INF112 (systemkonstruksjon) våren 2004

Prosjektoppgave 1, revisjon 1

Magne Haveraaen, kursansvarlig
Institutt for informatikk
Universitetet i Bergen

Sammendrag

Den første prosjektoppgaven på kurset inf112[1] går ut på å omstrukturere kode og kvalitetssikre den. Prosjektoppgaven utføres i konkurranse mellom 3 grupper, der den beste løsningen *vinner*, og blir brukt som basis for prosjektoppgave 2.

1 Innledning

Prosjekt 1 på kurset inf112 (systemkonstruksjon) går ut på å vedlikeholde gammel kode. Den koden vi skal arbeide med er for Druantia-systemet. Det ble utviklet av studentene på I122 (systemkonstruksjon) våren 2003, og er basert på programsystemet FMS som ble laget av studenter ved I122 våren 2001 og 2002.

Druantia er et javabasert system som er ment for interaktiv kommunikasjon og kan knyttes til fora som UiB sin studentportal – `http://studentportal.uib.no/` – og dens databaser. Slik Druantia-koden nå er består den av en del, Druantia rammeverk, som vi skal benytte videre, samt en to-tre moduler som benytter rammeverket til kommunikasjon. Disse modulene er utfomet slik at vi ikke ønsker å jobbe videre med dem, men de inneholder en god del støttekomponenter det er verdifullt å trekke med seg videre.

Oppgaven blir dermed å trekke ut rammeverket og så mange komponenter (i hovedsak basisklasser) som mulig fra Druantia, omorganisere og sikre kvaliteten på dette. I neste prosjekt vil vi så bygge ut rammeverket med en del velfungerende tjenestemoduler.

2 Generelle krav til besvarelser på inf112

Dette er en sjekkliste over punkt som skal ivaretas ved jobbing med inf112-prosjektoppgaver. De fleste av disse er temmelig opplagte, men tas likevel med.

- Tidsfrister for delinnleveringer og sluttinnleveringer/presentasjoner skal overholdes.
- Dokumentasjons-, kode- og arkiveringsstandarder skal overholdes, se Druantia plandokument som beskriver og henviser til disse. Oppsummert er dette:
  - Alle kildedokument (inklusiv kildekode) legges i et Subversion-arkiv. Hvert dokument skal inneholde et Subversion-generert versjonsnummer og navn på forfattere/inspektører. Se eget notat om bruk av Subversion. Subversion-arkivet skal ikke inneholde formatert eller kompilert kode.

    Det anbefales å legge så mye som mulig av mellomversjoner inn i Subversion-arkivet selv om disse ikke er en del av en innlevering. Dette gir anledning til kontroll med arbeidsprosessen, og vil i mange tilfelle gjøre det mulig å redde seg ut av de mange hjørner en lett maler seg inn i i løpet av en større programvareutviklingsprosess.

[1] `http://www.ii.uib.no/~magne/inf112v04.html`

- Dokumentasjon skal skrives i latex (se eget notat om dette), med unntak for detaljert kodedokumentasjon (se nedenfor).
- Programvare skal dokumenteres med UML-formalisme (se eget notat om denne), men der detaljerte klassediagram o.l. erstattes med dokumentasjon generert av javadoc. De nyttigste UML-diagrammene for oss er bruksmønster-, sekvens- og tilstandsdiagram for systemoversikt, eventuelt komponentdiagram for oversikt over sammenhengen mellom enkeltkomponenter.
- Programvaren skal organiseres etter *kontraktsbasert programmeringstankegang* (se neste seksjon).
- Programkode skal skrives i java i henhold JCC-standarden (*java code convention*, se henvisning i Druantia plandokument).
- For alle dokument som skal prosesseres før de kan brukes (f.eks. formatering med latex, produseres med javadoc, kompilering med javac) skal det lages en Ant-beskrivelse som genererer koden. Se eget notat om Ant. Slik Ant-kode må også dokumenteres, spesielt konfigurasjonsstyringen som utgjør et samspill mellom Ant, søkestier og kompilatorer/formaterere.

