DATALOGISK INSTITUT
DET NATURVIDENSKABELIGE FAKULTET
AARHUS UNIVERSITET

Hovedopgave

Master i Informationsteknologi
linien i Softwarekonstruktion

Distribuerede Unit Tests

af
Thomas Daugaard

15. juni 2010

Thomas Daugaard, studerende   Elmer Sandvad, vejleder

Datalogisk Institut
Aarhus Universitet
Åbogade 34
8200 Århus N

Tlf.: 89425600
Fax: 89425601
E-mail: cs@cs.au.dk
http://www.cs.au.dk/da

Contents

I Rapport 4

1 Motivation 4

2 Problemformulering 4

3 Afgrænsning 5

4 Baggrund 5
4.1 Disunit 5
4.1.1 Praktisk anvendelse 6
4.1.2 Inddragelse i denne rapport 6
4.2 Diagramnotation 6

5 Rapportens struktur 7

6 Teori 7
6.1 Tests 7
6.2 Unit tests 8
6.3 Integration tests 9
6.4 Distribuerede unit tests 10
6.4.1 Determinisme 10
6.4.2 Arkitekturer for test af distribuerede systemer 11
6.4.3 Controllability og observability 13
6.4.4 Test controllere og maskiner 15
6.5 Frameworks til distribueret unit test 15
6.5.1 GroboTestingJUnit 16
6.5.2 SysUnit 17
6.5.3 DisUnit 17
6.5.4 Pisces 17

7 Pisces og disunit 18
7.1 JUnit 18
7.2 Pisces 20
7.3 Disunit 20
7.3.1 Klassifikation af disunits arkitektur 20
7.4 T-Core disunit 21
7.5 Observe points 21
7.6 Eksempler på tests i disunit 21
7.6.1 Chatsystem 21
7.6.2 Design for test 22
7.6.3 FirstTestSuite 23
7.6.4 SecondTestSuite 24
7.6.5 ThirdTestSuite 24

8 Status 25
8.1 Erfaringer . . . 25
8.1.1 Anvendelsesområder . . . 25
8.1.2 Resultater . . . 25
8.1.3 Fordele og ulemper . . . 26
8.1.4 Cost/benefit . . . 28
8.1.5 Testfokus . . . 28
8.2 Disunit i forhold til eksisterende frameworks . . . 28

9 Videreudvikling af disunit 29
9.1 Arkitektoniske udfordringer i disunit . . . 29
9.1.1 Kobling til Pisces . . . 29
9.1.2 Inter-test kommunikation . . . 29
9.1.3 Stejl indlæringskurve . . . 29
9.2 Funktionelle udfordringer . . . 29
9.2.1 Differentierede timeouts . . . 29
9.3 Løsningsmuligheder . . . 30
9.3.1 Anvendelse af prototyping . . . 31
9.4 Prototypeforløbet . . . 32
9.4.1 Subprototype, kommunikationslag . . . 33
9.4.2 Subprototype, JUnit 4 . . . 33
9.4.3 Subprototype, T-Core JUnit 4 . . . 34
9.4.4 Endelig retrofitting . . . 34

10 Disunit 2.0 34
10.1 Tests i disunit 2.0 . . . 35
10.1.1 JUnit 4 . . . 35
10.1.2 Disunit 2.0 og JUnit 4 notation . . . 36
10.2 Ansvarsfordeling . . . 38
10.2.1 JGroups . . . 38
10.2.2 Communication . . . 39
10.2.3 Machines . . . 39
10.2.4 JUnit . . . 40
10.2.5 Observepoint . . . 40
10.3 Klassifikation af disunit 2.0's arkitektur . . . 40
10.4 Evaluering af disunit 2.0 . . . 41

11 Kildekode 42

12 Konklusion 42

13 Videre arbejde 44
13.1 Real distribution . . . 44
13.2 Non-funktionelle tests . . . 44
13.3 Continuous integration . . . 45
13.4 Parallel unit testing . . . 45

13.5 Automatisering 45

14 Tak **45**

II Bilag - detaljerede arkitekturbeskrivelser **48**

15 Disunit arkitektur **48**
15.1 Pisces 48
15.1.1 RemoteTestRunnerAgent 48
15.1.2 RemoteTest 49
15.1.3 Eksekvering af tests 51
15.1.4 Uhensigtsmæssigheder i Pisces 52
15.2 Disunit 54
15.2.1 Udvidelser til Pisces 54
15.2.2 DisunitTestSuite 55
15.2.3 Kommunikation mellem controller og TestMachines 56
15.2.4 Koordination af tests 56
15.3 Observepoints 57
15.3.1 Anvendelse 57
15.3.2 Timingsmæssige udfordringer 59
15.4 T-Core integration 59
15.4.1 T-Core 59
15.4.2 Disunit i T-Core 60
15.4.3 Principper i T-Core disunit 61
15.4.4 DistributedObservePointHelper 62
15.4.5 Non-funktionelle tests 62

16 Disunit 2.0 arkitektur **64**
16.1 Kommunikationslag 64
16.2 JUnit - DisunitRunner 66
16.3 T-Core disunit 68
16.4 Observe points 69

Part I
Rapport

1 Motivation

Test driven development er allerede godt på vej ind i virksomhedernes udviklingsprocesser og har gjort automatiske unit tests til en fast del af udviklernes arbejde, f.eks. gennem brug af frameworks som JUnit. Disse unit tests udføres som regel på entrådet, sekventiel kode. Når der arbejdes med distribuerede systemer er traditionelle unit tests ikke tilstrækkelige til at eksersere koden realistisk. Når kode kommunikerer på tværs af tråde, processer og netværk opstår en non-determinisme, som typisk ikke er til stede i enkelttrådet kode. Non-determinisme træder derfor først frem ved integrationstest, hvor koden sættes sammen med andre komponenter, og kan lede til uforudsete hændelser i koden.

I distribuerede systemer kan trinnet fra unit tests til integrationstest derfor være så stort, at det kunne være en fordel at indføre et trin imellem, der er i stand til automatisk at teste units eller grupper af units, som i dette dokument benævnes distribueret unit (DU) tests. DU tests tager højde for afhængigheder i kommunikationen i distribuerede systemer og kan udover at være en hjælp under nyudvikling udgøre et nyttigt supplement til regressionstest, når der videreudvikles på kode, der traditionelt kun er dækket af unit tests.

At genskabe en fejl i et distribueret system, observere dens følgende svigt, finde defekten og senere at validere, at defekten er fjernet, kan være en besværlig proces, hvis man er nødt til at teste koden manuelt, f.eks. på et test site med et antal maskiner. Her vil det være en fordel i et simpelt framework a la JUnit at kunne skrive en distribueret unit test, som kan observere svigt gennem input, der nødvendigvis må komme fra samtidigt eksekverende kilder.

Disunit er et sådant framework, udviklet i en softwareafdeling i Terma A/S. Det praktiske arbejde med disunit afslører nogle problemer med udviklingen af distribuerede unit tests, og det kunne være en fordel mere grundigt at undersøge teoretisk baggrund og eksisterende frameworks for at få inspiration til videreudvikling eller refaktorering af disunit.

2 Problemformulering

Analyser disunit's arkitektur og løsninger, som har fundament i praktiske udviklingssituationer. Fremsøg og undersøg litteratur om distribuerede tests, og undersøg eksistensen af andre frameworks, som ligner disunit. På baggrund af teori, andre frameworks og de praktiske erfaringer med disunit, kom med forslag til ændringer af disunit og argumenter for, hvorledes de vil påvirke det videre praktiske arbejde med udviklingen af disunit.

3 Afgrænsning

Der kigges udelukkende på automatiske tests udviklet i J2SE[1].

Undersøgelserne vil koncentrere sig om defekt opdagelses teknikker, dvs forsøg på at få systemer til gå i fejl-tilstand, der kan observeres som et svigt, med henblik på at finde den eller de udløsende defekter i koden.

Arbejdet spænder over unit tests, men bruges typisk i en bredere forstand, således at internt kommunikerende, asynkrone strukturer i en unit testes. Det ligger på et niveau over unit tests men under integration tests og navngives distribuerede unit tests. Dog kan disunit, dvs. det eksisterende værktøj, også anvendes til integrations test.

Der kigges udelukkende på unit og DU tests. Integration tests vil kort blive berørt, mens system og acceptance tests ikke betragtes. Kun funktionelle aspekter undersøges, dvs. om DU'er leverer forventet output i forhold til input. Test af DU'ers kvalitetsmæssige attributter, såsom tests af performance og fault tolerance, betragtes som udgangspunkt ikke.

4 Baggrund

En del af arbejdet, som er beskrevet i dette dokument, har været i gang i omkring to år og har udmøntet sig i et framework til distribuerede unit tests, der bliver anvendt i en softwareudviklingsafdeling i Terma A/S. Arbejdet blev påbegyndt som et diplom-speciale, der blev udskudt til dette master-speciale, men i mellemtiden har udviklingsafdelingen haft en del erfaringer med frameworket. Det er blevet brugt både til dets planlagte formål og til længerevarende test af stabilitet, og selvom sidstnævnte vil blive kort behandlet, ligger hovedfokus på funktionelle aspekter.

Arbejdet med frameworket, som blev døbt disunit, inddrages derfor i dette dokument. Disunit beskrives kort i dette afsnit, hvorefter en uddybende gennemgang af arkitekturen foreligger efter undersøgelsen af teori og andre frameworks.

4.1 Disunit

Frameworket disunit opstod som en resultat af nogle forsøg med et eksisterende framework, Pisces [Pisces], i forbindelse med forberedelse til et diplom-speciale. Afprøvningerne foregik i en softwareudviklingsafdeling i Terma, som kunne se nogle fordele i at kunne lave automatiske tests af deres distribuerede komponenter og dermed undgå i hvert fald en del af de manuelle og omstændige test af komponenterne på et testsite.

Afprøvingerne begyndte med en version af Pisces, som viste sig ganske passende til opgaven på grund af sit udgangspunkt i JUnit[JUnit 3] og de simple abstraktioner, der var en del af værktøjet. Abstraktionerne blev udbygget i et tyndt lag ovenpå Pisces, som blev døbt disunit (navnet er en sammentrækning

[1] Java 2 Standard Edition

af distributed unit og minder samtidig om det engelske ord disunite (adskille) for at afspejle det distribuerede aspekt).

Kort fortalt, giver disunit testprogrammøren mulighed for at instantiere en række maskiner, som repræsenterer fysiske maskiner, som kan emuleres på en enkelt maskine ved at hver maskine repræsenteres af en separat proces. Dette gør det nemt for testprogrammøren at sidde ved sin PC og udvikle og afprøve testen. Sidenhen kan testen flyttes til rigtige fysiske maskiner, uden at testkoden skal ændres.

4.1.1 Praktisk anvendelse

Disunit udviklede sig og blev efterhånden indført som standard framework til distribuerede unit tests i udviklingsafdelingens platforms-team. Dette team udvikler og vedligeholder Termas C3[2] system platform, T-Core. T-Core platformen er kendetegnet ved en høj grad af distribution forstået således, at dens enkelte komponenter indeholder meget multitrådet kode og at en stor del af dem beskæftiger sig med kommunikation med andre komponenter i platformen eller med klientkomponenter på tværs af tråde, processer og netværk.

Disunits muligheder efterlod dog testprogrammøren med et stort fodarbejde i forbindelse med udvikling af nye tests. Derfor blev der udviklet en udvidelse, som understøtter T-Core platformen, og som er i stand til at starte den op i de enkelte maskiner, som programmøren kan instantiere.

4.1.2 Inddragelse i denne rapport

Som nævnt inddrages erfaringerne med disunit i denne rapport. De mange praktiske erfaringer kan i kombination med teorien og undersøgelsen af andre frameworks bidrage til at belyse de områder, hvor disunit kan forbedres, eller hvor det med fordel kunne have været lavet anderledes fra begyndelsen. Arbejdet med disunit udgør altså et stærkt fundament for denne rapport.

Arkitekturen i Pisces og disunit og eksempler på testcases findes i særskilte afsnit.

4.2 Diagramnotation

Arkitekturbeskrivelserne i dette afsnit er inspirerede af metoden beskrevet i [Christensen et. al], som foreslår tre forskellige viewpoints sammen med en beskrivelse af arkitektoniske krav. Der anvendes som udgangspunkt UML 2.0. Arkitektoniske krav beskrives i et enkelt tilfælde ved hjælp af kvalitetsattributter som i [Bass et. al]. Alt efter, hvad der vurderes passende for at beskrive den enkelte arkitektur, kan der indgå følgende viewpoints:

- Module viewpoint: Pakke- og klassediagrammer
- Component and Connectors viewpoint: Sekvensdiagrammer og objektdiagrammer. En speciel version af objektdiagrammer er af og til brugt, hvor

[2] command, control and communications

en UML node anvendes til at illustrere i hvilken JVM[3] et givent objekt lever i tilfælde af associationer på tværs af processer.

- Allocation viewpoint: Deployment diagram.

Eventuelle afvigelser fra UML 2.0 er forklaret i de enkelte diagrammer.

5 Rapportens struktur

Her følger en kort beskrivelse af rapportens struktur og opbygning.

Først gennemgås teori for unit tests og beskrivelse af de begreber, fra eksisterende litteratur om emnet og fra distribuerede test, som anvendes i resten af rapporten. Herunder præsenteres to arkitekturer for test af distribuerede systemer.

Eksisterende frameworks, som understøtter test af distribuerede systemer, beskrives med henblik på at udnytte eventuelle erfaringer. Herunder undersøges arkitekturen af Pisces, som er et open source framework til test af distribuerede systemer. Disunit, som ligeledes er et framework til test af distribuerede systemer, er baseret på Pisces og er blevet anvendt i en periode i Terma A/S. Arkitekturen i disunit præsenteres sammen med konkrete eksempler på udvikling af test.

Der laves status over disunit med inddragelse af erfaringer fra den praktiske brug i Terma A/S. Ulemper og fordele afklares, og på basis af disse, defineres en række arkitektoniske punkter, hvor det kan være fordelagtigt at videreudvikle disunit.

Et prototype forløb, hvor erfaringerne fra disunit anvendes til disunit 2.0, som er navnet på det nye framework, beskrives dernæst, hvorefter arkitekturen af disunit 2.0 præsenteres.

Rapporten afsluttes med beskrivelse af en evaluering af disunit 2.0, foretaget i Terma A/S, samt en opsummering af resultaterne af hele projektet.

6 Teori

Termerne, der fastlægges i dette afsnit er alle inspirerede af [IEEE], men er oversat til danske betegnelser for at passe naturligt ind i sproget

6.1 Tests

At udføre en eller anden form for test af systemer er en forankret disciplin i mange udviklingsafdelinger.

> test: Processen at operere på et system eller en komponent under bestemte forudsætninger, mens man observerer eller optager resultaterne og laver en evaluering af et aspekt af systemet eller komponenten

[3] Java Virtual Machine

Dette dokument koncentrerer sig om tests af units og distribuerede units (se definition senere). I test processen tales som regel om to processer.

> validering: Processen at evaluere et system eller en komponent undervejs i eller ved slutningen af en udviklingsproces for at beslutte om det opfylder de specificerede krav
>
> verifikation: Processen at evaluere et system eller en komponent for at beslutte om produkterne fra en given udviklingsfase opfylder betingelserne, som forelå ved begyndelsen af fasen.

Dette dokument beskæftiger sig med verifikation, som populært sagt forsøger at afprøve, om koden er bygget rigtigt. Her er altså fokus på, om koden fungerer, som det forventes i den nuværende kontekst. Det forventede er desuden teknisk målbart i modsætning til valideringen, hvor der er fokus på, om koden leverer det, kunden ønsker. Verifikation skal ikke forveksles med formel verifikation, hvor man forsøger at føre matematisk bevis for kodens korrekthed og som ikke er en testdisciplin.

Når man tester forsøger man gennem forskellige teknikker at bringe det testede i sådan en tilstand, at det udviser en opførsel, der afviger fra det testedes specifikationer.

> defekt (bug): en design eller kodefejl i en unit, som kan lede til en fejl.
>
> fejl (error) - manifestationen af en eller flere defekter i en units runtime tilstand. En fejl kan lede til et svigt.
>
> svigt (failure) - en observeret afvigelse i en units runtime tilstand i forhold til dens specifikationer. Et svigt skyldes en eller flere fejl.

Med andre ord forsøger man at få sin unit i en fejl-tilstand, som gør at man kan observere et svigt. Derefter kan man forsøge at finde den eller de defekter i koden, som er skyld i fejlen.

I daglig tale vil man typisk omtale ovenstående definitioner af defekter, fejl og svigt som fejl. Bemærk forskellen mellem en fejl og et svigt: Et svigt er en observeret afvigelse mellem specifikation og opførsel, mens fejl blot er en systemtilstand, der (ikke nødvendigvis) kan lede til et svigt. Se evt [Bruegge & Dutoit].

Som eksempel kan betragtes en liste-implementation i et system. Der er en defekt i en algoritme i liste-implementationen, som gør det sidste element i listen utilgængeligt på grund af en forkert pointer. Denne (runtime) tilstand er en fejl. Men svigtet opstår først, når eller hvis systemet runtime tilgår det sidste element i køen.

I denne rapport bruges ovenstående begreber som defineret i dette afsnit. Bemærk at der er en uoverensstemmelse mellem disse definitioner og JUnits navnekonventioner. Se nærmere i afsnit 7.1.

