt på de ulike elementene SimpleChat består av.

Innledende spørsmål

Kjennskap til Eclipse og Java: en del kjent

Kjennskap til Open Source-utvikling: en del kjent

Oppgave 1

Kjente igjen Roster og Chat fra andre "instant messageing"-klienter.

Ser at den har kontakter lagt inn med prosjekter under. Hva er "default"? en chat session?

Prøvde å høyreklikke og fant kontekstmenyen i Roster.

Lurer på om en kontekst er noe som kan kobles til en fil.

Så tooltip på attach-lenken, kikker i Package Explorer. Høyreklikker på en javafil og finner SimpleChat-menyen. Kikker i denne, men klikker ikke på noe.

Laget en ny kontekst, så at denne hadde et annet ikon uten helt å se hvorfor. Åpnet denne med å dobbeltklikke med en gang, fant det naturlig. Har to kontekster oppe, skriver i begge for å se hvordan de oppfører seg.

Lurte litt på forholdet mellom chat-tab og brukere/prosjekter. Er det bare i forhold til bruker eller har også prosjekt en innvirkning her.

Ser i høyreklikkmenyen for en javafil i Package Explorer igjen og attachet en fil, så ut til å forstå sammenhengen. Lurer på om man kan attache flere filer, finner at dette ikke går og at man i stedet erstatter attachet fil i en kontekst.

Lagret en chat og la merke til at [Chatlog] dukket opp i Package Explorer. Ser i høyreklikkmenyen for filen og klikker på "View Saved Chats". Finner igjen chatten han hadde lagret og dobbeltklikker på denne som åpner den. Utrykker at han forstår hvordan koblingen foregår med at man attacher en

fil og lagrer loggen til denne, hvor man så kan finne den igjen under "View Saved Chats".

Prøver til slutt å lage en link og får denne opp i chaten.

Oppgave 2

Åpner javafilen, maksimerer editorvindu. Ser igjennom koden, forsøker å sette seg inn i den.

Usikker på hvordan man skal implementere, kommenterer at man kunne brukt chatten til å spørre om hjelp. Ser ikke etter om det er allerede lagrede chatter til å begynne med. Etter en stund gir jeg et hint om dette. Det kommer da frem at tester hadde tenkt på chatlogs som en mulig kilde for informasjon, men tenkte på dette som "sitt eget system" og trodde dermed at det ikke var andre chatter enn de som han selv hadde lagret. Han hadde sett at det var chats kobla til filen, men tenkte at dette var igjen fra tidligere testing og ikke chats som kunne være nyttig.

Leste igjennom chats og fant at man kunne lage en variabel i klassen som blir reflektert i XML'en.

Tilbakemeldinger

Ga en rask forklaring av de ulike elementene og hvordan de var tiltenkt i bruk. Avsluttet så med en runde med tilbakemeldinger fra testpersonen.

Bør kunne linke til flere filer i en gitt kontekst.

La ikke merke til fargekodingen av filer med [Chatlog], gjør den litt mer markant.

Roster og Chat tar mye Screen Estate, Eclipse har allerede mange vinduer som tar plass.

Kategoriser kontakter i prosjekt i stedet for omvendt. En kontakt skal kunne være i flere prosjekter.

Bør kunne slette en kontekst igjen.

Litt mye klikking for å koble til (default case var tiltenkt men mangler: skal kunne koble til åpnet fil ved å klikke på linken, men dette ble ikke implementert før testing).

Hvis det blir mange diskusjoner knyttet til en fil kunne det være nødvendig med søkefunksjonalitet.

Undersøkelse om verktøyintegrasjon

Intervju av utviklere som både driver med programvareutvikling på frivillig basis som deltagere i Open Source-prosjekt og som også jobber med programvareutvikling i arbeidslivet. Det ble sendt ut epost med spørsmål i to omganger grunnet at jeg ønsket å få svar på noen ekstra spørsmål som jeg fant relevante etter å se svarene fra første omgang. I tillegg til spørsmålene var det en del forklarende tekst for å utfylle spørsmålene. Disse er gjengitt under, men kuttet fra svarene for å unngå unødvendig duplisering av samme teksten. Svarene er satt sammen fra epostkorrespondansen og det kan være manglende svar på noen. Svarene er behandlet anonymt.