Videre skal ethvert dokument (kildekode, Ant-beskrivelse etc.):

- ha
  - navn på forfatter og dato for ferdigstilling (`@author`-felt for javadoc)
  - eventuelt navn og dato for andre som har vært inne og rettet på dokumentet
  - navn på alle som var involvert i inspeksjon av dokumentet, hvilke roller de hadde, dato for når inspeksjonen ble utført, og om det kom frem vesentlige merknader
  - et automatisk gjenkjennbart Subversion-versjonsnummer (`@version`-felt for javadoc), kortfattet informasjon om hvorfor denne versjonen ble laget (f.eks. oppretting av dokument, rettelse av feil slik og sånn, implementering av metoden sånn, osv.)
- kvalitetssikres ved å utføre dokumentinspeksjon. Ved inspeksjon er det naturlig at følgende roller dekkes:
  - forfatter av dokumentet/koden som noterer hva som må gjøres av forbedringer og gjennomfører disse
  - dokumentbruker, f.eks. den som skal lese en brukerveiledning, den som skal skrive et svartbokstestprogram, ...
  - en megler som driver dokumentgjennomgangen videre og noterer vesentlige punkt som kommer opp, oppsummerer hovedmerknadene og avgjøre om retting og kanskje ny inspeksjon er nødvendig
  - inspektører som skal se kritisk men konstruktivt på dokumentet
- Kvaliteten på kode skal i tillegg sikres ved å utvikle testprogram som så kjøres for å kontrollere at ting virker som forutsatt (se neste seksjon). JUnit tilbyr et testrammeverk som skal brukes for å bygge (og kjøre) tester.
- Programvaren skal i utgangspunktet være plattformuavhengig, spesielt vil vi kreve at den kan kjøres på undervisningsanlegget (UA), som er et linux-anlegg ved Institutt for informatikk UiB.

- Innleveringer skjer ved å si fra (dvs. sende e-post til gruppe- og kursledere) om hvor det befinner seg en `README`-fil. Denne filen skal beskrive hvordan hele innleveringen kan hentes ut fra et Subversion-arkiv, hvilken kommando som skal brukes for at alle dokument formateres/kompileres, hvordan testene for alle komponenter kjøres og hvordan det kontrolleres at dette var vellykket. Videre må denne filen utpeke et dokument som gjør det mulig å finne ut av alle sider ved programvaren og dokumentasjonen av den, deriblant alle standarder og konvensjoner som ble fulgt.

Som en del av innleveringen skal også være et dokument som beskriver gruppens erfaring med prosjektarbeidet. Dette skal være et kortfattet dokument og skal inneholde:

- Beskrivelse av gruppens organisering (eventuelt ulike organiseringer dersom den er blitt endret underveis) med begrunnelse og en vurdering av hvordan det fungerte. Personnavn skal knyttes til de ulike roller.
- Hvilken programvareprosessmodell som ble benyttet til arbeidet, og en oppsummering av de arbeidsoppgaver som ble utført, med en angivelse av hvilke tidsrom i prosjektperioden de ulike aktivitetene strakk seg over. Det er helt vanlig at flere aktiviteter skjer samtidig. I slike tilfelle bør det angis hvor stor del av gruppens totale ressurs som til enhver tid (f.eks. for hver dag) som er knyttet til de ulike aktivitetene. Denne bør presenteres som et diagram med tidsaksen horisontalt og aktivitetsaksen vertikalt.
- En oppsummert oversikt over ressursinnsatsen som gikk med til arbeidet (basert på førte timelister som skal være i Subversion-arkivet). oppsummeringen skal fremheve ressursbruken innenfor følgende områder:
  - tid til planlegging
  - tid til å sette seg inn i problemene
  - tid til å skrive dokumentasjon
  - tid til å skrive/jobbe med kode
  - tid til støttefunksjoner (Ant, utprøving av tester, sjekking av README-filen, ...)
  - tid til administrasjon (føre timelister, møter etc.)
- En vurdering av prosjektet som skal inneholde
  - hva dere trodde dere skulle lære av prosjektet
  - hva dere lærte av prosjektet
  - hva dere tror var meningen med prosjektoppgaven

- Til presentasjonene vil være tilgjengelig tavle med kritt, lysarkprosjektør og bærbar PC med videoprosjektør.