6.2 Unit tests

De fleste softwareudviklere stifter bekendtskab med verifikation gennem udvikling af unit tests.

unit - en stump kode, som udgør så lille en del, at den er egnet til traditionel unit tests. I dette dokument er en unit typisk en klasse eller en begrænset mængde af klasser i Java.

unit test - kode, typisk sekventielt, som ekserserer koden i en unit og verificerer at output er som forventet i forhold til input. I dette dokument er en unit test en klasse eller en begrænset mængde klasser som er udviklet i frameworket JUnit

Mange udviklingsafdelinger har indført en disciplin med at udvikle efter test driven development eller i det mindste at udvikle unit tests efter units er blevet udviklet. Disse unit tests kan køres hurtigt, så man udover til umiddelbar respons under udvikling af en unit, også kan anvende dem til regressionstest og continous integration.

regressionstest: Selektiv gentesting af et system eller en komponent for at verificere, at ændringer ikke har forårsaget uhensigtsmæssig effekt og at systemet eller komponenten stadig opfylder de specificerede krav.

continuous integration: Udførsel af regression testing ved check-in af kode til kildekodestyringssystem med formålet at sikre, at flere udviklere, som arbejder på det samme kode gennem kildekodestyringssystemet ikke checker kode ind, som forårsager uhensigtsmæssig effekt.

JUnit[JUnit 3] er et populært og udbredt framework til udvikling af unit tests. De fleste Java-programmører er bekendte med frameworkets simple mekanikker og kan umiddelbart forstå kode, som er skrevet i frameworket. Et framework som JUnit kan også betragtes som en del af et test harness.

test harness (el. harness): En samling af software komponenter og test fixture som skaber et kørende miljø for en unit, giver den input og monitorerer dens output.

test fixture (el. fixture): Data, som er tilgængelig for den unit, der testes, og som bevirker, at resultatet af testen kan genskabes.

6.3 Integration tests

I niveauet over unit tests tales typisk om integration tests.

integrationstest: Tests hvor software eller hardware komponenter eller begge, kombineres og testes for at evaluere interaktionen mellem dem

Hvor snittet lægges mellem unit og integration testing kan være et fortolkningsspørgsmål, men typisk tester man i integrationstesten på interfaces, altså med black-box teknikker, hvor komponenternes interne struktur er skjult for testeren. Derimod er en unittest ofte baseret på white-box teknikker, hvor man aktivt udnytter kendskabet til unit'ens indre struktur, f.eks. ved at sikre sig at unit'en gennemgår alle mulige stier mindst en gang.

Det kan være nødvendigt at anvende white-box teknikker for at teste interaktion mellem eller internt i units. Således er der brug for mellem unit tests og integration tests at indføre et led, når man i et stærkt distribueret miljø udvikler units, som f.eks. i et system kodet i J2SE med flere JVMs kørende på tværs af netværk eller processer. Også i multitrådede programstumper kan man have behov for at verificere, at de internt kører som forventet, inden de kastes ud i en egentlig integrationstest. Ellers risikerer man at observere svigt på baggrund af fejltilstande, hvis defekter ligger dybt i forhold til en test af kontrakterne mellem komponenters interface.

Derfor definerer dette dokument distribuerede unit tests, som et lag af tests, der kan anses for at ligge på et højere niveau end unit tests, men lavere niveau end integration tests.

6.4 Distribuerede unit tests

I Terma A/S's softwareafdeling, som er omtalt i dette dokument, blev der før indførelsen af disunit anvendt unit tests, hvorefter de komponenter, units'ene indgår i, er blevet afprøvet i integrationstest, hvor de er blevet sat sammen med andre komponenter - bl.a. diverse GUI-komponenter - til observation af svigt. Det har gjort det svært at lokalisere defekter og verficere at de er rettet, men fremgangsmåden er blevet anvendt i høj grad, bl.a. til at genskabe brugerrapporterede svigt. I flere tilfælde er der, for at rette op på ovenstående problemstilling, blevet udviklet specielle og domænespecifikke test frameworks for hver enkelt komponent, som ikke umiddelbart er nemme at forstå for en udvikler, der ikke før har arbejdet med den specifikke komponent.

Som nævnt under integration tests, defineres distribuerede unit tests som liggende lige over unit tests, og definitionen ligger da også nær unit tests.

> distribueret unit - et stykke logisk sammenhængende kode i Java, bestående af en eller flere units, som har intern kommunikation på tværs af tråde, processer eller netværk. En distributeret unit betegnes af og til i dokumentet som en DU. DUUT står tilsvarende for distribueret unit under test.
>
> distributeret unit test - et program, som tester en distribueret unit ved at give inputs fra flere kilder, som kan befinde sig i andre tråde, processer eller på andet hardware, og verificerer at output er som forventet i forhold til input. En distribueret unit test betegnes af og til i dokumentet som en DUT.

Altså lægges der vægt på at distribuerede unit tests skal være automatiske, dvs. kan udføres maskinelt, og at der skal være et element af distribution i det testede kode.

6.4.1 Determinisme

At gå fra et framework, som er beregnet til at teste sekventielle units til at teste distribuerede units, er ikke helt trivielt. Typisk er units deterministiske og vi

ved, f.eks. gennem brug af white-box teknikker, hvad coverage af en test af en unit er. Gennem det input vi giver, kan vi forudsige de tilstande, som unit'en gennemgår og dermed forudsige output, som vi kan evaluere på. Såfremt en unit er non-deterministisk (den anvender random funktioner, eller er afhængig af ekstern diskdata eller tid), kan den bringes i en deterministisk tilstand ved at testframeworket leverer en test-harness, som dikterer disse omgivelser.

Distribuerede systemer (og dermed distribuerede units) er af natur non-deterministiske, da de kan påvirkes af scheduling-algoritmer (multitrådede components) eller af tid ved sand parallel udførsel, som bevirker at mængden af mulige tilstande bliver uendelig. Dermed skal de til en vis grad gøres deterministiske, for at testen er i stand til at evaluere fornuftigt på outputtet.

Artiklen "Architectures for testing distributed systems" [Ulrich&König] omhandler problemstillinger ved at teste distribuerede systemer. En vigtigt pointe fra artiklen er:

> "In testing distributed systems the information about action names observed during a test run is not sufficient to assess conformance between specifications and the SUT [4]. Due to the existence of multi-rendezvous among components of the SUT, the tester must also know what components participate in a specific multi-rendezvous. Furthermore, the issue of non-determinism within the SUT requires that the tester has the power not only to observe an action of the SUT, but must also control the occurence of a multi-rendezvous. Thus the crucial point in testing distributed systems is to perform a deterministic run." [Ulrich&König, s98]

Der understreges altså i ovenstående citat at for at teste distribuerede systemer, må testeren have mulighed for at gøre (relevante dele af) forløbet i DUUT deterministisk. Som det senere uddybes, er de helt centrale mekanismer i disunit og i andre frameworks netop at styre eller være i stand til at reagere på hændelser, som kommunikeres internt i testframeworket eller gennem output og input fra DUUT.

6.4.2 Arkitekturer for test af distribuerede systemer

I [Ulrich&König] skelnes mellem to forskellige arkitekturer for test af distribuerede systemer. Den ene består af en global tester, som egenhændigt styrer testforløbet. Den anden består af distribuerede testere, som hver især udfører en del af den globale testcase, og som typisk kommunikerer med hinanden.

I et distribueret system, som ønskes testes, taler man om PCO'er (Points of Control and Observation).

> PCO: They (PCO's) are defined in terms of points closest to IUT [5] at which control of input events and observation of output events may take place during the testing process [Cacciari&Rafiq, s769]

[4] SUT: System Under Test

[5] Implementation Under Test

Arkitekturen med den globale tester er illustreret i figur1. Her ses én tester, som kontrollerer DUUT ved at give input til PCO 1 og PCO 2. Samtidig får den output fra PCO 3 og PCO 4, som f.eks. kunne være DUUT's eksponerede interface (dermed er PCO 3 og PCO 4 en del af DUUT, selvom tegningen ikke helt præcist udtrykker det). PCO 1 og PCO 2 er placeret inde i DUUT for at illustrere at det kan være håndtag, som er indført i forbindelse med design-for-test, altså hvor DUUT er udviklet med henblik på at blive testet af det framework som Tester indgår i.

Fordelen ved den globale tester er simpliciteten. Ulempen er, at den globale tester kontrollerer SUT så hårdt, at der kan argumenteres for, at aspekter af samtidighed forsvinder [Ulrich&König, s99].

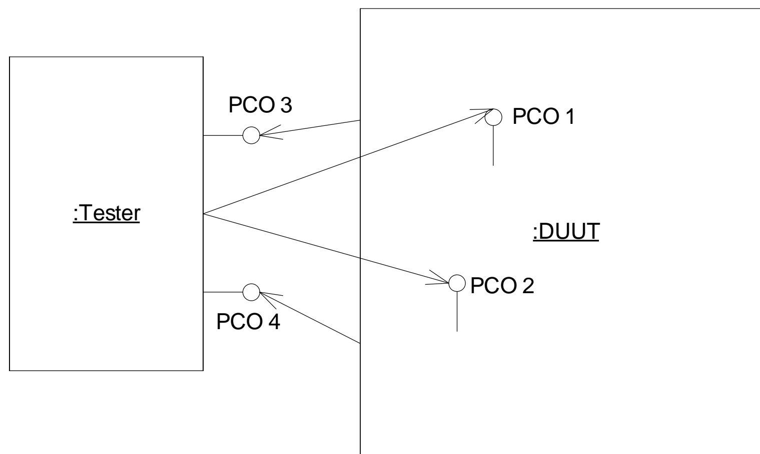

Figure 1: Test arkitektur med global tester. Kasserne repræsenterer henholdsvis den globale tester og DUUT. PCO'erne er illustreret ved hjælp af UML interface notation, da de udgør en grænseflade til testeren. Interfaces placeret inde i en kasse viser, at det er indført som en del af design-for-test. Pilene indikerer brug af interfaces

Den anden arkitektur består af distribuerede testere, som i princippet kører uafhængigt af hinanden, men som tilsammen udfører en global test case. Hver distribueret tester kan ses som en partiel testcase af den globale. De distribuerede testere må kunne koordinere testforløbet med hinanden, dels for at holde et "globalt overblik" over forløbet og for at de kan synkronisere mellem hinanden for at vide, hvornår de skal give input eller vente på output. Denne koordination kaldes for TCP.

> TCP: En Test Coordination Procedure styrer de distribuerede testeres opførsel.

Arkitekturen med distribuerede testere er illustreret i figur 2. Her er de distribuerede testere udstyret med en kommunikationskanal, som er uafhængig af

DUUT og som anvendes til at realisere en TCP. TesterA, TesterB, TesterC og TesterD kører på forskelligt hardware eller i forskellige processer.

Ulempen ved distribuerede testere i forhold til en global er kompleksitet. Desuden skal en TCP implementeres, hvilket kræver en eller anden form for separat kommunikation. Til gengæld testes SUT på mere distribueret vis, idet nogle tests kan køre samtidigt.

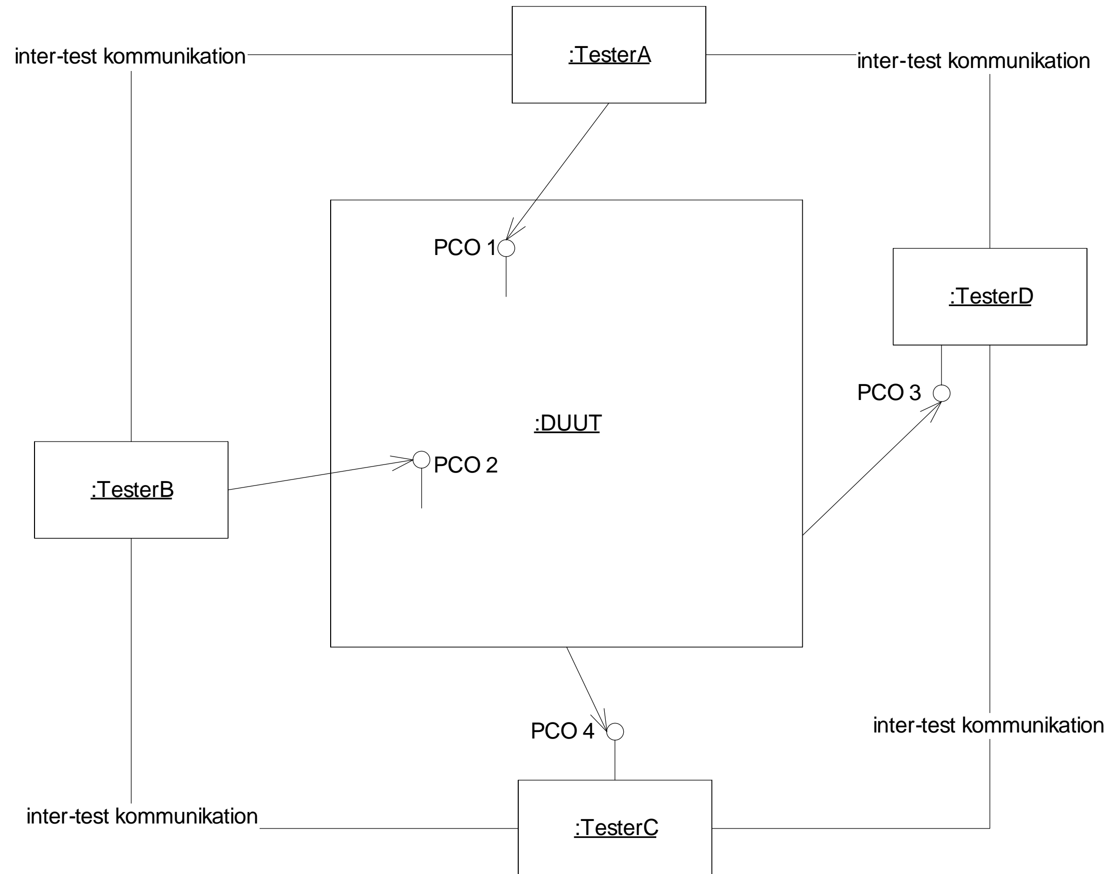

Figure 2: Test arkitektur med distribuerede testere. Kasserne repræsenterer testerne og DUUT. Testerne kommunikerer ved hjælp af en kommunikationskanal (inter-test kommunikation), som bruges til at realisere testens TCP. PCO'erne er vist som UML interfaces. Interfaces placeret inde i en kasse viser, at det er indført som en del af design-for-test. Pilene indikerer brug af interfaces

6.4.3 Controllability og observability

I forbindelse med distribuerede testere nævnes de to begreber observability og controllability, som blev defineret i [Cacciari&Rafiq].

> Controllability: ... may be defined as the testing power or the capability of the test system to realize input events at corresponding PCOs in a given order [Cacciari&Rafiq, s771]
>
> Observability: ... may be defined as the testing power or the capability of the test system to determine the output events and the order in which they have taken place at the corresponding PCOs. [Cacciari&Rafiq, s771]

Controllability giver testeren mulighed for at styre DUUT'en, så rækkefølgen af input events til DUUT'en bliver deterministisk. I kontrast hertil betragtes et controllability problem, når distribuerede testere sender input til DUUT via forskellige PCO'er. DUUT'ens distribuerede natur bevirker, at testerne ikke ved i hvilken rækkefølge, input behandles, hvilket kan have afgørende betydning for udfaldet af testen.

> Eksempel på et controllability problem: DUUT er en chatserver, som modtager beskeder fra chatklienter og sender dem ud i den rækkefølge, de er blevet modtaget. Tester 1 og Tester 2 agerer chatklienter. Tester 1 sender besked A og Tester 2 sender besked B. Tester 1 og 2 kører distribueret, altså parallelt, og den non-deterministiske natur gør, at man ikke på forhånd kan afgøre om DUUT modtager besked A eller besked B først.

Observability er testernes mulighed for at observere output fra DUUT'en, således at de er klar over i hvilken rækkefølge de er ankommet til PCO. I kontrast betragtes et observability problem, når en tester ikke kan bestemme, hvilken testers input, der er årsag til et givet output.

> Eksempel på et observability problem: Betragt chatserveren som DUUT fra før. Tester 1 og 2 sender nu samme besked A. Det antages, at controllability problemet er løst, så det er sikkert, at Tester 1 har sendt før Tester 2. En Tester 3, som blot betragter beskeder fra chatserveren, kan ikke være sikker på, hvilken af Tester 1 og Tester 2's input, de to outputs fra DUUT kommer fra.

En letlæselig og -forståelig beskrivelse af controllability og observability kan findes i en sammenfatning i [Binder]:

> "To test a component, you must be able to control its input (and internal state) and observe its output. If you cannot control the input, you cannot be sure what has caused a given output. If you cannot observe the output of a component under test, you cannot be sure how a given input has been processed."

[Cacciari&Rafiq]argumenterer for, at controllability og observability problemer kan løses ved at indføre et kommunikationslag, som er uafhængigt af DUUT, og som testerne kan bruge til at udveksle beskeder på. Distribuerede testere med et kommunikationslag kan ses illustreret i figur 2.

Controllability problemer løses ved, at testere sender koordinationsbeskeder efter de har sendt input til DUUT og ved, at den er modtaget. Andre testere kan modtage koordinationsbeskeden og vide, hvornår DUUT'en har modtaget hvilke input. I forhold til controllability eksemplet fra før, betyder dette, at Tester 1 kan sende en koordinationsbesked til Tester 2, efter den har sendt sin chatbesked til DUUT og har fået modtagelsen bekræftet, og dermed er rækkefølgen sikret.