Spørsmålstekst

The issue at hand is Open Source development and tool integration. I'm looking at what can be gained from further integration of tools in frequent use in Open Source development in an Integrated Development Environment (IDE). Furthermore I'm looking at what sort of integration is appropriate for this development paradigm. I'm using Eclipse to implement and test my findings, but any IDE can be considered and the questions here are not specific to Eclipse.

Please answer the questions in some depth. If you feel a question doesn't apply to you, just skip it.

Q: What is your occupation?

Q: What Open Source Projects are you actively participating in?

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

Synchronous tools like instant messaging (Jabber, etc), IRC and such are widely used in Open Source development, but they are currently mostly used as standalone applications with no connection to the tools used for development.

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

There are several types of tool integration:

Presentation integration - concerns user interaction/interface Data integration - concerns how the tools use and share data Control integration - concerns tool communication and interoperation Process integration - considers the role of tools in the software process

Pres. int. is for instance like Eclipse where you have a unified and consistent user interface to a lot of different underlying tools.

Data integration is a tighter integration where the tools use for instance a DB to share the data.

Control integration can be done by for instance a defined API for tool interoperability.

Process integration is about the software process itself, where common models for traditional software development are iterative, prototyping, spiral model, sequential etc

So data and control is a tighter and more rigid integration than presentation integration, while process is about having support for the development process itself in the software used in the development.

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

If you are employed in the software industry:

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

Svarene på undersøkelsen

Person A

Q: What is your occupation?

Student / Programmer / IT-administrator. Currently employed 60% as a programmer for a medical imaging company.

Q: What Open Source Projects are you actively participating in?

Freeciv and Warzone Resurrection.

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

For open source programming, I do not use an IDE, as I have yet to find one for Linux that I like and that is lightweight and integrates well into my workflow. I prefer editing source code in nano.

For work, I use Xcode.

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

Subversion and autotools support. Integration with bug tracking software. Few mandatory features. An IDE that does not force users to conform to their idea of a good workflow.

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

Not sure what the question is.

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

Presentation integration is very nice. See for example Trac, which connects Subversion, wiki and bugtracking in a brilliant (and brilliantly simple) way. Data integration is for me rather unimportant. The rest I do not have many opinions on.

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

Interoperability. Portability.

If there must be project files, they and changes to them must be transparent,

keep changesets as small as possible, and in readily readable plain text formats.

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

Yes, occasionally. In the Freeciv project we used, when we were more active developers, to have regular irc meets in which we discussed and planned what to do, and going back to check what was agreed in the irc logs, which we sent to every project maintainer, was quite useful. IRC meets was a very good way to coordinate many far-flung developers with very diverse interests, opinions and schedules. We started doing them in response to the problem of coordinating ourselves when we got so many active maintainers that it got problematic to plan ahead and agree on design issues. Nowadays we are less active maintainers, and so we do not have this problem"anymore, and no more irc meets either.

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

I could use pretty much the same tools for the open source projects as for work, but for the former I can choose which tools to use myself.

Person B

Q: What is your occupation?

Computer programmer.

Q: What Open Source Projects are you actively participating in?

Large: Xen, Linux kernel, Wesnoth Small: ccontrol, module-init-tools

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

No, I use emacs. I need to be able to build code without an IDE (all Free Software should aim for "./configure;make;make install"), so it is simplest to stay close to that basic environment.

Q: What sort of tools appropriate for Open Source development do you feel

are lacking in current IDE's and would you like them to be integrated in an IDE?

There is still a great deal to be done below the IDE (although it could be implemented within an IDE, it would be more useful as a standalone). We're seeing great progress in distributed version control (bzr, mercurial, git), and static and dynamic checking (sparse, valgrind). Integration of these tools is one lack of current IDEs, but they are still developing so they are something of a moving target.

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

Have never used such a tool, so cannot really comment. I often work in a different timezone from my colleagues, too.

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

Presentation integration certainly. The others are less obviously appropriate, although process integration for OSS could be quite interesting.

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

Flexibility and parallel development. Flexibility means integrating with others using other tools and methods as much as possible, and at any stage in the process. Parallel development means avoiding introduction of centralization.

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

Yes, but only for Wesnoth.

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

Yes, frequently.

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

Heh, same projects, so same tools. 8)

Person C

Q: What is your occupation?