  Programvare som skal demonstreres og dokument som skal vises fra PC må være innstallert og testet ut minst 2 timer før presentasjonen. Utlåns-PCen vil ha CD-leser og tilknytning til instituttets lokalnettverk.

  Gruppelederne og instituttet vil være behjelpelig med å lage (farge)lysark og installere dokumenter på den bærbare PCen. Husk at det blir svært kort tid til å installere og klargjøre maskinen i auditoriet før en gruppes presentasjon. Alt må derfor være klart og sjekket ut på forhånd.

Presentasjonene (også de andres) er en sentral del av pensum.

3 Programmering ved kontrakt og Java

Den programmeringsmetodikken vi skal bruke er basert på *programmering ved kontrakt*, et uttrykk formulert av Bertrand Meyer [1], sammen med noen elementer fra XP (extreme programming). Fordelen med denne arbeidsmåten er at vi raskt, dvs. når kontraktene er formulert, kan fordele mange mindre arbeidsoppgaver: de som skal bruke kontraktraktene og de som skal oppfylle kontraktene. Som oftest vil en arbeidsoppgave gå ut på å oppfylle noen kontrakter og stå fritt til å bruke andre (og til en viss grad) de samme kontraktene.

Det er også slik at ethvert programmeringsspråk kommer med sitt sett av basiskontrakter (f.eks. for heltall, flyttall og boolske verdier) som er oppfyllt av språkets innebyggede typer og operasjoner. Språk som Java er dessuten utstyrt med en stor samling kontrakter som dels leveres

sammen med språket (standard pakker), dels kan hentes fra andre leverandører ved behov[2]. Alle kontrakter, fra de som er levert med språket til de vi lager selv, utvider programmeringsspråket med en funksjonalitet vi ønsker oss for å løse de problemene vi skal takle.

I Java vil vi koble kontraktsbegrepet til grensesnitt-begrepet (*interface*) og en beskrive funksjonalitet vi ønsker som en del av grensesnittet. Følgende er en standardkontrakt på inf112 som vi ønsker at alle klasser skal oppfylle.

```
/**
 * @(#)Inf112Standard.java    1.0 26.01.2004
**/

/**
 * This contract described basic programming concepts we want all our classes
 * to support
 *
 * @version     1.0 26.01.2004
 * @author      Magne Haveraaen, modified by Jan Ivar Beddari
**/
public interface Inf112Standard {

/**
 * Tests whether the data invariant is satisfied.
 *
 * @return <code>true</code> if the test is successful, <code>false</code> otherwise
**/
public boolean DI();

/**
 * <pre>
 * Modifies:  None
 * Result:    Tests whether the argument value, which should belong to
 *            the same class, represent the same abstract value.
 * Equations: The equals method should be a equivalence relation, i.e.,
 *
 *            for any Inf112Standard x, y, z;
 *             x.equals(x)   // reflexive: x == x
 *             x.equals(y) == y.equals(x)   // symmetrical:  x == y  =>  y == x
 *             not x.equals(y) || not y.equals(z) || x.equals(z)
 *             // transitive: x == y && y == z  =>  x == z
 * </pre>
 *
 * @param x   the object to compare with
 * @return    <code>true</code> if the test is successful, <code>false</code> otherwise
**/
public boolean equals(Object x);

/**
 * Modifies: None
 * Result:   Returns a textual representation of the object. For
 *           complicated structures an XML format should be defined.
 * Example:  This can be used for outputting values in case an error is
```