Observability problemer løses ved, at testere sender koordinationsbeskeder, efter de har modtaget output fra DUUT, som svarer til deres input. I forhold til observability eksemplet fra før, får Tester 3 nu mulighed for at modtage en koordinationsbesked fra Tester 1, når sidstnævnte har modtaget sin egen besked fra DUUT. Hvis Tester 3 venter på koordinationsbeskeden har den mulighed for at afgøre, hvilken Tester output kommer fra.

Hvis problemerne ikke kan løses ved brug af DUUT's interface, kan design for test overvejes. I design for test bliver DUUT forberedt på at skulle testes, når den udvikles. Derved opstår der mulighed for at koble sig på PCO'er, der ikke er en del af DUUT's egentlige (funktionelle) interface.

6.4.4 Test controllere og maskiner

I denne rapport anvendes begreberne (test) controller og (test) maskine.

> (test) maskine: Funktionalitet, som er i stand til at modtage en test fra en controller, udføre den og returnere resultatet. En maskine vil altid køre i en anden proces end en controller og muligvis på andet fysisk hardware. Begrebet maskine skal ses som en abstraktion over denne varians (Andet fysisk hardware vs. anden proces på samme fysisk hardware) og skal ikke forvekles med "andet hardware". I rapporten anvendes ofte betegnelsen "fysisk maskine" om "andet hardware", dvs. f.eks. en PC.
>
> (test) controller: Funktionalitet, som er i stand til at håndtere maskiner, dvs. starte dem dynamisk, samt delegere test til dem og indsamle resultater fra flere maskiner. En controller udgør derfor en (delmængde af) TCP.

6.5 Frameworks til distribueret unit test

Unit tests i Java er som nævnt understøttet af det velkendte JUnit[JUnit 3] framework. Som udgangspunkt understøtter JUnit frameworket ikke tests, som tester multitrådet kode og dermed heller ikke tests af kode som kører i flere Java VMs.

Det er interessant at kigge på, hvilke frameworks der eksisterer til at løfte JUnit frameworket op til at kunne understøtte distribuerede unit test. Såfremt der findes sådanne frameworks, stilles en række krav, hvis det skal kunne sammenlignes med disunit:

- at det anvender samme termer som JUnit, hvor det er muligt, så Java udviklere, der allerede kender JUnit, oplever en grad af genkendelse

- at det er forholdsvis simpelt at skrive en test, f.eks. at det ikke kræver et unødigt overhead af konfigurationsfiler.
- at det ligesom JUnit opererer i et kodenært niveau. Dette er et krav, fordi systemerne i udviklingsafdelingen er udviklet i J2SE, og derfor kræver testmiljøet lige så høj fleksibilitet som selve udviklingsmiljøet. Det er ikke hensigtsmæssigt at være bundet op på et bestemt paradigme for testudførslen bortset fra, hvad JUnit definerer.
- at det leverer abstraktioner, som dynamisk kan afvikle kode i flere JVMs på samme tid.
- at det, jævnfør afsnittet om observability og controllability, har en intern kommunikationsmekanisme, som er uafhængigt af systemet, der testes.

Sammenfattet kan det udtrykkes, at frameworket skal være så kodenært og simpelt som muligt, så udviklere i J2SE opnår maksimal fleksibilitet i forhold til sprogets muligheder og til at integrere det ind i eksisterende software, der som udgangspunkt - ligesom T-Core - ikke er udviklet til at blive testet med det.

Efter søgninger på internettet, viser det sig, at der eksisterer indtil flere forsøg på at gøre JUnit i stand til at udføre distribuerede tests. De næste underafsnit gennemgår disse.

6.5.1 GroboTestingJUnit

GroboTestingJUnit (fundet gennem [Le]), som er et subprojekt af GroboUtils, er en række opensource extensions til JUnit, som bl.a. muliggør test af multitrådede Java programmer.

På projektets hjemmeside [Grobo] illustrerer et eksempel formålet med tests lavet i GroboTestingJUnit. Det står hurtigt klart, at ideen ikke er at introducere determinisme, men derimod at stressteste en unit, så den afslører defekter i trådsikkerheden, altså i synkroniseringen mellem forskellige trådes adgang til fælles data.

Det er muligt at nedarve fra en TestRunnable, som er en test, der vil køre i sin egen tråd. Her kan man tilgå og ændre data i unit'en, der testes. Med en MultiThreadedTestRunner er det muligt at køre en række instanser af disse TestRunnables parallelt i håbet om, at få unit'en til at svigte. Desuden er det muligt at lave nogle TestMonitorRunnables, som overvåger kritisk data i unit'en mens testene køres, f.eks. ved at teste på en invariant.

GroboTestingJUnit er ikke interessant for dette projekt, da det ikke understøtter interaktion mellem flere JVM processer. Dets muligheder kunne dog være interessante et sted imellem unit tests og distribuerede unit tests for at stressteste units der understøtter intensiv multitrådet adgang til data, såsom synkroniserede collections.

Det er svært at klassificere denne testarkitektur, da målet ikke er at opnå determinisme i testen men i stedet at opnå svigt gennem stress. De enkelte

TestMonitorRunnables kunne eventuelt ses som distribuerede testere, der ikke kommunikerer. En JUnit TestCase binder elementerne sammen og denne testcase kan betragtes som TCO.

6.5.2 SysUnit

SysUnit, som er et opensource projekt, blev fundet ved søgning på internettet. Beskrivelserne som findes, lover understøttelse af test af multitrådede programmer og på tværs af JVM'er. Deres projekthjemmeside (www.sysunit.org) er nede, men et source repository blev fundet andetsteds. Seneste rettelse ser ud til at være foretaget i 2006. Der foreligger ingen dokumentation af frameworket og der er ingen javadoc i koden.

Undersøgelser af koden tyder på at frameworket har noget til fælles med disunit rent funktionalitetsmæssigt. Det ser ud til, at SysUnit kan starte en Master maskine op og en række Slave maskiner og sende kommandoer mellem dem, men det er svært at afgøre præcist hvordan uden dokumentation.

Det er derfor ikke muligt at klassificere testarkitekturen med sikkerhed, men umiddelbart kunne eksistensen af Master maskine og Slave maskiner tyde på et sammenfald med, hvordan disunit klassificeres (se afsnit 7.3.1).

6.5.3 DisUnit

[Nicolescu et. al] omtaler dette værktøj, som i muligheder lyder til at svare til disunit, der tilfældigvis lyder samme navn, bortset fra at dette værktøj skrives med stort D og U. Værktøjet er ikke open source, men ifølge [Nicolescu et. al] kan en trial version hentes på en navngiven internet adresse, som desværre ikke fungerer. E-mail adressen fra artiklen blev kontaktet, men den blev sendt retur med modtager ukendt. Derefter blev institutionen, AIST [6], kontaktet, men de svarede heller ikke.

Udfra [Nicolescu et. al] ser det ud til, at arkitekturen indeholder rene distribuerede testere, og at der er mulighed for kommunikation imellem testerne.

6.5.4 Pisces

Pisces [Pisces] giver mulighed for at starte en række RemoteTestRunners på forskellige maskiner, hvorefter man kan bede de enkelte maskiner om at udføre tests og tilbagerapportere resultatet, transparant for en controller (ikke eksplicit implementeret), der delegerer testene. Pisces udgør grundlaget for disunit og er gennemgået kort i afsnit 7.2 og i detaljer i afsnit 15.1.

Controlleren i Pisces betyder, at der er en vis grad af central styring af testforløbet, hvilket umiddelbart peger mest på en global tester. Der er dog mulighed for at køre tests på maskiner i parallel, hvilket giver distribuerede testere. En mulighed for tests til at kommunikere udover gennem controllerens delegering eksisterer ikke.

[6] National Institute of Advanced Industrial Science and Technology (AIST) AIST Tsukuba Central 2, Tsukuba, Ibaraki, 305-8568, Japan

7 Pisces og disunit

Dette afsnit introducerer JUnit og beskriver kort opbygningen af Pisces, som disunit bygger på, samt disunit og dets integration til Termas T-Core system. Et overordnet diagram over klassesammenhænge mellem Pisces, disunit og T-Core integrationen er afbilledet i figur 3.

For en mere detaljeret arkitektur og uddybing af Pisces, disunit og T-Core integrationen henvises til afsnit 15.

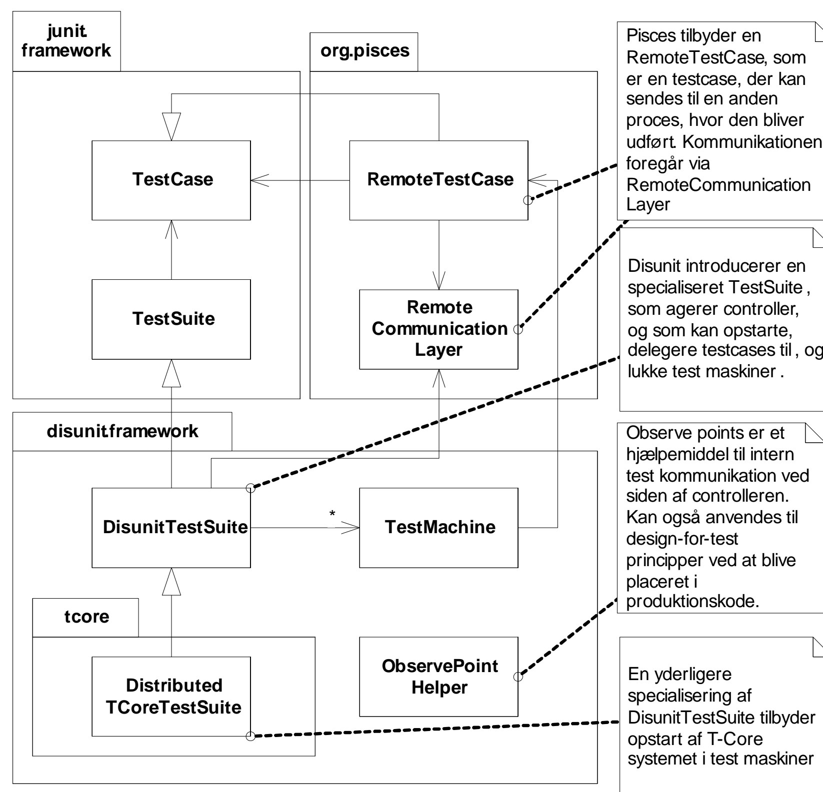

Figure 3: Overordnet klassediagram over afhængigheder mellem Pisces, disunit og T-Core integrationen

7.1 JUnit

Da Pisces bygger på JUnit 3.x introduceres her kort de begreber, som JUnit 3.x opererer med.

For at lave en test i JUnit, nedarves fra junit.framework.TestCase. I den specialiserede klasse defineres en test metode for hver enkelt test, der ønskes kørt. Kravet til testmetoderne er, at de ikke tager parametre og starter med "test" i metode navnet.

Hvis der ønskes noget fælles opstarts kode i to test metoder, er der mulighed for at definere TestCase'ns fixture gennem TestCase's setUp() metode, der køres inden hver test metode. Ligeledes eksisterer der en tearDown() metode, som køres efter hver test metode. Hver test metode kører desuden i sin egen instans af den specialiserede TestCase.

Et simpelt eksempel på en TestCase er StringTestCase fra figur 4, som tester metoderne length() og charAt() i String. setUp() metoden sørger for at sætte member field'et aString, som TestCase's assert...(...) metoder tester på. assert...(...) metoderne kaster exceptions som repræsenterer svigt, hvis asserten ikke holder eller hvis andre problemer opstår, såsom NullPointerExceptions.

Bemærk at der er en kollision ved begreberne svigt og fejl, som blev defineret i afsnit 6.1, og JUnits begreber "failure" og "error". JUnit kalder det en "failure" (svigt) hvis uhensigtsmæssigheder opfanges ved brug af en assert...() metode og en "error" (fejl) hvis de opfanges ved at koden under test kaster en (uventet) exception. Begge disse situationer er ifølge teorien svigt, da de er blevet observeret som afvigelser i forhold til det forventede resultat af testkørslen.

Typisk vil en "failure" i JUnit indikere en defekt, der manifesterer sig som en funktionalitetsmæssig afvigelse i runtime udførslen af koden under test, mens en "error" i JUnit kan indikere en defekt i testkoden eller koden under test - typisk som en forglemmelse i form af NullPointerExceptions eller andre runtime exceptions.

```
import junit.framework.TestCase;

public class StringTestCase extends TestCase {

    private String aString;

    protected void setUp() throws Exception {
        super.setUp();
        aString = "Test string";
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testLength() {
        assertEquals(11, aString.length());
    }

    public void testCharAt() {
        assertEquals('s', aString.charAt(2));
    }
}
```

Figure 4: Eksempel på en test i JUnit 3.x

Hvis flere TestCases ønskes grupperet for at blive kørt sammen, kan en TestSuite instantieres, hvor TestCase klasserne kan tilføjes. Til TestSuites kan også

TestSuites tilføjes, så grupperinger igen kan grupperes osv.

7.2 Pisces

Pisces [Pisces] er et open source framework, som bygger på JUnit 3.x og kan benyttes til at udvikle distribuerede unit tests.

Pisces blev valgt af praktiske årsager. Det er simpelt og har en stærk abstraktion i form af sin definition af en "remote" TestCase, som er en specialisering af en almindelig JUnit test case, der er i stand til at blive udført af en såkaldt RemoteTestRunnerAgent, der kører på en test maskine.

Se nedenstående afsnit under bilag for detaljerede beskrivelser af følgende:

- Afsnit 15.1.1: Implementation af maskine-begrebet
- Afsnit 15.1.2: Implementation af JUnit TestCase, som kan køres på en maskine.
- Afsnit 15.1.3: Deployment af klassefiler ved kørsel af Pisces tests
- Afsnit 15.1.4: Oversigt over uhensigtsmæssigheder.

7.3 Disunit

Disunit er bygget ovenpå Pisces for at gøre det nemmere for testudvikleren at anvende Pisces. Disunit introducerer en decideret testcontroller, som kan starte og stoppe TestMachines, en abstraktion der reflekterer begrebet maskine, som tests kan udføres på. Dermed retter disunit op på de uhensigtsmæssigheder i Pisces, som listes i afsnit 15.1.4.

Se nedenstående afsnit under bilag for detaljerede beskrivelser af følgende:

- Afsnit 15.2.1: Udvidelser i forhold til Pisces
- Afsnit 15.2.2: Konkret implementation af controller
- Afsnit 15.2.4: Controllers muligheder for at delegere tests til maskiner

7.3.1 Klassifikation af disunits arkitektur

Disunit implementerer specifikt en test controller, som delegerer tests til maskinerne. Maskinerne fungerer som distribuerede testere, men kører som udgangspunkt ikke fuldstændig autonomt, da de er bundet af controllerens delegering.

Der er dog ingen hindring for, at testcontrolleren blot igangsætter en række testere parallelt, som derefter vil køre som rene distribuerede testere. Dog er der ingen implementation af inter-test kommunikation som udgangspunkt i disunit bortset fra, hvad testcontrolleren muliggør gennem delegering af tests.

Overordnet klassificeres disunit som en testarkitektur med distribuerede testere, da dette er den mest generelle struktur.

7.4 T-Core disunit

Ovenpå disunit blev der lavet en udvidelse, som kan starte Termas T-Core system op på test maskinerne i disunit, og dermed give test udvikleren et harness, som fungerer i kontekst af T-Core systemet.

Se nedenstående afsnit under bilag for detaljerede beskrivelser af følgende:

- Afsnit 15.4.1: Kort introduktion til T-Core.
- Afsnit 15.4.2: Udvidelser i forhold til disunit og integration i T-Core.
- Afsnit 15.4.4: Implementation af distribueret observe point (se også afsnit 7.5)

7.5 Observe points

Observe points, som er en måde for tests at kommunikere indbyrdes og som derfor er en del af en tests TCP, introduceredes i disunit.

> observe point: Et observe point er et bestemt sted i enten test- eller produktionskode, hvor testeren ønsker at kunne observere en tilstand eller modtagelsen/afsendelsen af data. Testen bliver notificeret, når observe pointet kaldes. Såfremt et observe point er placeret i DUUT, er der tale om en design for test teknik. Et observe point deltager i testens TCP.

Ved hjælp af observepoints kan tests til dels sætte sig til at lytte på input fra andre tests, eller afgive input til andre tests, som lytter.

Observe points kan også anvendes til design-for-test principper, hvis de bygges ind i produktionskode.

Se afsnit 15.3 under bilag for en teknisk beskrivelse af observe points og en introduktion til de timingsmæssige udfordringer, som eksisterer ved deres brug.

7.6 Eksempler på tests i disunit

For at illustrere mulighederne og principperne i disunit, er der lavet et eksempel på en distribueret unit samt tre eksempler på test af denne unit.

7.6.1 Chatsystem

Eksemplet på en distribueret unit består af en chatserver og en chatklient. Chatserveren stiller sig ved opstart til at lytte på en TCP/IP socket, hvor den modtager requests til at deltage i chats. Hvis en chat klient connecter til denne socket, spawner chat serveren en tråd til at tage sig af kommunikationen med den nye klient. Når en chatklient sender en chat (i form af en streng), sendes denne chat til alle chat klienter, undtagen afsenderen selv. Chatklienterne udskriver modtagne chats på standard error.

7.6.2 Design for test

Der er tre eksempler på distribueret test af chatsystemet, og de anvender i tiltagende grad mekanismer, som disunit stiller til rådighed, for at få mere og mere kontrol over chatsystemet. Dvs. at der i sidste test er en højere grad af controllability og observability end i de to foregående.