Officially, Senior Software Developer". Basically just a programmer. :)

Q: What Open Source Projects are you actively participating in?

Just Battle for Wesnoth, and even that, not so much as I used to.

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

I prefer to use gvim and the command line when developing in Linux. When developing in Windows I use Visual Studio, but this is mostly because that's the easiest way to use Microsoft's compiler.

I am not opposed to the idea of an IDE in theory, but have never found one that I find easier and more powerful to use than gvim with ctags. In addition to Visual Studio, I have used kdevelop several years ago, but at that time I did not think it was mature. I have also played with Eclipse, and concluded it would be rather nice if I wanted to develop in Java, or if they improved C++ support.

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

Well, to be honest, I haven't tried out different IDEs for some time, so I can't say I'm completely sure what features are currently available. So, instead, I'll just list features I'd like to see, and you can exclude any that aren't already available.

The first and most important thing for me is a very flexible editor. Different programmers like very different editors, so the editor for an IDE must be able to act like vim, like emacs, or like a generic Windows-type editor. For me, an IDE that doesn't have support for embedding vim, or have a vim-like editor is a non-starter in Linux. Of course, for many programmers, an IDE that doesn't embed emacs is a non-starter, so for wide acceptance, an IDE must allow any popular editor.

The ability to point out common mistakes (such as mis-spelled variable names) would be a very nice feature. I believe this already exists in Visual Studio for C++, and in Eclipse for Java. I'm not sure if there is anything like this for C++ in any Open Source IDE.

The ability to do common refactorings (such as renaming a variable) would

be nice. (I know this is also available in Eclipse for Java). In C++, having support for doing things like iterating over an STL container would be nice, since writing the code for this is rather tedious, but should be easily automated. Likewise for adding a new function to both a header file and cpp file.

One feature I'd like to see, which I'm not sure of the availability of, is the ability to recognize different types of symbols, and to color-code them accordingly. A class member could be coded differently to a local variable which would be different to a global variable. It is common convention in many languages to use different naming conventions for different types of symbols, but an IDE recognizing this would make things easier.

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

Well, I find it hard to imagine how integrating a 'chat' program into an IDE could be very useful. The only possible way I can think of that there could be any cohesion created is that if someone types the name of, say, a function into the chat program, one might be able to click on the function to look it up in the IDE. But I think that this would only be marginally useful. I can't think of much other functionality that would make the chat program being 'integrated' any more useful than it just being a separate program.

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

I think the most important thing is making things flexible for the programmer. One of the nice things about the Unix command line is the ability to combine tools in many different ways using the simple concept of pipes.

Often I think IDEs try to make things too 'easy', but in doing so lose the power of the command line. I've noticed this in particular with Visual Studio, where often I resort to doing things using Cygwin.

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

The answer to both these questions would be yes for me. I fairly frequently do this.

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

Well, at my workplace we use lots of Open Source tools for development, so the tools are mostly the same. We have gcc, Eclipse, vim, emacs, etc available for development and programmers take their pick of what to use. I use gcc/vim/gdb personally.

Person D

Q: What is your occupation?

software engineer

Q: What Open Source Projects are you actively participating in?

wesnoth

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

I use Vim as my main coding environment. I have tried a couple of IDE (most of them based on visual studio) and always switched baack to vim because most environment focus mainly on debugger integration, which is only a small part of the development process. I need something that helps me navigate my code easily, and VIM's advanced navigation features (in particular search highlighting) is a must

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

see earlier. debugging is a small phase, I need a tool to code, more than debug.

most IDE are usually cluttered by extra useless windows, it's important to maximize source code space, both in width and space (the worst is MS visual studio, where the default editor is a syntax highlighted notepad and the window is cluttered by useless things on all sides including three lines of shortcuts)

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

I don't think "integrating"would be that usefull. We do use IRC a lot, and I usually have IRC on one screen and source code on the other. however, copy/pasting is simple enough, and I don't think it's worse having things like colaborative coding in...

source code highlighting in IRC would be nice but again most IRC discussions are about design, not implementation. and at design stage, IDE are not usefull, an IRC drawing board"could be usefull, or integration into UML tools... but most debugging won't need much more than what copy paste allows us to do

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

no opinion

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

command line backends.