[2]Dessverre er disse kontraktene ofte integrert i den pakken som oppfyller dem, noe som gjør handel med utbyttbare komponenter vanskeligere enn nødvendig.

```
 *           detected.
 * @return   a String representating the state of the object
**/
public String toString();

/**
 * <pre>
 * Modifies:   Object variable
 * Result:     Parses the input string x (which is assumed to be in a format
 *             compatible to that generated by the toString method) and sets the
 *             object variable to the corresponding value.
 * Exceptions: Throws an appropriate exception if the String has a wrong
 *             format.
 * Equations:  Inf112Standard x, y;
 *             ...
 *             x = y.clone();
 *             y.fromString(x.toString());
 *             assert x.equals(y)
 *               : "toString/fromString error: x=" + x.toString()
 *                                     + " y=" + y.toString();
 * </pre>
 * @param x    String describing the state of an object
 * @throws     Exception if the String has wrong format
**/
public void fromString(String x) throws java.lang.Exception;

/**
 * <pre>
 * Modifies: None
 * Result:   Clones the object, i.e., returns a deep copy of the object.
 * Remark:   Java automatically generates a shallow clone method (one-level) if the
 *           contract java.lang.Cloneable is implemented.
 * </pre>
 * @return   a clone (deep copy) of the object
 * @throws   CloneNotSupportedException if the clone operation fails
**/
public Object clone() throws java.lang.CloneNotSupportedException;

}
```

Hver klasse som implementeres oppfyller en (liste av) kontrakter. Det er vanlig at det finnes flere implementasjoner av hver kontrakt. F.eks. bør hver klasse vi implementerer oppfylle Inf112Standard-kontrakten i tillegg til minst en kontrakt for det egentlige formålet til klassen.

For å løse et problem trengs det ofte et tilpasset begrepsapparat, og ved å velge et passende sett kontrakter (uavhengig om de på det tidspunktet er implementert eller ikke) vil dette begrepsapparatet kunne bli definert. Om problemområdet vårt er testing og feilsøking kan vi typisk anta at alle klasser skal ha en toString-metode (Inf112Standard) slik at vi kan skrive ut de (abstrakte) verdiene et objekt inneholder. Om vi holder på å implementere en ny klasse (som bl.a. skal oppfylle Inf112Standard-kontrakten) så kan vi ved implementasjonen av klassens toString-metode anta at alle attributtene i klassen oppfyller Inf112Standard-kontrakten, og dermed har en toString-metode vi kan bruke.

I grunnen kan vi anta at alle problem vi skal løse kan beskrives i en kontrakt, og programmering blir da å definere attributtene og implementere metodene til denne kontrakten ved å bruke relevante kontrakter for å konstruere datastrukturen og beskrive algoritmene vi trenger.

Kontrakter som er logisk relaterte vil naturlig samles i en pakke. Grupper av implementasjoner med en viss indre sammenheng vil også kunne samles i en pakke. Med denne arbeidsformen vil det dog være naturlig at kontrakter (grensesnitt) og implementasjoner (klasser) kommer i hver sine pakker.

En kontraktsdefinisjon skal inneholde:

- et java-grensesnitt (`interface`) med en overordnet beskrivelse av hvilke begrepsapparatet kontrakten tilbyr.
  - for hver konstruktør som er relevant for kontrakten en metode som fanger opp konstruktørens funksjonalitet (både fordi grensesnitt i Java ikke tillater oss å beskrive konstruktører, men også fordi det er nyttig med slike metoder).
  - for hver metode
    - en liste over hvilke av argumentene som kan endre verdi (her skal objektvariabelen tas med om den kan endre verdi pga. metodekallet)
    - beskrivelse av hvilke forkrav som må være oppfyllt for at metoden skal kunne hvirke
    - beskrivelse av resultatet for hvert argument som kan få endret verdi pga. metodekallet.
- en svartboks testklasse som kaller alle metodene i grensesnittet med argumenter valgt etter følgende kriterier:
  - typiske, lovlige argumentverdier der det er mulig å sjekke metodens resultat mot på forhånd fastsatte resultatverdier. Dersom det lovlige området etter beskrivelsen er satt sammen av flere delområder skal det være noen typiske dataverdier innenfor hvert delområde.
  - verdier som ligger helt i grensen av det lovlige området. Det er i grenseområdene mange feil oppstår. Dersom det lovlige området er delt opp i flere underområder må det også lages testdata som ligger helt i kanten mellom hvert av de lovlige underområdene. For disse lovlige argumentverdiene må metodens resultat sjekkes mot forventede resultat.
  - typiske, ulovlige verdier. Disse skal oppdages av forkravssjekker i form av `assert`-setninger, og det kan da sjekkes om de rette unntakshoppene blir sendt ut fra metoden.