Figur 5 viser, hvorledes ChatServer og ChatClient anvender LocalObservePointHelper (implementation af ObservePointHelper, som fungerer indenfor én enkelt proces) til at gøre opmærksom på interne events. Dermed er DUUT designet for test. TestCases anvender ObservePointListenerAdapter til at lytte på DUUTs interne events.

Alle tre tests er i princippet struktureret på samme måde. De laver tre testmaskiner og består af følgende trin:

1. Start chatserver i maskine "chatserver"
2. Start chatklient 1 i maskine "chatclient"
3. Start chatklient 2 i maskine "chatclient2"
4. Send en tekstbesked fra chatklient 1
5. Verificer, at tekstbeskeden modtages på chatklient 2
6. Verificer, at der ikke modtages en tekstbesked på chatklient 1
7. Stop chatserver

Trinene kører sekventielt, bortset fra 5. og 6., der kører parallelt.

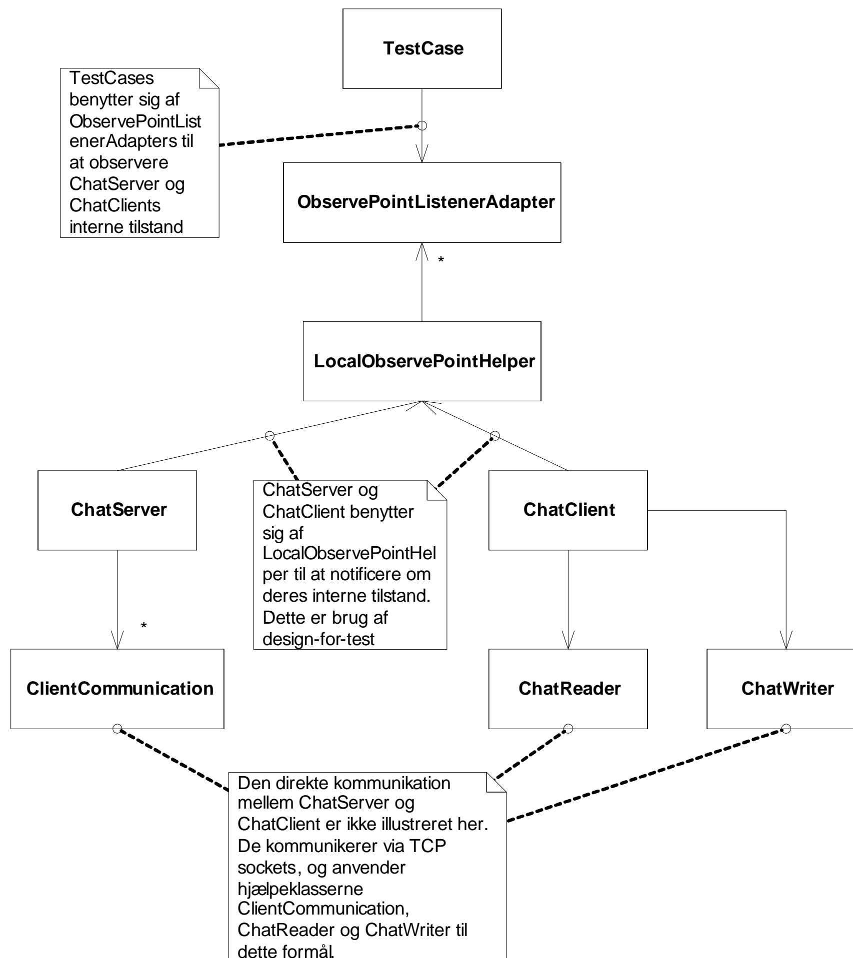

Figure 5: Module view: Klassediagram over chatsystemet og dets tests

7.6.3 FirstTestSuite

Denne første test benytter sig slet ikke af observe points til at lytte på DUUTs interne tilstande. I stedet benytter den sig af sleep() kald, som er en potentielt fejlagtig og langsom mekanisme. Koden kan ses i kildekodefilen /disunit/src/test/disunit/framework/FirstTestSuite.java.

FirstTestSuite benytter sleep() kald for at vente på data i punkt 5. og 6. (jævnfør afsnit 7.6.2). Der er flere problemer med denne tilgangsmetode:

1. Uanset hvad der sker, kommer testen til at vente i det specificerede tidsrum, uanset om data modtages tidligere
2. Hvis der er lag på dataoverførslen, risikerer man ikke at modtage data, med mindre tidsrummet sættes endnu mere op (selvom der naturligvis

aldrig er en garanti)

3. Chatklient 2 startes i en separat tråd i punkt 3. Hvis det skulle ske, at denne tråd ikke får køretid før efter chatklient 1 har sendt data, kan det risikeres, at data aldrig modtages i testen. Dette er essentielt en ændring af testforudsætningerne, hvor det ønskes testet, om chatklient 2 modtager data fra chatklient 1, såfremt de begge kører.

7.6.4 SecondTestSuite

Den anden test forsøger at rette op på problemerne fra den første ved at benytte sig af ObservePointListeners på DUUT's interne tilstand. I relation til FirstTestSuite's problemer, løser SecondTestSuite dem således:

1. Der angives på ObservePointListeners anticipate() metode et tidsrum, men det er kun et maksimalt tidsrum. Såfremt data modtages tidligere, ventes der ikke længere.

2. Tidsrummet kan roligt sættes op, på grund af punkt 1. Så fremt testen fejler i dette punkt, vil det dog resultere i en længere testeksekveringstid.

3. Dette problem kan stadig opstå i SecondTestSuite.

Koden kan ses i kildekodefilen /disunit/src/test/disunit/framework/SecondTestSuite.java

7.6.5 ThirdTestSuite

Den tredje og sidste test indfører et ekstra trin til listen fra afsnit 7.6.2. Efter trin 4 er der tilføjet en test, som via en ObservePointListener venter på, at serveren har afsendt en besked til en klient. Dette punkt kan kun nås, såfremt klienten er chatklient 2, eller der er en defekt i DUUT (hvis svigt vil blive observeret senere).

Desværre løser denne test ikke problem 3, som stadig kan opstå. For at gøre testforløbet fuldstændig deterministisk, er det nødvendigt at indføre observe points, når chatserveren og chatklienterne er klar til at modtage og afsende data. Der skulle ventes på disse observepoints i punkt 1, 2, 3 således, at testen aldrig går videre, med mindre den netop startede chatserver eller -klient helt sikkert er klar.

Eksemplet illustrerer kompleksiteten i at teste selv simple distribuerede units. Det sker erfaringsmæssigt ofte, at sådan potentiel non-determinisme bliver overset af testudvikleren, og et deraf følgende svigt opstår måske kun i meget sjældne tilfælde, hvilket gør det svært at gennemskue, om DUUT eller selve testen har defekter.

Koden kan ses i kildekodefilen /disunit/src/test/disunit/framework/ThirdTestSuite.java

8 Status

Disunit har i to år kørt som testframework i Termas T-Core udviklingsafdeling og i kortere tid i et andet projekt.. Dette afsnit opsummerer erfaringerne med det distribuerede unit test framework og kommenterer den nuværende status.

8.1 Erfaringer

8.1.1 Anvendelsesområder

Disunit blev primært udviklet til at hjælpe udviklerne med at genskabe svigt, som brugere af systemerne rapporterede. Svigtene er typisk relateret til en bestemt T-Core komponent, og blev tidligere genskabt manuelt på et testsite. Det fandt dog andre anvendelsesmuligheder og de samlede anvendelsesområder er listet herunder med hyppigst anvendelse først:

1. Genskabelse af brugerrapportede svigt: Udvikleren laver en disunit test som genskaber et svigt i en T-Core komponent. Når defekten er fundet og rettet, inddrages disunit testen som regressionstest. Disse tests igangsættes manuelt af udvikleren.

2. Understøttelse ved nyudvikling: Udover traditionelle unit tests, skriver udvikleren disunit tests, som tester nyudviklet funktionalitet i en eksisterende eller ny T-Core komponent. Disse tests inddrages i regressionstest og igangsættes manuelt af udvikleren.

3. Erstatning af manuelle tests: Visse dele af den manuelle test af T-Core komponenterne, som blev gennemført ved manuelt at udføre procedurer på et testsite, er erstattet af disunit tests, som igangsættes manuelt af udvikleren før release.

4. Stress/langtidstest: Et dedikeret testsite med klient- og servermaskiner stresser døgnet rundt T-Core systemet ved at udføre tilfældige disunit tests. Dette brugsmønster anvender real distribution.

5. Enkelttests på testsites: Disunit tests udføres manuelt på et testsite, primært for at teste nyudvikling af et distribuerede, kritiske kommunikationskomponenter. Dette brugsmønster anvender real distribution[7].

8.1.2 Resultater

I relation til ovenstående anvendelsesområder, oplistes her resultaterne, som er nået ved brug af disunit:

1. Disunit har vist sig som et stærkt værktøj til genskabelse af brugerrapportede svigt. Det er som oftest muligt at genskabe dem via disunit og det giver i regressionstesten udvikleren en større sikkerhed i, at eventuelle rettelser i eksisterende funktionalitet ikke indfører nye defekter i systemet.

[7] Real distribution er navnet på den kørselmetode, hvor maskinerne faktisk kører på fysiske separate maskiner. Se afsnit 15.1.4 for en udvidet forklaring.

2. Ligesom i punkt 1) giver disunit tests udvikleren en større tiltro til, at rettelser ikke indfører nye defekter og at nyudviklet funktionalitet fungerer, som det skal.

3. Der spares tid under hver manuel tests, fordi visse dele nu er automatiske. Komplicerede manuelle tests, hvor servere, der foretager en fail-over procedure[8] og hvor systemets funktionsdygtighed testes efter en sekundær server har taget over, kan nu udføres automatisk og ved brug af meget kortere tid.

4. Stress- og langtidstests har været nyttigt til at fange memory leaks og faldende performance over tid, hvilket ikke testes i andre disunit test, der helst skal køre så hurtigt som muligt. Testene er løbende blevet udbygget for at dække mere og mere af systemet.

5. Nyudviklingen af et nyt kommunikationssystem er blevet lettet ved at kommunikationsmekanismerne kan gennemtestes semi-automatisk (igangsættes manuelt) og intensivt på kort tid.

8.1.3 Fordele og ulemper

To udviklere, som har anvendt disunit, blev i forbindelsen med dette dokument spurgt om fordelene og ulemperne ved frameworket. Svarene er sammenfattet herunder:

Fordele:

- Understøttelse af distribuerede problemstillinger i unittests.
- Integration til JUnit framework, så disunit tests kan udføres gennem enhver TestRunner, f.eks. i IDE'et.
- Genkendelighed, i form af JUnit integrationen.
- Giver mulighed for at få udført hvad som helst på en maskine, ikke kun en test.

Ulemper:

- Høj indlæringskurve på grund af kompleksiteten i distribution.
- Tests tager lang tid at skrive
- Svært at gennemskue fejlmeddelelser, da alt, inklusive intern kommunikation er tests.
- Svært at udvide disunit med ny funktionalitet, da det er bundet op på den interne kommunikation med tests.

[8] Redundante, sekundære (inaktive) servere står klar i systemet, hvis primære (aktive) server svigter. Overtagelsen af rollen som primær server af en sekundær server kaldes fail-over

- De antal tests, der rapporteres som gennemført, inkluderer den interne kommunikations test, som intet har med den konkrete test at gøre. De interne tests gør testrapporter fra JUnit uoverskuelige for testudvikleren. Dette konkrete problem er anskueliggjort i figur 6, som er et eksempel på en testrapport fra JUnit efter kørsel af en disunit test.

- Ved svigt i disunits opstartsmetode er der lang ventetid på grund af kommunikationsprotokollen. Såfremt noget svigter under opstart observeres det ofte gennem en timeout på controlleren, selvom en test godt kunne fange den egentlige fejl.

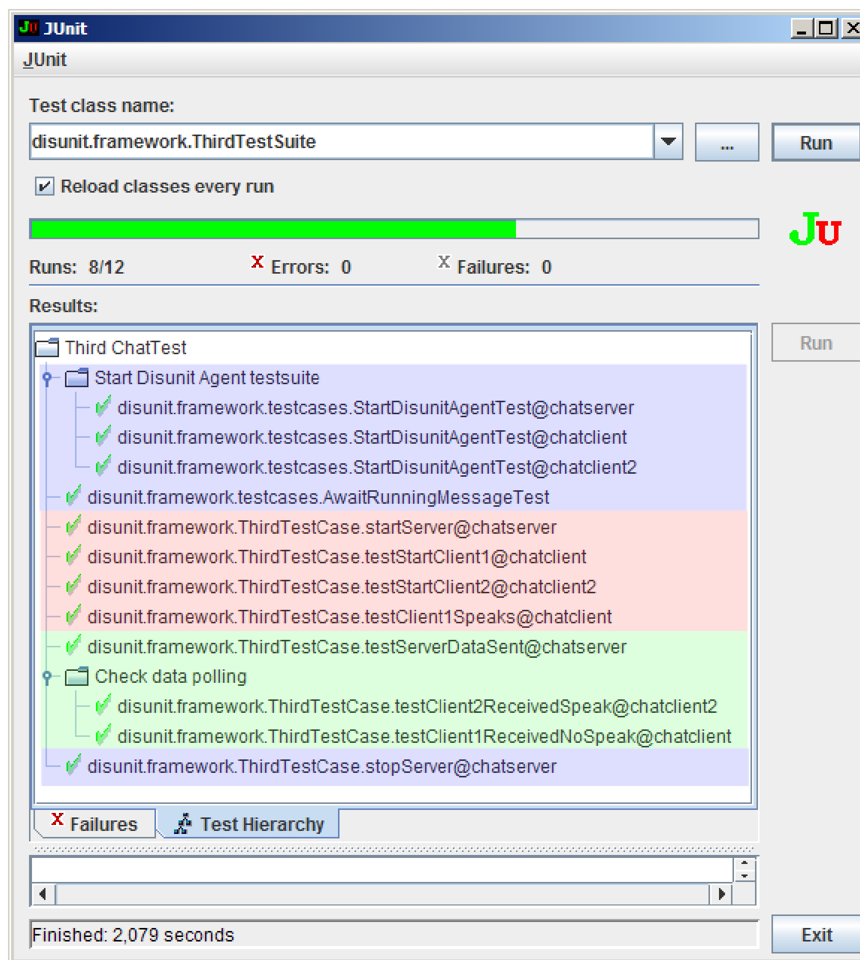

Figure 6: Dialogboks fra JUnit TestRunner efter kørsel af ThirdTestSuite fra eksemplet: Testene i de blålige sektioner er disunits/Pisces specfikke. De rødlige er test fixtures, mens de grønlige er faktiske tests.

8.1.4 Cost/benefit

Det er altid vigtigt at tage cost/benefit i betragtning, inden man laver automatiske distribuerede tests. Kodningen af testene er næsten lige så kompleks som kodningen af DUUT. Erfaringerne viser, at der ofte indføres defekter i de distribuerede tests. Visse tests kan køre grønt i flere kørsler i træk, hvorefter de pludselig kan observere svigt. Årsagen er ofte et observability problem, som er opstået i forhold til DUUT, der er blevet eksekveret i en timings-mæssig varians, der sjældent opstår.

Dette betyder, at der udover tiden til at udvikle testen, tit følger et vedligeholdelsesarbejde. Det tager også tid at sætte sig ind i testkoden igen, og derfor er det meget vigtigt at dokumentere, hvad testen gør i kodekommentarer. I unittests er der lagt op til, at udvikleren gerne må skrive grim kode, men det er en fordel at lade være i disunit.

Disse forhold bør tages i betragtning, inden man anvender disunit i en udviklingsproces.

8.1.5 Testfokus

Disunit test skrevet i T-Core gruppen har indtil videre været skrevet af DUUT udvikleren selv. Til dels har det været praktisk, da kodningen af distribueret unit test som oftest kræver et større kendskab til DUUTs indre struktur (whitebox testing).

Desuden har fokus på testene været på test af krav. Testene har således som udgangspunkt ikke været særlig destruktive, og det har været op til hver enkelt testers temperament at forsøge at få testene til at dække grænseområder (f.eks. ved brug af boundary testing og path testing[Bruegge & Dutoit, s454-459]). Typisk har det været indrapportering af svigt fra brugere, som har ledt til mere destruktive disunit tests. Derved kunne man overveje, om testene bør drives i en mere destruktiv retning, da defekter ofte findes her. Igen er det et spørgsmål om cost/benefit, da udviklingen af disse tests er dyre.

8.2 Disunit i forhold til eksisterende frameworks

Det er svært at forholde disunit til de eksisterende frameworks, der er beskrevet i afsnit 6.5, da der ikke foreligger dokumentation på anvendelsen af dem på konkrete projekter. DisUnit [Nicolescu et. al] omtaler test af HORB, et netværks middleware produkt, men beskriver ikke eventuelle problemstillinger, som blev mødt under testene. Derfor vil videreudviklingen af disunit basere sig på de erfaringer, som Terma A/S har gjort i de konkrete projekter, hvor disunit er blevet anvendt.

Disunit er i realiteten en prototype, som er gået i produktion. Fordelene er store for udviklerne, men som beskrevet i afsnit 8.1.3 er der også store ulemper forbundet med disunits knopskydning fra prototypestadiet. Ulemperne danner grundlag for de videreudviklingsmuligheder, der ligger i disunit og som er beskrevet i afsnit 9.