svn is integrated both in eclipse and vim, but in very different ways. each IDE has it's own philosophy and the tool should not force a work model on the IDE (the ide should not force it on the user, but that's another problem) command line is the best way to have an easy way for tools to communicate with the rest of the world, and any tool that implements a GUI can't be integrated easily. tools can provide a separate GUI but if a tool is meant for dev, it should be comand line first

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

no. I do use irclog to get up to date with what's going on on the project regularly, but I don't look for specific information"

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

I don't even know where the ML archives are :)

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

IRC : at work all communication is done by mail, talking and documentation

in shared directory.

the big consequence is that simple, short information is hard to get, because that information is too small to be formalized, and getting 5' of attention from an main developer forces you to ask his manager to schedule some time (usually not less than an hour, usually you have to ask your own manager first)

thus it's faster to reimplement or reverse-engineer. IRC allows us to simply ask the main concerned person and get the answer immediately

Person E

Q: What is your occupation?

I am a freelancer. I worked mostly as a trainer for microsoft office, windows 2000 administration and a couple of programming languages. I also did some programming most of the time, starting with Visual Basic, then moving on to Java and C#. I wrote a few books about programming for Microsoft Press and Franzis. Right now i am involved in a bigger project (actually more than one) at a german bank that lasts almost three years now.

Q: What Open Source Projects are you actively participating in?

Wesnoth. It is my first and only one so far.

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

I use Microsoft Visual C++ 6.0. I prefer to use IDE's because i feel that they increase my overall productivity compared to a bunch of standalone tools.

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

Refactoring integration has just started to be integrated into an IDE (well, regarding microsoft at least). This has the potential of making development more comfortable if you know how to deal with it. That is, if you are familiar enough with refactoring techniques to use them to your advantage.

Version control systems come to my mind, especially svn which we use for Wesnoth. I wish that would be integrated into MSVC++. Probably this is rather a microsoft problem, i am not familiar enough with other open source IDE's.

Regarding object oriented programming, i would also appreciate a lot to have

a tight integration of UML modeling into an IDE. The current solutions don't seem to be all too good with this. I have to admit that i did not work with market leader tools like Together and Rational Rose so far. I touched Together once as a trial version and it did not convince me but it also was a couple of years ago. But still i have the feeling that IDE's focus to support the development of code. I haven't seen a product yet, that supports a whole development process like for example the Rational Unified Process.

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

Hmm, i am kind of biased about this question. Out of my commercial experience, i feel that there are phases where a lot communication is needed (during design) and phases that don't need much communication if any. This however is based on a controlled process. Open Source projects (at least Wesnoth) have participants that are more loosely coupled than in a normal software project. Hence, the need for communication increases. On the other hand, i don't see a immediate benefit of integrating such communication abilities into an IDE. To me this only makes sense if there is a direct relationship to the code, which i don't see here.

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

As i mentioned before, i miss software development tools having the focus on the software development process. However, i would not make a difference between my commercial activity, my personal preferences or open source development in general. I am of the opinion that every kind of development would benefit from that (as well as a tighter integration of other tools). Compare it to using indentation. Developers do it to get a better overview, it helps them to write code and make less mistakes. It is a rule that every development benefits from. The same goes for design rules like where to declare a variable best up to the use of patterns. This happens on a different level but still every development benefits from it. You can enlarge this to software architecture (like for example tier models) and on the end you find the whole software development process. Every rule on every level has the same goal: Get some organization into things. Even if it means more effort here and there developers still use it (sometimes ;-), because they know that they benefit from it in the long turn.

Of course, with Open Source projects the situation is different in that you can't make any preassumptions about people joining the project. You also will have difficulties to get everyone on the same skill level. After all you can't really say: "Well, these are our requirements and here is a list of E-Learning

courses you have to take before you can join the project!".

But i think it would rather be like with a whole lot of other software: There is a huge offer of functionality and 95% of its users use only a fraction of it. As long as you are able to write code without designing a sequence diagram first i don't feel it's going to be a problem at all. So yes, i am all for having useful tools integrated into the IDE (independent if it is used for OS development or not).

Having said that, i am pretty sure, that you need at least data integration for tools to be really useful in an IDE. Not sure about control integration here, but it seems to make sense if tools are provided by different sources".