  Dersom en metode er delvis er karakterisert ved ligninger er det naturlig å bruke slike uttrykk for å bygge opp grundige tester.

  Det er ikke sikkert at beskrivelsen i kontrakten er spesialisert nok til at det er mulig å identifisere hvilke verdier som det kan testes mot. Da må testene skrives som metoder der de aktuelle testdataene og forventede resultatene gis som argumenter (når de blir kjent). Dermed kan et detaljert testopplgg være klart til denne tilleggsinformajson blir tilgjengelig og testene kan gjennomføres.

Det siste problemet ser vi når vi skal lage en svartbokstest for kontrakten Inf112Standard. Her har vi ikke nok informasjon til å si noe om formatet på strengrepresentasjonen eller for å lage objekter vi kan teste likhet på. Vi kan likevel lage en testklasse som vi så kan bruke i en sammenheng der vi har en klasse som implementerer grensesnittet.

```
/**
 * @(#)Inf112StandardBlackboxTest.java    1.0 26.01.2004
**/

/**
 * The blackbox test class for the Inf112Standard contract
 *
```

```
 * @version     1.0 26.01.2004
 * @author      Magne Haveraaen, modified by Jan Ivar Beddari
**/
public class Inf112StandardBlackboxTest {

/**
 * Modifies: None
 * Result: Tests the equivalence relation properties of equals
 *
 * @param x object to test
 * @param y object to test
 * @param z object to test
 * @return <code>true</code> if the test is successful, <code>false</code> otherwise
**/
public boolean testEqualsEquivalenceRelation
        (Inf112Standard x, Inf112Standard y, Inf112Standard z) {
    return x.equals(x)   // reflexive
            && (x.equals(y) == y.equals(x))   // symmetrical
            && (not x.equals(y) || not y.equals(z) || x.equals(z)); // transitive
}

/**
 * Modifies: None
 * Result:   Tests precondition of fromString
 *
 * @param x           object to test
 * @param wrongFormat string
 * @return <code>true</code> if the test is successful, <code>false</code> otherwise
**/
public boolean testFromStringPrecondition
        (Inf112Standard x, String wrongFormat) {
    Inf112Standard t = x.clone(); // avoid destroying x
    try {
        t.fromString(wrongFormat);
        return false;
    } catch (java.lang.AssertionError) {
        return true;
    }
    // should we also have tested that the value of t is unchanged?
}

/**
 * Modifies: None.
 * Result:   Tests string output/input.
 *
 * @param x object to test
 * @return <code>true</code> if the test is successful, <code>false</code> otherwise
**/
public boolean testToStringFromString (Inf112Standard x) {
    Inf112Standard t = x.clone(); // avoid destroying x
    t.fromString(x.toString());
    return t.equals(x);
}
```

```
} // end class
```

Merk hvordan vi ar brukt ligningene fra beskrivelsen av metodene til å formulere testene.

Når vi skal implementere en (liste av) kontrakter skal den inneholde:

- deklarasjon av hvilke kontrakter vi oppfyller og hva vi eventuelt arver. Husk at kontrakten Inf112Standard skal være med.