9 Videreudvikling af disunit

Med henvisning til ulemperne beskrevet i afsnit 8.1.3, kan udpeges en række problemer i disunit. Problemerne deles op mellem funktionalitet og arkitektur. Med arkitektoniske problemer menes arkitekturmæssige forhold, som begrænser implementationsmulighederne ([Bass et. al, s29]) i disunit og integrationen til T-Core. De beskrives ved brug af kvalitetsattributter som defineret i [Bass et. al, kap. 4], og løsningsforslag udarbejdes med udgangspunkt i taktikker som beskrevet i [Bass et. al, kap. 5].

De funktionelle problemer er mere umiddelbart mulige at løse uden at ændre i frameworkets grundlæggende strukturer.

9.1 Arkitektoniske udfordringer i disunit

9.1.1 Kobling til Pisces

Erfaringsmæssigt er langt fra alt, der udføres på en maskine i disunit tests. Som det også ses i eksemplerne beskrevet i afsnit 7.6, er formålet med mange af testmetoderne at skabe fixture for de reelle test metoder (som er repræsenteret ved setUp() metoden i JUnit, se afsnit 7.1). Det er ikke tilstrækkeligt med en setUp() metode, da dannelsen af fixture kan kræve koordineret indsats mellem testere.

Det bør være muligt at få udført andet end en test - f.eks. en kommando - på en test maskine gennem controlleren. Med besvær kan det lade sig gøre at kamuflere tests som kommandoer, men det får sideeffekter i form af at kommandoen indgår i selve testen (se afsnit 8.1.3).

9.1.2 Inter-test kommunikation

Det er svært for tests på test maskinerne at kommunikere. Kun gennem controllerens brug af sammensætningen af test suites (som beskrevet i afsnit 15.2.4), kan der synkroniseres mellem dem. I T-Core udvidelsen til disunit er det muligt gennem brug af T-Cores kommunikationskomponent (se afsnit 15.4.4), men inter-test kommunikationen bør være uafhængigt af systemet, der er under test.

9.1.3 Stejl indlæringskurve

Det tager tid at lære at skrive en disunit test. Kompleksiteten i at skrive testen er næsten så stor som at skrive den distribuerede unit. Typisk tager det flere forsøg at få skabt den rette fixture, fordi udvikleren overser steder, hvor der skal synkroniseres for at opnå determinisme.

9.2 Funktionelle udfordringer

9.2.1 Differentierede timeouts

Når Pisces beder en maskine om at udføre en test, venter den på svar. Det er muligt generelt at konfigurere en tidsperiode, før Pisces rapporterer et svigt

som følge af en time-out. Dermed bliver time-out end for alle tests ens, idet den længste timeout bliver gældende. Det bør være muligt at differentiere, således at tests eller kommandoer kan vurderes til individuelt at svigte indenfor et kortere timeout.

9.3 Løsningsmuligheder

For at hjælpe med at finde løsninger på de akitektoniske udfordringer fra afsnit 9.1, anvendes et kvalitesattributframework fra [Bass et. al]. Således knyttes hver udfordring sammen med en eller flere af frameworkets kvalitetsattributter, som hver beskriver en type arkitektonisk kvalitet. Dernæst identificeres hvilke taktikker, der kan anvendes for at afhjælpe de kvaliteter, som attributterne beskriver.

Tabel 1 lister de udfordringer, som er blevet identificeret i det gamle disunit og kobler dem sammen med en eller flere kvalitetsattributter. I tabel 2 er problemerne igen listet med den løsningsmulighed, der er valgt ud fra taktikkerne, som er beskrevet i [Bass et. al].

Med henblik på at anvende teknikkerne til at udvikle en ny version af disunit vælges prototyping. Erfaringerne fra disunit viser, at kan være mange afhængigheder, som ligger på et teknisk niveau, der bedst egner sig til at blive udforsket ved at skrive kode. Et prototypeforløb beskrives i afsnit 9.3.1.

Problem	Attribut	Kommentarer
Kobling til Pisces	Modifiability, Conceptual integrity	Det er besværligt at rette i disunit, både i selve frameworket og i tests skrevet til det. Modifiability angår prisen på ændringer [Bass et. al, s80], og det ville i den henseende gøre det billigere, hvis ændringer kan føres hurtigere igennem. Hele disunit bygger på Pisces' måde at kommunikere på. Det ville være en fordel for systemet som helhed, hvis ansvaret er mere bredt delt ud i stedet for at være bundet op på ét subelements funktionsmåde.
Inter-test kommunika-tion	Modifiability	Der er et behov for at testudvikleren nemmere kan styre testenes kommunikation og synkronisering og dermed lettere opnå determinisme i testen.
Indlæringskurve	Modifiability, Usability	Testudvikleren skal hjælpes så meget som muligt med hensyn til konfiguration af systemet.

Table 1: Problemerne i disunit og deres tilhørende kvalitetsattributter

Problem	Taktik	Kommentarer
Kobling til Pisces	Prevention of ripple effect: Hide information, Abstract common services, Restrict Communication Paths	For at undgå, at disunit kun kan kommunikere i termer af Pisces's tests, skal kommunikationen skilles ud i et separat modul og generaliseres, så det kan anvendes til generel kommunikation mellem testmaskiner og controller. Det fjerner også kompleksitet at reducere afhængigheden af andre modulers brug af Pisces' test kommunikation.
Inter-test kommunikation	Abstract common services	Ved indførelse af et separat, generelt kommunikationslag i forbindelse med løsningen af "Kobling til Pisces" vil der være etableret et grundlag for at anvende dette til den DUUT-uafhængige inter-test kommunikation. Dette vil gøre det muligt for testudvikleren at synkronisere mellem testmaskiner.
Indlæringskurve	Hide information, Support System/User initiative	Det er svært at gøre noget ved kompleksiteten med at udvikle distribueret/multitrådet kode i Java. Udvikleren kan dog hjælpes ved at gøre konfigurationen af disunit så simpel og kortfattet som mulig, samt ved at gøre brug af design-for-test principper i DUUT. Inter-test kommunikationen kan hjælpe med at indføre sidstnævnte. Desuden vil et separat kommunikationslag gøre systemets feedback nemmere at forstå og disunit kan i visse tilfælde hurtigere stoppe ved svigt eller ved forespørgsel fra testudviklerens kode

Table 2: Problemerne i disunit og taktikkerne til at fjerne dem

9.3.1 Anvendelse af prototyping

Som tidligere nævnt, er disunit faktisk en prototype, som er blevet anvendt i produktion. I stedet for at videreudvikle på den eksisterende kode, vil fokus blive rettet imod en ny ansvarsfordeling på nye subelementer af disunit, inspireret af problemerne og taktikkerne som blev beskrevet i tabel 1 og 2.

Der vil derfor blive startet forfra og udviklet en ny prototype, hvis formål er at udforske og afteste, at de interfaces, der defineres mellem subelementerne, holder i praksis. Processen for udviklingen af den nye prototype svarer til beskrivelsen i [Christensen], hvor følgende begreber er anvendt:

- AP: Arkitektonisk prototype.
- Super-AP: Prototype med identificerede arkitektoniske udfordringer, der udforskes ved hjælp af en sub-AP.

- Sub-AP: Prototype, hvor en bestemt udfordring i en super-AP udforskes.
- Code/Knowledge harvesting: Proces, hvor arkitektonisk vigtig kode eller viden trækkes ud fra en super-AP til brug i en sub-AP
- Code/Knowledge retrofitting: Proces, hvor arkitektonisk vigtig kode eller viden sættes ind i en super-AP fra en sub-AP

Den nye prototype dannes hovedsageligt ved hjælp af knowledge harvesting, hvor erfaringerne fra den første disunit (i realiteten den hierarkisk højest placerede super-AP), bruges som grundlag for at udforske det arkitektoniske udgangspunkt, som blev defineret i tabel 2. Der anvendes også en smule code harvesting, men det er så lidt, at den nye prototype reelt er en reimplementation.

9.4 Prototypeforløbet

Forløbet med udviklingen af prototyper er vist i figur 7. Som udgangspunkt forelå det "gamle" disunit, hvorfra viden (knowledge) blev anvendt til at skabe et første udkast til en ny disunit-prototype. Ideen med denne prototype var at sikre, at delkomponenterne i disunit blev delt fornuftigt op. Det første udkast til disunit-prototypen ses i figur 8.

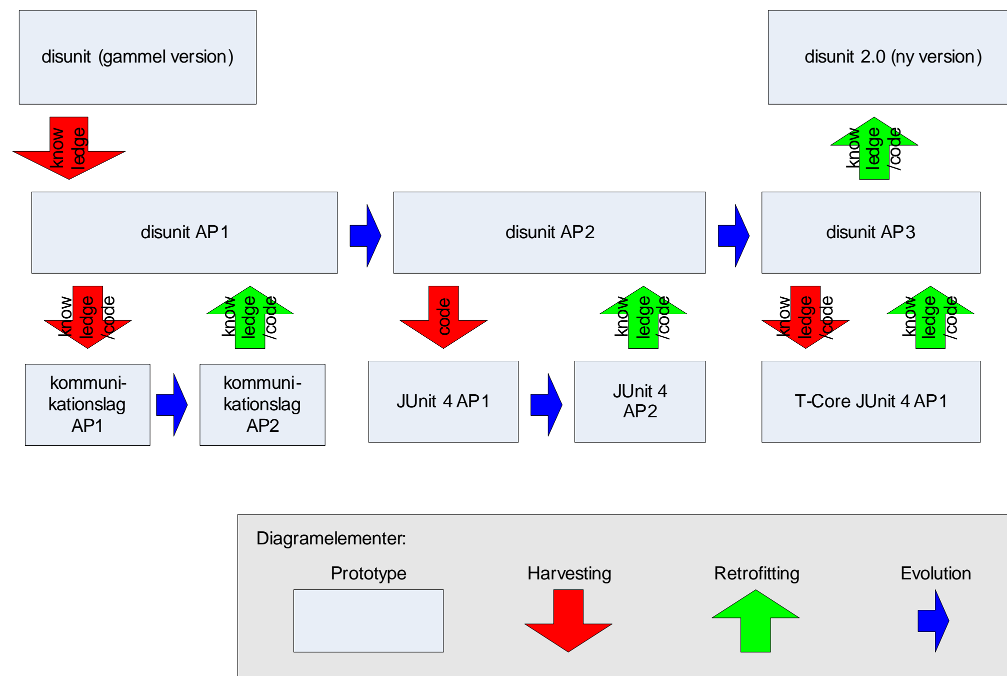

Figure 7: Illustration af prototypeforløbet med anvendelse af diagramtype og begreber fra [Christensen]

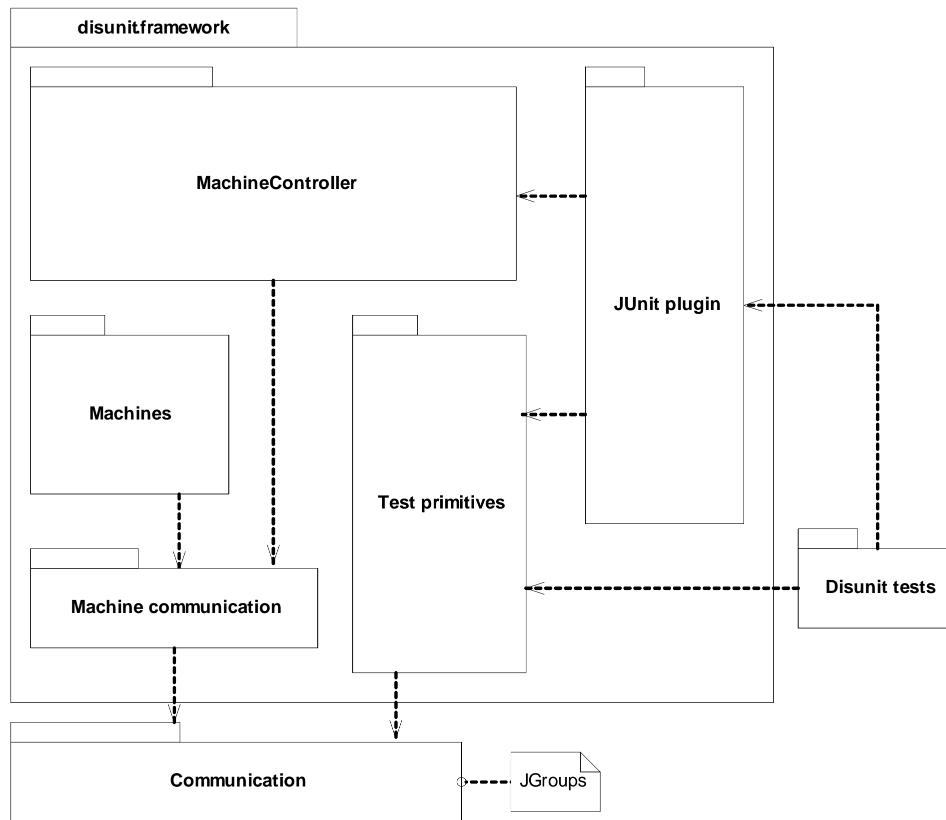

Figure 8: Første udkast til prototype, som blev implementeret i Interfaces AP 1

9.4.1 Subprototype, kommunikationslag

En af de største forandringer i figur 7 er, at kommunikationslaget (i figur 7 benævnt "Communication") er skilt helt ud fra disunit frameworket. Erfaringer fra det gamle disunit viste, at kommunikationsdelen (oprindeligt Pisces') krævede en del vedligeholdelse og desuden havde det en række defekter. Derfor blev der lavet forsøg med et tredjepartsprodukt, JGroups, som er i stand til at levere pålidelig kommunikation på en måde, som passer godt ind i disunit (se afsnit 10.2.1 om JGroups).

Samtidig med afprøvingen af JGroups blev interfaces til maskiner og controller rettet op og afprøvet i en let implementation.

9.4.2 Subprototype, JUnit 4

Det gamle disunit anvendte JUnit 3.x, men i flere år har JUnit 4.x været tilgængelig. JUnit 4.x benytter sig i høj grad af annotations til at beskrive tests og det skulle afprøves, hvorledes disunit kunne anvendes med den tankegang.

Kunne konfigurationen af testsuites f.eks erstattes af annotations? JUnit 4.x skulle desuden integreres som et plugin i stedet for, at disunit afhang af JUnits paradigme, som det var tilfældet i det gamle disunit.

9.4.3 Subprototype, T-Core JUnit 4

Formålet med denne subprototype var at sikre, at de krav, som et kompliceret system udviklet uafhængigt af disunit stiller til et testframework kunne opfyldes. Fremgangsmåden var her at forsøge at basere integrationen på en plugin-tankegang og udnytte så meget som muligt af subprototypen til JUnit 4. Desuden blev der her lavet forsøg med at implementere understøttelse af observe-points ved brug af JGroups, da den tidligere implementation baserede sig på en T-Core komponent.

9.4.4 Endelig retrofitting

Efter prototypeforløbet, som var afbilledet i figur 7, forelå der en på visse måder anderledes opdeling af delkomponenterne. Det endelige snit mellem interfaces/-pakker er vist i figur 9 og blev som sådan retrofittet ind i den nye version af disunit.

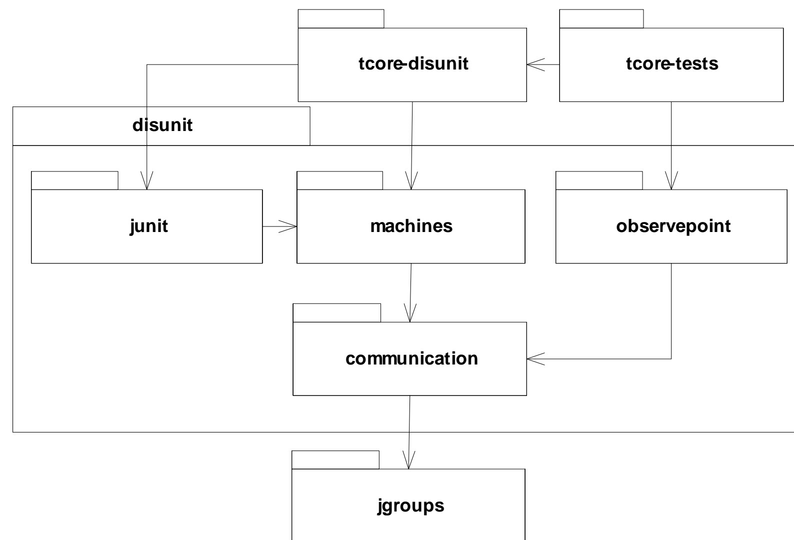

Figure 9: Endelig opdeling af pakker og interfacesnit som blev retrofitted til det endelige disunit

10 Disunit 2.0

Den nye arkitektur er naturligvis mere end den pakkeopdeling, som er vist i figur 9. Disunit er blevet totalt reimplementeret, og helt nye strukturer er dannet ud

fra den nye prototype, der er grundlag for disunit 2.0 [9]. Denne sektion beskriver først hvordan tests skrives i disunit 2.0 og derefter beskrives den overordnede ansvarsfordeling. For en detaljeret beskrivelse af arkitekturen, se afsnit 16.

10.1 Tests i disunit 2.0

Dette afsnit beskriver, hvordan distribuerede unit tests skrives i disunit 2.0.

10.1.1 JUnit 4

I modsætning til disunit, anvender disunit 2.0 i sin JUnit integration en notation, som ligner JUnit 4 [JUnit 4]. Dvs. at en tests opsætning beskrives ved hjælp af Java Annotations, som er syntaktisk metadata, der kan tilgås runtime. Figur 10 viser en JUnit 4 version af JUnit 3 testen fra figur 4.

```
public class StringTest {

    private String aString;

    public StringTest() {
    }

    @Before
    public void setup() throws Exception {
        aString = "Test string";
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testLength() {
        Assert.assertEquals(11, aString.length());
    }

    @Test
    public void testCharAt() {
        Assert.assertEquals('s', aString.charAt(2));
    }
}
```

Figure 10: Eksempel på en test skrevet i JUnit 4. Testen er den samme som JUnit 3 testen fra figur 4. Bemærk, at ingen nedarving er nødvendig, hvilket mindsker koblingen mellem tests og test framework. Desuden er navnekonventionen i testmetoder erstattet af @Test annotation, og de nedarvede fixture metoder er erstattet af @Before og @After annotations.