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

I am not familiar enough really with open source tools to answer that.

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

Yes, i do that if i have enough time or search for a specific information. However, most of the time it is easier to directly ask people :-)

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

Yes, definitely. I mainly use it (in this order) for: - Getting help with error messages
- Getting help for the programming language (on pattern/howto level mostly, not syntax)
- Dealing with IDE/tools bugs
- Configuring the development environment

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

The IDE is (more or less) the same so there is not much left:

Version control: At work we use locks on files (check out, check in) at the moment. We have this possibility for wesnoth but it is not used afaik.

Bug/Feature-tracking is pretty much the same. The tool we use there (OnTime) is focused a little bit more on project managers needs, so there is

effort estimation and reports that give an overall status of the project.

Packaging: This is difficult to say. Although we are not supposed to, developers have the possibilities to directly interact with the production system. Especially in the past this has been used to implement patches. A defined software development process does not exist at the moment (although we are working to establish it) so there are releases, that don't have packages in the sense of this word. That is there is no directory on a server somewhere where you could get the tarball for a certain release and there is no installer as well. Admittedly, for a microsoft web application this is somewhat harder to do and not well supported by the development environment atm.

Documentation: The standards for documentation are higher than for open source, because if people pay money for getting a software they also expect it to be documented somehow. However, this is rather not related to tools because the existing tools don't make for a sufficient documentation in this sense (and probably never will).

Person F

Q: What is your occupation?

I'm a CS student, and I currently have a part-time job as system administrator.

Q: What Open Source Projects are you actively participating in?

I contribute to projects related to my university (not useful at all for outsiders). Also, I still do a few maintenance tasks in Wesnoth, and I'm now trying to contribute more to the Debian project.

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

No. When it comes to software development, I tend to like to be able to "control" everything. However, when I'm using an IDE I feel that this is not possible at all.

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

I usually prefer to use different tools for different things (that's another reason why I don't like IDE's at all).

I have sometimes been forced to use an IDE, and one of the things I have

found more troublesome is precisely the compatibility with other “styles” of development. For instance, it is often necessary to create a new configuration of the project specifically for the IDE. It would certainly make things easier if it was possible to avoid this duplication.

Another thing that I would like to see in IDE’s is better support for Control Version Systems. Yeah, they are mostly supported, but not _fully_ supported: only available as plug-ins, missing features, etc.

(Note that this is based on my own experience with IDE’s, but I certainly don’t know most of them. And I’m not sure if this points are still valid, it might have changed in newer versions.)

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

I simply don’t see any apparent benefit.

(Am I missing something? I think it would be more interesting to integrate non-synchronous communication tools such as bug tracking systems or even wikis.)

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

I think any kind of integration is welcome as long as it is doesn’t try to enforce something more than necessary.

For instance, I wouldn’t like data integration if it forced me to use only a specific database engine. So, in this respect, less tighter integration is probably preferred.

Also, process integration seems like an interesting topic too, even though it is not easy to define an Open Source development model. If I understood it right, it would be correct to say that Control Version Systems are part of the process, so in some ways this has already been done. Further improvements would be welcome.

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

Freedom to choose the tools. In Free Software / Open Source development, people often prefer to use different tools for the same thing (TIMTOWTDI), which is a good thing.

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you

are involved in?

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?

Person G

Q: What is your occupation?

Q: What Open Source Projects are you actively participating in?

Q: Do you use an IDE yourself? if so, which one? if not, why do you prefer a different type of development environment?

Q: What sort of tools appropriate for Open Source development do you feel are lacking in current IDE's and would you like them to be integrated in an IDE?

Q: What benefits, drawbacks and possibilities do you find with integrating synchronous communication tools in an IDE?

Q: What types of integration of external tools in an IDE do you think is appropriate for Open Source Development?

Q: What qualities of the current tools (in widespread use in Open Source Dev.) and the way they interoperate do you think is the most important to preserve in an integration process?

Q: Do you ever use archived logs from synchronous tools (IRC, Jabber, etc) to search for information concerning development issues in the projects you are involved in?

Q: Do you ever use archives from asynchronous tools (Mailing list, News Groups, etc) to search for information concerning development issues in the projects you are involved in?

Q: What are the main differences and similarities in the tools you use at work compared to the tools you use when working on the Open Source Projects you participate in?