  For hver klasse må vi ha med:

  - en overordnet beskrivelse av hva denne implementasjonen representerer (f.eks. en subklasse av List med sorteringsmetode som er effektiv på lister som nesten er sortert på forhånd).
  - for attributtene: en overordnet beskrivelse av datastrukturen som er valgt med begrunnelse. For hvert attributt kan det også kommenteres hva den representerer. Husk at alle attributtene må være `private`.
  - for DI-metoden en skisse av hva datainvarianten er.
  - for equals-metoden en beskrivelse av hvilke verdier som skal være like.
  - for toString/fromString-metoden en skisse over strengformatet (gjerne XML-format).
  - for hver metode som er markert som konstruktør-metode i kontrakten en tilsvarende konstruktør.
  - for nye metoder eller konstruktør (de som eventuelt ikke er beskrevet i kontraktene) en beskrivelse av effekten i kontraktsformat. Lokale metoder gis attributt `private` mens de andre gis attributt `public`.
  - for definerte metoder (og konstruktører) må eventuelle justeringer av beskrivelsen gis. Dette kan være en styrking av forkravet (dersom en metode ikke kan implementeres blir forkravet `false`) eller en presisering av effekten.
  - for hver metode og konstruktør som implementeres
    - en skisse av algoritmen med en begrunnelse eller analyse. Merk at dette skal være en skisse som skal gjøre det lettere å forstå koden. Der selve koden er kort og oversiktlig kan det være unødvendig med en ytterligere beskrivelse.
    - test av forkrav med `assert` i starten av hver metode (om det lar seg gjøre). Dette sikrer at vi kan oppdage feil bruk av en metode i enhver kontekst ved å slå på `assert`-sjekkene.
    - test av datainvarianten med `assert` ved retur fra metoden. Dette vil avsløre mange programmeringsfeil på et tidligst mulig tidspunkt.

    Ellers skal god programmeringsskikk følges, med kommentarer til lokale variable, forklaring til vanskelige deler av koden, beskrivelse av invarianter (ting som er sanne) i vanskelige deler av koden, eventuelt betinget testing av invarianter med `assert`. Dersom koden f.eks. utnytter en spesiell egenskap ved datastrukturen eller en lite kjent del av java (altså noe som ikke er opplagt) bør dette kommenteres spesielt.

- Et klarbokstestprogram, vanligvis et for hver klasse, som tester alle konstruktørene/metodene i klassen.

  Et klarbokstestprogram kaller alle svartbokstester (med relevante data) for de kontraktene som inngår, og har i tillegg egne klarbokstester for hver metode. En metodes klarbokstest tar utgangspunkt i algoritmen når testargumentene skal velges, slik at hver setning i algoritmen blir utført minst en gang. I praksis forsøker vi å tilstrebe at

  - hver gren av valg/`try`/`assert`/...-setning blir utført
  - hver løkke blir utført 0 ganger, 1 gang, og mange ganger

For å oppnå dette må vi kalle samme rutinen flere ganger med ulike argument som fører med seg at de ulike setningene blir utført. Resultatet av hvert kall i testprogrammet skal sjekkes mot en kjent fasit. Å velge argument som skal teste hver gren kan være svært tidkrevende, og det er ofte vanlig at i første omgang velges et rimelig utvalg av argument. Etterhvert som svakheter i koden blir avslørt, så utvider man testene til også å dekke disse tilfellene. Testene blir dermed mer komplette etter som tiden går.

Det er viktig at testprogrammene ikke belaster brukeren med unødig informasjon. Testprogram bør derfor ikke ha annen utskrift enn en linje der det bekrefter at alle testene er godkjent. Dersom feil oppstår kan inntil en linje per feil skrives ut slik at det er lett å se hva som er galt, men strengt tatt kunne testprogrammet nøyd seg med å si at feil var detektert. Det er først ved avlusing at vi egentlig er interessert å vite hva slags feil som er oppstått.