[9] Fremover kaldes den videreudviklede disunit for "disunit 2.0", mens den gamle version kaldes for "disunit"

10.1.2 Disunit 2.0 og JUnit 4 notation

Disunit 2.0 følger JUnit 4's annotation-metode til at beskrive tests og udvider JUnit 4's annotations med følgende nye annotations:

> @BootMachines: Beskriver en liste af navne på maskiner, som er påkrævet for at køre denne test
>
> @Composition: Beskriver kørselsmetoden via default attributten "value", som enten kan være sekventiel eller samtidig. Der er her mulighed for at angive en yderlige attribut "classes", som angiver en række @Composition annoterede klasser. Dermed er det muligt rekursivt at kombinere kørselsmetoder.
>
> @Remote: Angives på metoder, som skal køre på en (test)maskine. Attributten "machine" angiver den specifikke maskine.

Eksemplet i figur 11, som er alt, der skal til for at lave en distribueret test i disunit 2.0, anvender først JUnits @RunWith annotation til at give frameworket besked om, hvilken Runner implementation, testen skal køres med. Her angives DisunitRunner, som er en del af disunit 2.0s JUnit udvidelse. Dernæst defineres behovet for 2 maskiner, "client" og "server" via @BootMachines annotation, og der køres fem trin i sekvens (som beskrevet med @Composition):

1. testClient(): En test metode, som laver en System.out.println() på "client" maskinen.

2. testClientTimeout(): En test metode, som laver ved hjælp af timeout attributten en time out på en test metode.

3. testClientError(): En remote metode på client maskinen som udløser en NullPointerException. Metoden er ikke en test metode.

4. testClientTestError(): En test metode, som udløser en NullPointerException på client maskinen. Svarer til metode 3, men er en test metode

5. testServerFailure(): En test metode, som udløser et svigt.

Ved kørsel af denne disunit 2.0 test i Eclipse IDE, vises en dialog, som vist i figur 12. Bemærk, at der vises, at fire tests er kørt i alt, da metode nummer 3 ikke er en test. Der er opstået én fejl (metode 4), to svigt (metode 2 og 5) og en succes (metode 1). Metode 3 er ikke repræsenteret, da den ikke er markeret som test. Fejlen, som denne metode fremprovokerede er logget via Java Logging.

Forskellen til disunit dialogen fra figur 6 er tydelig, idet frameworkets interne kommunikation ikke er afspejlet i disunit 2.0, simpelthen fordi den ikke længere er implementeret som tests i sig selv. Dette giver testudvikleren en meget mere præcis testrapport, der ikke er viklet ind i rapporter, der stammer fra disunit 2.0's interne kommunikation.

Bemærk desuden hvor lidt kode, der skal til for at lave en distribueret test med to maskiner i disunit 2.0. Syntaksen er stærk og har et minimum af redundant information, som er koncentret omkring de steder, hvor det er relevant. Til sammenligning kan betragtes de disunit tests, som er vedhæftet i

kildekode i filerne /disunit20/src/test/disunit/framework/ThirdTestSuite.java (som beskriver en såkaldt test suite) og /disunit20/src/test/disunit/framework/ThirdTestCase.java (som indeholder selve testen). Det er sværere at navigere og se sammenhængene i disse gamle disunit tests.

Som et længere og mere realistisk eksempel er den sidste test af det lille chat-system fra disunit eksemplerne (se afsnit 7.6.5) blevet konverteret til disunit 2.0 og kan ses i den tilknyttede kode i filen /disunit20/src/test/disunit/framework/junit/ChatTest.java

```
@RunWith(DisunitRunner.class)
@BootMachines( { "client", "server" })

@Composition(value = Type.SEQUENTIAL)
public class SimpleDisunitRunnerTest {

    @Test
    @Remote(machine = "client")
    public void testClient() {
        System.out.println("Hello client");
    }

    @Test
    @Remote(machine = "client", timeout=2000)
    public void testClientTimeout() throws InterruptedException {
        System.out.println("Hello client timeout");
        Thread.sleep(2500);
    }

    @Remote(machine = "client")
    public void testClientError() {
        System.out.println("Hello client error");
        String string = null;
        string.charAt(3);
    }

    @Test
    @Remote(machine = "client")
    public void testClientTestError() {
        System.out.println("Hello client test error");
        String string = null;
        string.charAt(3);
    }

    @Test
    @Remote(machine = "server")
    public void testServerFailure() {
        System.out.println("Hello server failure");
        Assert.assertEquals("Didn't match", true, false);
    }
}
```

Figure 11: En simpel test skrevet i disunit 2.0

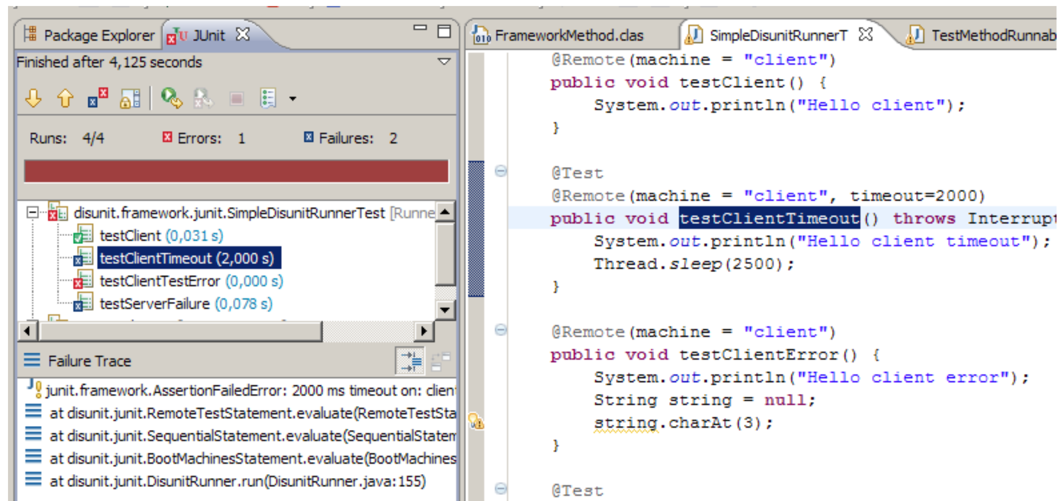

Figure 12: Screenshot fra Eclipse efter kørsel af en disunit 2.0 test. Til venstre ses testrapporten med de fire test metoder og deres resultat. Til højre ses test koden, markeret med det valgte test metode navn. Dialogboksen kan sammenlignes med outputtet fra det gamle disunit i figur 6, hvor en stor del bestod af rapporter om intern kommunikation.

10.2 Ansvarsfordeling

Pakkeopdelingen (figur 9) fordeler de væsentlige, overordnede ansvarsområder i disunit 2.0. For en detaljeret beskrivelse af arkitekturen, se afsnit 16.

10.2.1 JGroups

JGroups blev valgt til at tage sig af netværkskommunikationen i disunit 2.0. JGroups er open source under LGPL[10]. Det omtales som et toolkit til pålidelig multicasting og indeholder en hel del konfigurationssmuligheder, men de relevante punkter er listet herunder.

[10]Lesser GNU Public Licence

Funktionalitet	Beskrivelse
Gruppekommunikation	Klienter kan tilmelde sig grupper og kommunikere med andre klienter, enten direkte til enkelte medlemmer eller til alle.
Dynamisk discovery	Klienter kan tilmelde sig uden at være kendte på forhånd og uden at kende andre klienter
Detektion af crashes	Kan håndtere klienter, som crasher og notificerer andre klienter om andre klienters exit eller tilmelding
Understøttelse af tilstande	Klienter som logger på en gruppe har mulighed for at modtage en historik (el. tilstand) over beskeder sendt tidligere
Netværk	JGroups kan konfigureres til at fungere på mange forskellige netværksopsætninger

Helt overordnet opfylder JGroups de krav, der stilles til det uafhængige kommunikationlag, som beskrevet under "Kobling til Pisces" i tabel 2. Desuden hjælper de beskrevne funktionaliteter med at håndtere crashede maskiner, tilstande til brug i observepoints og forskellige netværksopsætninger, som især kan være relevant ved real distribution tests, der typisk kører på et dedikeret test-site. Dynamisk discovery af klienter reducerer konfigurationen, som en testudvikler belastes med.

Afprøvningen i prototypen viste, at JGroups fungerer stabilt og at det kan bruges til de formål, disunit 2.0 har. En finjustering af konfigurationsparametre kan foregå senere uden at det påvirker andre delelementer.

10.2.2 Communication

I stedet for at bruge JGroups direkte, anvendes et kommunikationslag ovenpå for det tilfælde, at JGroups skal skiftes ud. Communicationlaget afskærmer JGroups mere komplicerede funktionaliteter og leverer en simpel service til de delelementer af disunit 2.0, som har brug for kommunikation. Communication indeholder således en begrænset funktionalitet i forhold til JGroups: Det er muligt at lytte på beskeder og sende beskeder af typen Serializable, samt at lytte på hvem der forlader og kommer ind i en kommunikationsgruppe.

Formålet med Communication er at afkoble JGroups så meget som muligt fra disunit 2.0, så det er lettere at udskifte med et andet værktøj, såfremt der skulle opstå problemer med JGroups. Detaljerne omkring kommunikationslaget kan læses i bilagene i afsnit 16.1.

10.2.3 Machines

Machines delelementet leverer en testcontroller, kaldet MachineController, og har ansvar for at starte og stoppe maskiner, der spawnes i egne processer. Desuden kan Machines betragtes som endnu et lag ovenpå Communication elementet, idet det via MachineController tilbydes at sende RunnableMessages

til maskiner. RunnableMessages er beskeder, som implementerer Runnable[11]. Protokollen består i, at en RunnableMessage sendes (serialiseres) til en maskine, som udfører den som en Runnable, hvorefter beskeden serialiseres tilbage igen som svar.

Formålet med Machines er således at indkapsle al intern kommunikation mellem controller og maskiner samt at levere en generisk service til at få udført arbitrær kode på en maskine. Den generiske service er således en del af afkoblingen fra test-paradigmet, som Pisces tilbød, og dermed afhælper det problemet med "Kobling til Pisces", som beskrevet i tabel 2.

Detaljerne omkring maskin implementationen kan læses i bilagene i afsnit 16.1.

10.2.4 JUnit

Dette delelement indeholder funktionalitet til, at disunit tests kan køres igennem JUnit. Al JUnit specifik kode er isoleret her. Det kan diskuteres, om denne pakke er en del af disunit 2.0. I stedet kunne man betragte det som en plugin, da man lige så godt kunne lave et plugin med et andet test framework. JUnit er dog meget udbredt og de fleste IDE'er indeholder grafisk integration til kørsel af JUnit tests, hvilket denne plugin også understøtter.

Ved brug af JUnit 4.x annotation notation er der forsøgt at gøre testopsætning så simpel som mulig, hvorfor det afhælper den stejle indlæringskurve, som beskrevet i tabel 2.

Flere detaljer omkring JUnit 4 integrationen kan læses i bilagene i afsnit 16.2.

10.2.5 Observepoint

Observepoint elementet indeholder en implementation af DistributedObservePointHelper ved brug af Communication laget. Det er en simpel implementation, som kan udvides, men prototypen viste, at implementationen var mulig ved hjælp af Communication (der bruger JGroups). Senere kan man forestille dig at dette element kunne få et mere generelt navn, såfremt flere hjælpeværktøjer implementeres til disunit 2.0.

Formålet med Observepoints er at indføre den uafhængige inter-test kommunikation, som blev nævnt i tabel 2.

10.3 Klassifikation af disunit 2.0's arkitektur

Klassifikationen af disunits testarkitektur blev beskrevet i afsnit 7.3.1.

Disunit 2.0 svarer til denne klassifikation bortset fra, at disunit 2.0 indeholder en sand, uafhængig inter-test kommunikationsmekanisme gennem JGroups implementationen af observepoints. Dermed giver disunit 2.0 fuld support for synkronisering imellem testere, der i princippet kan holdes helt fri fra testcontrollerens styring, og dermed blive rene distribuerede testere.

[11] Standard Java interface for klasser som indeholder kode, der skal udføres i en separat tråd

10.4 Evaluering af disunit 2.0

Det er ikke muligt at nå at teste disunit 2.0 i praksis indenfor omfanget af dette projekt. Dog blev arkitekturen og det tekniske design som beskrevet i denne rapport, præsenteret for nogle af T-Core gruppens medlemmer: Arkitekten, en erfaren disunitudvikler og et nyere medlem, som også er erfaren udvikler fra andre projekter.

De delelementer, som blev gennemgået mest grundigt var junit og tcore-disunit delen, da de udgør testudviklerens direkte interface til disunit 2.0. For JUnit delen blev følgende bemærkninger noteret:

- Det er muligvis en uheldig ide at annotere test metoder i disunit 2.0 med JUnits @Test annotation, idet de egentlig ikke repræsenterer det samme. I JUnit står en @Test annoteret metode som oftest alene, mens den i disunit 2.0 ofte er afhængig af andre metoder, der opsætter data. Der blev talt om at omdøbe @Test til @TestStep og at indføre en @TestScenario annotation til at markere hele testcasen.

- De eksplicitte maskinnavne bør repræsenteres af enums eller på anden måde refaktoreres til at kunne indgå i IDE'ts typecheck, så stavemis-forståelser i maskinnavne undgås mellem @BootMachines annotation og @Remote annotation.

Bemærkningerne angående tcore-disunit omhandlede primært reduktion af konfigurationen i den enkelte test case. Der er en stor mængde annotations med mange konfigurationsparametre, og efterhånden som tcore-disunit bliver udviklet til at omfavne mere funktionalitet, kan de vokse til for udvikleren belastende størrelser. Diskussionen gik derfor på at identificere redundant information, som der er en del af i denne første prototype. Dette har dog ingen arkitekturmæssige konsekvenser.

De dele af konfigurationen, som ikke henfører direkte til opsætningen af testen, men som i stedet omhandler konteksten, såsom antallet af fysiske maskiner, operativ system, databaseimplementation til DUT osv., bør udskydes til runtime. Således kan tests skrives og udføres på både en enkelt fysisk maskine (typisk testudviklerens PC) og på et testsite til f.eks. regressionstest.

Et interessant spørgsmål, som blev stillet ved evalueringen, var: Hvorfor egentlig JUnit? Når disunit 2.0 nu ikke er unit testing som sådan, hvorfor så forsøge at presse et unit testing framework ned over? Spørgsmålet er uden tvivl relevant. JUnit blev valgt, fordi mange udviklere (i Terma A/S) er bekendt med dette framework og derfor virker springet til disunit 2.0 måske ikke så stort. Desuden er der integration til JUnit i de IDE'er, Terma A/S anvender (Eclipse og IntelliJ IDEA), og derfor kan disunit tests køres nemt, mens udvikleren arbejder. Men det meste af (kerne)funktionaliteten ligger i machines/observepoint/kommunikation pakkerne og er således totalt uafhængige af JUnit. Så man vil nemt kunne enten integrere til et andet test framework eller lave sin egen grænseflade senere, såfremt det skulle vise sig nyttigt.

11 Kildekode

I zip-filen sourcecode.zip, der følger med denne rapport, ligger kildekode og javadoc for såvel disunit som disunit 2.0. Kildekoden kræver J2SE version 1.6 eller senere for at kompilere.

Mappestruktur og -indhold er som følger:

```
/disunit        - Overordnet mappe til disunit filer
    /src        - Overordnet kildekode
        /main   - Kildekode
        /test   - Test kode
    /bin        - Pre-kompilerede binære klasser
    /lib        - Nødvendige biblioteker
    /doc        - Genereret javadoc

/disunit20      - Overordnet mappe til disunit 2.0 filer
    /src        - Overordnet kildekode
        /main   - Kildekode
        /test   - Test kode
    /bin        - Pre-kompilerede binære klasser
    /lib        - Nødvendige biblioteker
    /doc        - Genereret javadoc
```

Kildekoden er veldokumenteret og der er desuden genereret javadoc, som kan ses i en browser ved at gå ind i de to /doc mapper.

Det er nemmest at kigge på kildekoden ved brug af et IDE, som f.eks. Eclipse. Følgende trin skal gennemgås for at indlæse et af projekterne (disunit eller disunit20) i Eclipse (version 3.5):

1. Start Eclipse
2. Vælg File->New->Java Project
3. Vælg "Create project from existing source"
4. Vælg relevant overordnet mappe (/disunit eller /disunit20)
5. Tryk "Finish"

Testen af chatsystemet, som er beskrevet i afsnit 7.6.5, er konverteret til disunit 2.0 og kan køres ved at åbne klassen ChatTest (disunit20/src/test/disunit/framework/junit/ChatTest.java). Højreklik og vælg Run As->JUnit Test. Vælg "ChatTest" i dialogboksen.

Hvis eksemplet fejler, er det sandsynligvis fordi, der ingen loopback adapter er på netforbindelsen. JGroups er konfigureret til at kommunikere med multicast og kræver ved kørsel på én fysisk maskine, at den kan multicaste til sig selv. Testen kører desuden kun på Windows.

12 Konklusion

Jeg har gennem forarbejdet til dette projekt udviklet et framework, disunit, til test af distribuerede systemer, baseret på Pisces [Pisces] og JUnit 3.8.2 [JUnit 3].

Frameworket er blevet anvendt i Terma A/S i en periode, og har vist sig nyttig til automatisering af manuelle test og genskabelse af brugerrapporterede svigt.

Jeg har med reference til [Ulrich&König] præsenteret to arkitekturer for distribueret test, en med en global tester og en med distribuerede testere, og placeret disunit og de andre framework i denne referenceramme. Desuden har jeg redegjort for begreberne observabillity og controllability, som ofte optræder når der arbejdes med distribuerede testere.

I projektet har jeg undersøgt eksistensen af andre frameworks, som tilbyder test af distribuerede systemer, men har ikke fundet noget, som er i aktivt brug.

Jeg har gennemgået fordele og ulemper ved disunit. Disse bestod primært af tæt kobling af de distribuerede testeres kommunikation til et testparadigme og af afhængigheder til Terma A/S T-Core platform, som i praksis har fungeret som SUT[12]. Desuden har udvikling af test i disunit været konfigurationstungt, og det har været svært at lære.

På baggrund af de identificerede fordele og ulemper, lavede jeg et forslag til arkitektoniske ændringer af disunit ved hjælp af kvalitetsattributframeworket fra [Bass et. al].

Jeg implementerede og afprøvede ændringerne i en ny version af disunit, disunit 2.0, gennem et prototypeforløb inspireret af [Christensen]. Disunit 2.0 baserer sig på JUnit 4 [JUnit 4] og tilføjer stærke og kortfattede annotations, som kan beskrive distribuerede tests. Disunit 2.0 er kendetegnet ved et uafhængigt kommunikationslag, baseret på JGroups [JGroups] og generisk funktionalitet til instantiering af test controllers og test maskiner, samt ved at specifikke testframeworks (såsom JUnit) nu ligger helt udenfor den centrale funktionalitet, og er indbygget som plugins. Derved kan disunit 2.0 nemt integreres i andre testframeworks.

Gennem prototypeforløbet testede jeg disunit 2.0 på Terma's T-Core platform for at sikre, at der kunne udvikles integration til et større softwaresystem, der ikke er direkte forberedt til et bestemt testframework.

Udviklingsteamet på T-Core platformen evaluerede disunit 2.0 og havde en række forslag om minimering af konfiguration, nogle navnerettelser og mindre funktionelle ændringer. Værktøjet er modtaget med åbenhed og er valgt som standardværktøj til udvikling af automatiske distribuerede tests i T-Core. Der foreligger nu konkrete planer om at anvende disunit 2.0 til test af non-funktionelle krav. Desuden arbejdes der med at lave en større grad af automatisering i forbindelse med udførelsen af testene, herunder transparent udførelse på dedikerede test-sites.

I forhold til teorien og praktiske erfaringer beskrevet i denne rapport bør følgende være til stede i et framework til test af distribuerede systemer:

- Mulighed for distribuerede testere for at kunne teste parallelitet
- Mulighed for DUT-uafhængig kommunikation, der sikrer mod timingsmæssige forhold og som muliggør opnåelsen af højere controllability og observ-

[12] System Under Test

ability. Kommunikationen kan anvendes både til inter-test kommunikation og til design-for-test.

- Stærk, kortfattet syntaks til beskrivelse af tests.
- Implementation af specifik og testframework uafhængig controller, som er i stand til at starte maskiner med distribuerede testere og håndtere deres livcyklus.
- Minimal compile-time konfigurationsopsætning for at understøtte flytning af test fra enkelt fysisk maskine til flere fysiske maskiner (testsite). Konfiguration bør ske runtime uden ændringer i selve testbeskrivelsen.

13 Videre arbejde

Der er stadig en del arbejde, som skal udføres for at disunit 2.0 kan indgå effektivt i en praktisk udviklingsproces. Dette afsnit belyser kort nogle af områderne, der kunne undersøges

13.1 Real distribution

Real distribution, som nævnt i afsnit 15.1.4 er ikke implementeret i disunit 2.0. Forskellen mellem at køre lokalt og såkaldt "real" begrænser sig til opstarten af maskinerne, som skal ske manuelt i stedet for automatisk af disunit 2.0. Man kunne dog godt forestille sig, at et andet værktøj kunne tage sig af at starte maskinerne, og så kunne disunit 2.0 koncentrere sig om at udføre tests'ene uvidende om, hvem der starter maskinerne. I Terma A/S anvendes allerede et egenudviklet værktøj, som er i stand til at distribuere software til en række (fysiske) maskiner og holde dem opdateret med seneste version. Disunit 2.0 kræver, at alle maskiner har samme Java klasser i deres JVM's og det ville være oplagt at anvende dette værktøj til at distribuere klasserne, som testes.

Den ideelle situation ville være, at tests'ene nemt kunne køres både lokalt og "real" distribueret. I disunit 2.0's nuværende form ville der være lagt op til, at kørselsmetoden blev angivet ved hjælp af annotations, men det ville selvsagt ikke være særlig smart at skulle rette i kildekoden til tests'ene for at ændre kørselsmetode.

I den nærmeste fremtid vil der i T-Core gruppen i Terma A/S blive oprettet et forsøg med henblik på at kunne anvende forskellige kørselsmetoder på disunit 2.0 så nemt som muligt for testudvikleren.

13.2 Non-funktionelle tests

Det oprindelige disunit er tidligere blevet anvendt til stabilitetstest, hvor tilfældige tests blev udført på et kontinuerligt kørende system. I denne proces blev memory forbrug og systemloggen overvåget, og metoden viste sig effektiv til at finde memory leaks.

I T-Core gruppen i Terma A/S bliver der oprettet et forsøg med at anvende disunit 2.0 til non-funktionelle tests. Forsøget vil koncentrere sig om at afprøve værktøjets evne til at udføre performance og stabilitets test på dedikerede test sites, eventuelt installeret med forskellige operativsystemer.

13.3 Continuous integration

Det ville være oplagt at kigge på, hvordan disunit 2.0 kunne anvendes i forbindelse med continuous integration, hvor udviklingsmiljøet udfører test automatisk på kode, som udviklere checker ind i systemet, for at sikre at fejlagtig kode ikke lægges i versionsstyringssystemer.

I dette tilfælde ville det være af meget høj prioritet, at disunit test kunne udføres hurtigt. Opstart og nedluk af maskiner for hver test kan bidrage til et betragteligt overhead, især i udvidelser, hvis der som i T-Cores tilfælde skal startes instanser af forholdsvis tunge softwaresystemer for at DUUT kan køre, og som ikke er bygget til hurtig opstart.

Her ville det være interessant at kigge på, hvorledes disunit 2.0 kunne fungere på maskiner (evt. inklusive kørende softwaresystemer ovenpå), der kører kontinuerligt og som kan modtage og udføre tests on demand. Her kan ligge problemstillinger i fixtures, dvs. hvordan sikrer man at tests ikke påvirker hinanden med de data, de eventuelt lægger i de kontinuerligt kørende systemer.

Man kunne forstille sig at have en farm af maskiner stående, som var i stand til at køre tests (lokalt på hver maskine), så tests kunne udføres parallelt (se også afsnit 13.4).

13.4 Parallel unit testing

På grund af, tidsoverhead i forbindelse med udførelse af disunit tests, kunne det være interessant at se på, hvordan man kunne få dem til at køre i parallel på flere fysiske maskiner af gangen. Eftersom testene ikke (burde) afhænge af hinanden, er det ligegyldigt om de køres i sekvens eller parallel. JUnit 4.x skulle indeholde noget funktionalitet til at understøtte dette.

13.5 Automatisering

På flere områder kører disunit semi-automatisk. Selve udførslen af testen er automatisk, men igangsætning sker manuelt. Det ville være interessant at kigge på, hvordan disunit tests kunne køres automatisk, f.eks. i forbindelse med continuous integration. Da der allerede er en JUnit integration er det ikke utænkeligt, at f.eks. et nightly tester værktøj er i stand til at tage testene direkte.

14 Tak

I forbindelse med udvikling af disunit vil jeg gerne takke kollegaer fra Terma A/S, Integrated Defense Systems, Ground Software, specielt hele T-Core grup-

pen, som har investeret tid og interesse for at hjælpe mig med at udvikle disunit til et framework, der alsidigt kan anvendes til tests af distribuerede systemer.

References

[IEEE] IEEE Standard Glossary of Software Engineering Terminology, E-ISBN: 0-7381-0391-8, 10 dec. 1990

[Bruegge & Dutoit] Object-Oriented Software Engineering, Second Edition, Bernd Bruegge, Allen H. Dutoit, Pearson Prentice Hall, ISBN 0-13-191179-1, 2004

[Ulrich&König] Architectures for testing distributed systems, Chapter 7, Andreas Ulrich, Hartmut König.Testing of Communicating Systems: Methods and Applications : Ifip Tc6 12th International Workshop on Testing of Communicating Systems, September 1-3, 1999, Budapest, Hungary. Springer, 1999. ISBN: 0792385810, 9780792385813

[Cacciari&Rafiq] Controllability and observability in distributed testing, L. Cacciari, O. Rafiq, Laboratoire TASC/Informatique, Université de Pau, avenue de l'université, 64000 Pau, France. Copyright 1999.

[Binder] Binder, R.V., Design for Testability with Object-Oriented Systems. Communications of the ACM, 1994. 37(9): p. 87-101.

[Le] Current Problems and Tools Support for Testing Distributed Systems. Anh Thu Le, A dissertation submitted for the degree of Bachelor of Applied Science with Honours in the Department of Information Science at the University of Otago, Dunedin, New Zealand. October 2006.

[Nicolescu et. al] Distributed Unit Testing, Runtao QU, HIRANO Satoshi, Takeshi OHKAWA and Takaya KUBOTA. Information Technology Research Institute, AIST. National Institute of Advanced Industrial Science and Technology (AIST). AIST Tsukuba Central 2, Tsukuba, Ibaraki, 305-8568, Japan. E-mail: nmg-all@aist.go.jp, Radu Nicolescu. Dept. of Computer Science, The University of Auckland, New Zealand

[Pisces] Pisces, Open source framework til udvikling af distribuerede unit tests i Java. Amir Shevat. Seneste version 1.3, 29/6 2006. http://sourceforge.net/projects/pisces/

[JUnit 3] JUnit, Open source framework til udvikling af unit tests i Java. David Saff, Erich Gamma, Erik G. H. Meade, Kent Beck. Version 3.8.2, 3/3 2006. http://sourceforge.net/projects/junit/

[JUnit 4] JUnit, Open source framework til udvikling af unit tests i Java. David Saff, Erich Gamma, Erik G. H. Meade, Kent Beck. Version 4.8.2, 8/4 2010. http://www.junit.org/

[JGroups] JGroups, A Toolkit for Reliable Multicast Communication, Bela Ban, Version 2.9.0.GA, 12/2 2010. http://www.jgroups.org

[Grobo] GroboTestingJUnit, Open source framework til udvikling af tests af multitrådede units i Java. Matt Albrecht, 2004. http://groboutils.sourceforge.net/index.html

[Christensen et. al] An Approach to Software Architecture Description Using UML Revision 2.0 Henrik Bærbak Christensen, Aino Corry, and Klaus Marius Hansen. Department of Computer Science, University of Aarhus, Aabogade 34, 8200 Arhus N, Denmark {fhbc,apaipi,mariusg}@daimi.au.dk, June 2007

[SysUnit] SysUnit, Open source framework til udvikling af tests af multitrådede og multiproces Java programmer. Anonyme udviklere. http://xircles.codehaus.org/projects/sysunit, seneste rettelse 2006.

[Bass et. al] Software Architecture in Practive, Second Edition. Len Bass, Paul Clements, Rick Kazman. Addison-Wesley, 2007. ISBN: 032115454959

[Christensen] Towards an Operational Framework for Architectural Prototyping, Henrik Bærbak Christensen, Department of Computer Science, University of Aarhus, hbc@daimi.au.dk, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, 0-7695-2548-2/05, copyright 2005.

Part II
Bilag - detaljerede arkitekturbeskrivelser

15 Disunit arkitektur

Disse afsnit beskriver disunits arkitektur. Først beskrives Pisces, som disunit er baseret på, og dernæst beskrives disunit, hvorefter en beskrivelse af observe points følger. Til sidst gennemgås disunit's T-Core integration.

15.1 Pisces

15.1.1 RemoteTestRunnerAgent

Pisces udvider JUnit, således at TestCases er i stand til at blive kørt i en anden proces. For at det kan lade sig gøre, skal en instans af Pisces klassen der implementerer en (test) maskine, RemoteTestRunnerAgent, køre i den anden proces (som kan køre på en anden fysisk maskine). Kommunikationen mellem maskinerne/processerne foregår gennem en abstraktion kaldet RemoteTestCommunicationLayer.

I den JVM, der beder om at få testen udført i en RemoteTestRunnerAgent i en anden proces, skal der blot køre en instans af det samme kommunikationslag, som RemoteTestRunnerAgent'en benytter. Typisk vil kommunikationslaget blive instantieret i en TestSuite, som i denne sammenhæng kaldes for test controller.

Figur 13 illustrerer runtime strukturen ved en remote test. De to noder repræsenterer to forskellige JVM'er med deres objekter. På controller JVM kører TestSuite'n som via kommunikationslaget kan bede RemoteTestRunnerAgent'en i Remote JVM om at udføre tests.

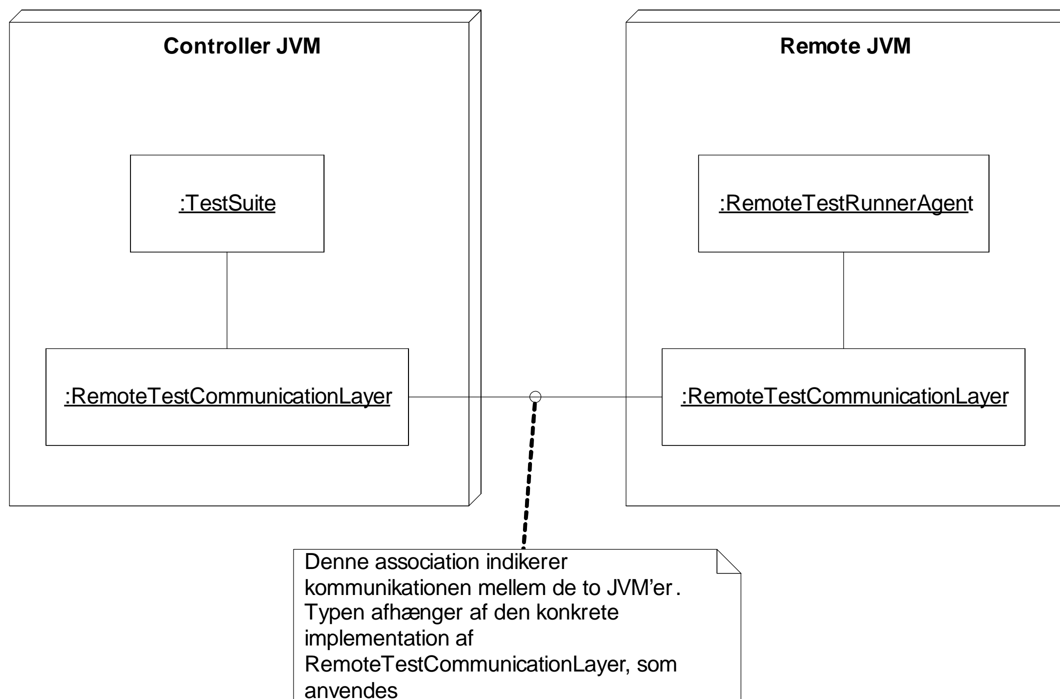

Figure 13: Component and Connector viewpoint: Runtime struktur ved udførsel af RemoteTests fra en controller til en RemoteTestRunnerAgent. De to noder, Controller og Remote JVM, illustrerer at objekterne indeni kører i den angivne virtuelle maskine

15.1.2 RemoteTest

Pisces klassestruktur og specialisering af JUnit TestCase kan ses i figur 14. Den helt centrale klasse er RemoteTestCase, som er en specialisering af JUnit's TestCase. Den giver test controlleren mulighed for at wrappe en almindelig TestCase i en RemoteTestCase, som så via RemoteTestCommunicationLayer kan bedes udført på en RemoteTestRunnerAgent ved hjælp af en RemoteTestRequest. RemoteTestRunnerAgenten som skal udføre testen, behandler testen efter JUnits principper med at lave en ny instans af klassen, køre setUp, køre test metode(r), hvorefter tearDown køres. Resultatet af testen returneres i et RemoteTestResult og præsenteres i test controlleren som om testen blev kørt lokalt.

Klassediagrammet i figur 14 illustrerer disse vigtigste klasser og sekvensdiagrammet i figur 15 viser, hvorledes kommunikationen og flow'et foregår ved udførelse af en remote test, fra instantiering, afsendelse til remote agent, udførelse og returnering af resultat til test controller.