Noen tips:

- Dersom et testprogram skal teste fil-IO er det lurt om programmet skriver filen først og så leser den inn – dermed kan programmet selv generere alle testfildata.
- Dersom et testprogram skal teste terminal-IO må testeren under testen få detaljert informasjon om hva som skal skrives/gjøres (f.eks. flytte skjermpekeren raskt i et bestemt mønster og klikke på visse punkt på skjermen). I så fall må det også fortløpende gis respons på hvordan reaksjonene fra programmet skal være, og eventuelt om deltestene var vellykket.
- Også slike ting som hastighet på database- og nettverkskode kan testes ved liten og stor belastning, og målte tider kan eventuelt sammenlignes med vedtatte makstider (enkeltvis eller som et gjennomsnitt).
- Verktøyet JUnit kan være nyttig i å sette opp testrammene og styre gjennomføringen av testene.

Testprogrammene bør kjøres regelmessig for å sjekke om koden til enhver tid oppfyller vesentlige egenskaper, spesielt etter endringer i kildekoden. Ant støtter dette på en god måte om den blir satt opp riktig.

Mye dokumentasjonen kan knyttes sammen med koden ved å legge den inn som strukturerte kommentarer. Et verktøy som "javadoc"vil da hente ut og formatere denne. Opplysninger om hvem som var med på å skrive en spesifikasjon eller lage en implementasjon, forfatterne, kan da legges inn som @author-kommentarer sammen med datoene for når endringene ble gjort. Videre kan (Subversion-)versjonsnummer og utfyllende opplysninger om det spesielle ved en versjon legges inn som @version-kommentarer.

4 Oppgaven

Druantia-systemet består av et rammeverk og ulike tjenester så som autentisering, FMS og Chat. Kvaliteten på koden i disse tjenestene slik den er ved semesterstart er for dårlig, mens rammeverket er noenlunde velstrukturert. Rammeverket inkluderer et databasegrensesnitt, en serverenhet, nettverksmodul og klientdel. Tjenestene er bygget over ulike samlinger av klasser og grensenitt, og mye av dette, så som trestruktur for meldinger, brukernavnhåndtering o.a. kan brukes videre.

For å kunne utnytte denne koden og tilhørende dokumentasjon fremover skal vi ekstrahere det som kan gjenbrukes og kvalitetssikre det som hentes ut. Det betyr at klassestrukturen må omorganiseres, kontrakter utarbeides der de ikke finnes, testprogram må skrives om og kompletteres, samt at støttedokumentasjonen må bearbeides. Videre bør beskrivelsene av kjente feil og svakheter i koden analyseres og funksjonsforbedringer introduseres. Dette kan være slikt som muligheten til å serialisere subtrær slik at de kan sendes over nettet.

Denne oppgaven skal løses i konkurranse mellom 3 jevnstore arbeidslag. Arbeidslagene ble nedsatt på samlingen 2004-01-22 kl. 1215, og etter den tid er forskjellige dokument som beskriver støtteverktøy og Druantia-systemet blitt gjort tilgjengelig. Selve oppgaven (dette dokumentet) ble gjort tilgjengelig på ettermiddagen 2004-01-28.

Innleveringstidspunkt er onsdag morgen 2004-02-18 kl. 1000.

De ulike arbeidslagene skal presentere sine løsninger i uke 9, med gruppe 1 på mandag, gruppe 2 på torsdag og gruppe 3 på fredag. Hver gruppe får 2*45 minutter til disposisjon, og det er viktig at alle deltakerne på hvert lag presenterer en bit. Dette krever koordinert fremføring og raske skift mellom personer. Hver gang en ny person kommer frem må denne presentere seg med navn.

Prosjektene vil evalueres basert på kvaliteten på presentasjonene og det innleverte arbeidet: dokumentasjon, enkeltheten i kjøring av tester og dekningsgraden av testene. Kvaliteten på testprogrammene vil sjekkes ved at det legges inn et sett med feil i de innleverte komponentene, og det sees hvorvidt testprogrammene detekterer disse.

Referanser

[1] Bertrand Meyer. Applying "design by contract". *Computer (IEEE)*, 25(10):40–51, October 1992.