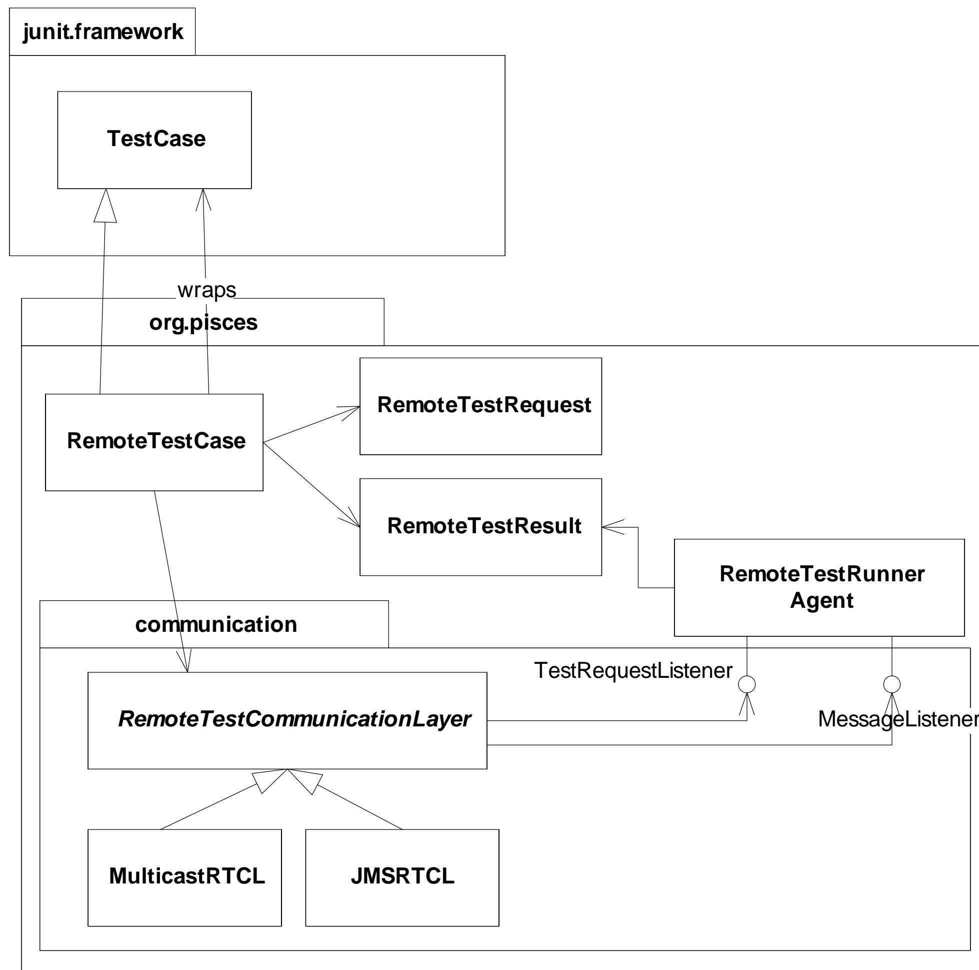

Figure 14: Module viewpoint: Klassediagram for arkitekturmæssig vigtige klasser i Pisces

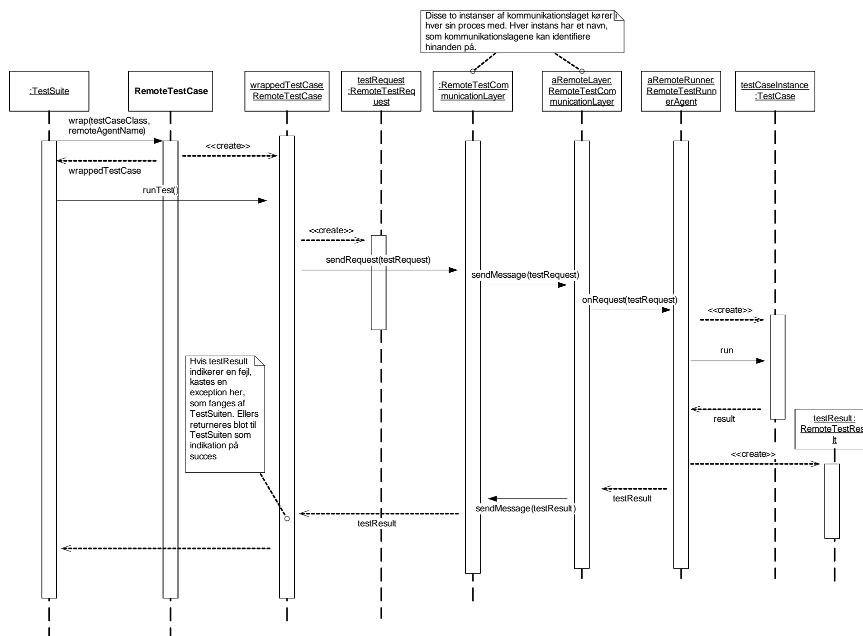

Figure 15: Component and Connectors viewpoint: Sekvensdiagram for udførsel af en RemoteTest fra en controller til en RemoteTestRunnerAgent

15.1.3 Eksekvering af tests

Pisces kræver, at controller og RemoteAgentTestRunners kører på samme binære Java klasser. Normalt køres der op imod samme jar-fil. De RemoteTestRequest som sendes fra controller til RemoteAgentTestRunner indeholder blot et klassenavn på testen, og RemoteAgentTestRunner må selv instantiere den ved hjælp af reflection.

Der er på ingen måde indbygget sikkerhed i Pisces, og typisk vil test også blive kørt på et internt, lukket netværk. Et allocation view kan ses i figur 16.

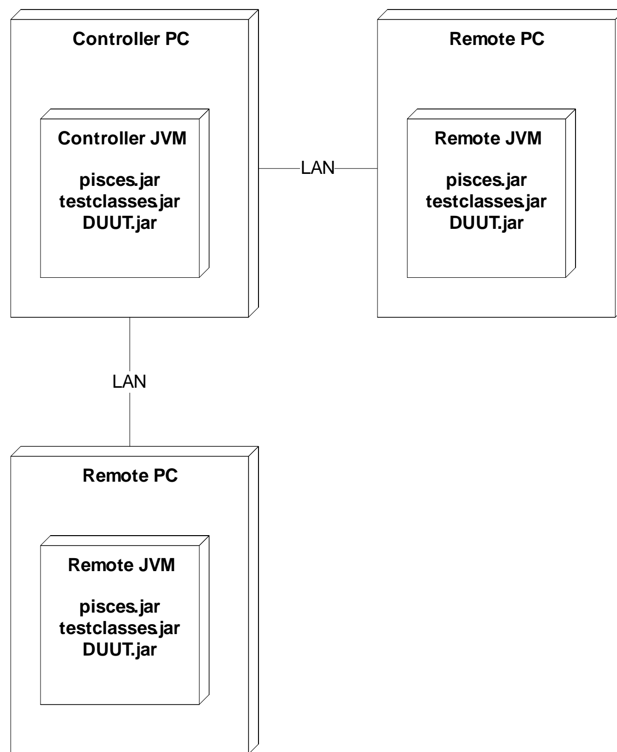

Figure 16: Allocation viewpoint: Deployment diagram for Pisces med en controller og to RemoteTestRunnerAgents (maskiner) som kører på hver sin PC i hver sin JVM (Remote JVM). Diagrammet illustrerer, hvordan de samme binære filer skal være tilgængelige for alle JVM'erne.

15.1.4 Uhensigtsmæssigheder i Pisces

Disunit er som nævnt bygget på Pisces, og derfor benytter det direkte Pisces' måde at fungere på som beskrevet i tidligere afsnit. Da disunit blev bygget, var der dog nogle elementer i Pisces, som blev fundet uhensigtsmæssige i forhold til at give udvikleren så nemt et redskab som muligt. Elementerne er beskrevet i dette afsnit.

Kommunikationslaget Testudvikleren skulle selv instiantere et konkret kommunikationslag i sin testcontroller (TestSuite), og det syntes at virke for lowlevel og kræve for dyb indsigt i Pisces virkemåde.

Derudover var der et valg af RemoteTestCommunicationLayer mellem en JMS[13] og en multicast implementation, der benyttede sig af UDP. Forsøg viste hurtigt, at UDP var ustabilt og JMS var for stor en mundfuld, idet tanken var, at disunit skulle bruges på en enkelt maskine eller højst blandt flere maskiner

[13] Java Message Service

i et LAN. JMS er implementeret for at kunne kommunikere over routere osv, hvilket ikke var relevant på daværende tidspunkt.

RemoteTestRunnerAgent I Pisces skulle man som udgangspunkt manuelt starte RemoteTestRunnerAgent'erne op, og det skal man også i disunit, hvis man ønsker at køre på flere fysiske maskiner. Men ideen var, at disunit også skulle køre på en enkelt fysisk maskine, der starter testcontrolleren og RemoteTestRunnerAgent'erne op i hver sin proces. Disse to måder at køre på blev senere døbt henholdsvis real distributed test og distributed test (til efterfølgende stor forvirring for udviklerne) og er illustreret i figur 17.

Det var besværligt at starte nye RemoteTestRunnerAgent'er sammenholdt med, hvad testudvikleren brugte i sin specifikke test, så der var behov for at testudvikleren kunne starte RemoteTestRunnerAgents dynamisk.

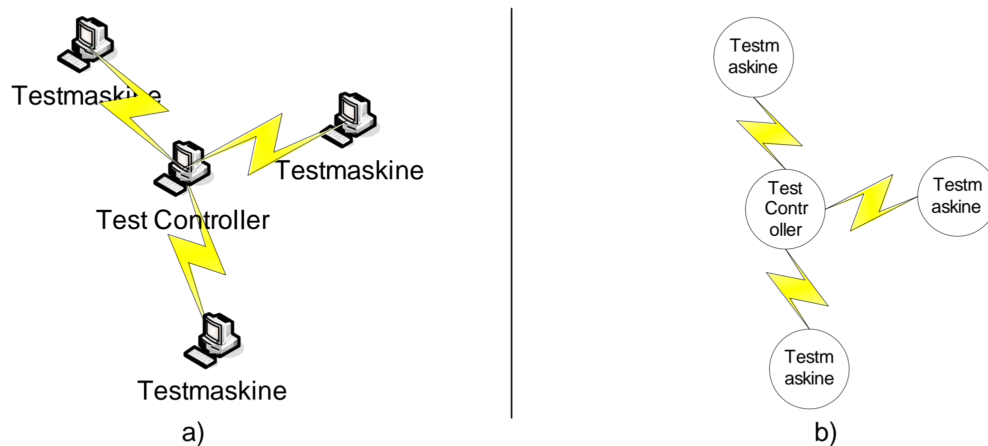

Figure 17: Diagrammet illustrerer løst analogien mellem at køre disunit på flere forskellige fysiske maskiner og at køre på en fysisk maskine. De to overordnede runtime scenarier for disunit. I a) kører testcontrolleren og hver RemoteTestRunnerAgent på sin egen fysiske maskine. Dette kaldes real distributed test. I b) kører de i hver sin proces på den samme fysiske maskine, også kaldet distributed test.

15.2 Disunit

15.2.1 Udvidelser til Pisces

- Disunit blev udviklet primært som et lag, som ligger ovenpå Pisces (jævnfør figur 18).
- En specialisering af Junit TestSuite, som er testudviklerens udgangspunkt for at skrive en test, blev indført. TestSuite'n, ved navn DisunitTestSuite, indeholder mulighed for enten dynamisk at kunne spawne RemoteTestRunnerAgent'er i egne processer (distributed test) eller at anvende RemoteTestRunnerAgent'er som kører på separate PC'er (real distributed test). To convenience klasser til de to metoder, DistributedTestSuite og RealDistributedTestSuite, blev desuden tilføjet.
- Disunit fik begrebet TestMachine, som er testudviklerens abstraktion for en RemoteTestRunnerAgent til brug i en DisunitTestSuite. Ved simple metode kald kan testudvikleren skabe en ny TestMachine, som kan få ordrer om at udføre RemoteTestCase's
- En ny implementation af RemoteTestCommunicationLayer, som anvender TCP til at kommunikere mellem RemoteTestRunnerAgent'er
- For at understøtte dynamiske TestMachines, som spawnes i egne processer, indførtes der en mulighed for via Pisces kommunikationslag at give besked til fra en TestMachine til en testcontroller om, hvornår den er klar til at modtage tests (dette er ikke muligt at vide i testcontrolleren, da TestMachines netop har deres egen proces). Controller har dermed mulighed for at sætte sig som lytter på beskeder fra kørende maskiner via interfacet RunningMessageListener.

Et diagram over disunits klasser i forhold til JUnits og Pisces kan ses i figur 18.

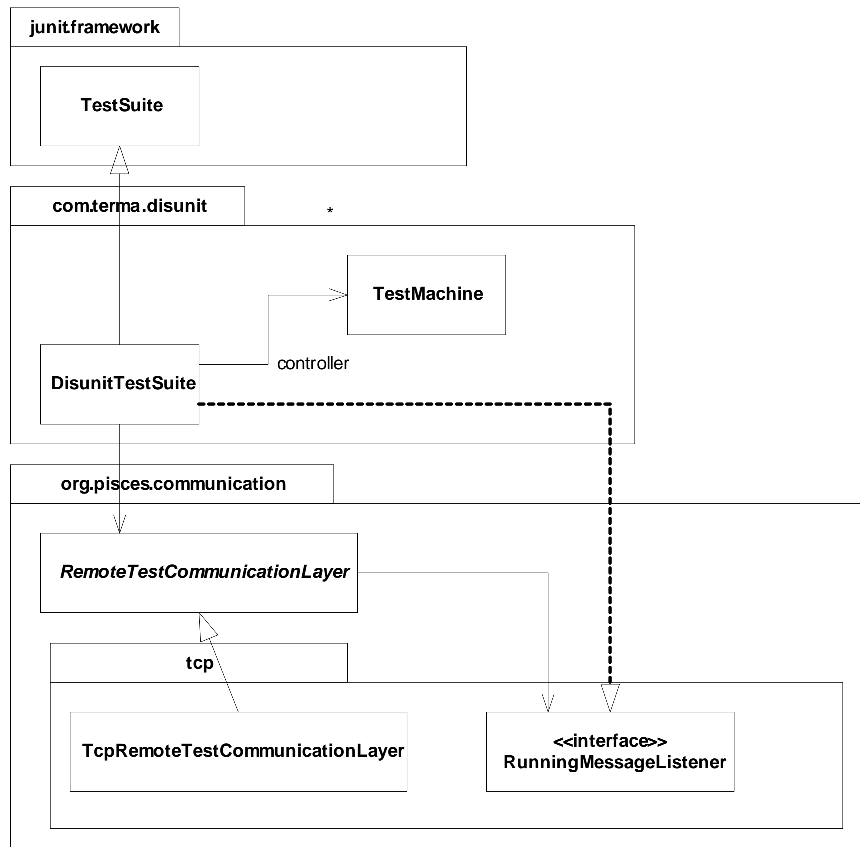

Figure 18: Module view: Klassediagram over disunit

15.2.2 DisunitTestSuite

Kernefunktionaliteten i disunit ligger i DisunitTestSuite. Udover at indeholde tests, skal suite'n også starte processer med RemoteTestRunnerAgenter i, der svarer til de TestMachines, som testudvikleren har skabt i sin test. For at kommunikere med de spawnede processer, bruges Pisces kommunikationslag, dvs. der tilføjedes en RunningMessageListener, som DisunitTestSuite lytter på og får besked når dens spawnede processer er oppe og klar til at modtage tests.

DisunitTestSuite's ansvar inkluderer:

- At holde styr på de TestMachines, som testudvikleren skaber
- At sørge for at spawne en process for hver TestMachine
- At holde overblik over, hvilke TestMachines, der er klar til at modtage tests
- Afsende tests og samle resultater

- Nedlukning af processer med TestMachines

Alt ovenstående arbejde foregår indenfor DisunitTestSuite's runTest() metode, dvs. at hele forløbet samlet set betragtes som et JUnit testforløb

15.2.3 Kommunikation mellem controller og TestMachines

Al kommunikation mellem controlleren og TestMachines foregår gennem Pisces. Dette betyder, at næsten hvilken som helst besked, der skal sendes til og fra controlleren foregår som et RemoteTestRequest (en undtagelse er udvidelsen med RunningMessageListener). Konsekvensen er, at de fleste opgaver i controlleren (DisunitTestSuite) pakkes ind i en instans af TestCase, selvom de intet har med selve testen at gøre, men derimod blot med kommunikationen, som skal gøre testen mulig.

15.2.4 Koordination af tests

Som nævnt, er det controlleren (DisunitTestSuite), som bestemmer hvilke test, der skal køres hvornår. I de helt simple test tilfælde tilføjes testene (som sendes til TestMachines vha. RemoteTestRequest funktionaliteten) blot til DisunitTestSuite, som så beder TestMachines om at udføre testene sekventielt.

Det er også muligt at tilføje TestSuites til DisunitTestSuite, og en mulighed er at anvende ActiveTestSuite, som er en del af JUnit. ActiveTestSuite udfører alle tests, som er tilføjet til den, i hver sin tråd, venter på at alle bliver færdige og samler resultatet. Dermed kan controlleren udføre flere tests samtidig, evt på flere fysiske maskiner.

Et eksempel er en test af et distribueret Observer pattern: En server indeholder en model af nogle data. Ved hjælp af lytterinterfaces kan klienter sætte sig som lyttere på opdateringer af modellen. Ved hjælp af RMI kald kan de modificere modellen. En test kunne være at lægge data i modellen fra flere maskiner samtidigt og så bagefter verificere, at alle data dukker op på alle maskiner. En ActiveTestSuite kunne indeholde den kode, som på hver sin maskine lægger data i modellen (dette er også en test, jævnfør disunits binding til Pisces' test paradigme). En anden ActiveTestSuite kunne indeholde tests'ene, som på hver sin maskine verificerer, at alle data er ankommet. De to ActiveTestSuites kan udføres sekventielt. Eksemplet er illustreret i figur 19.

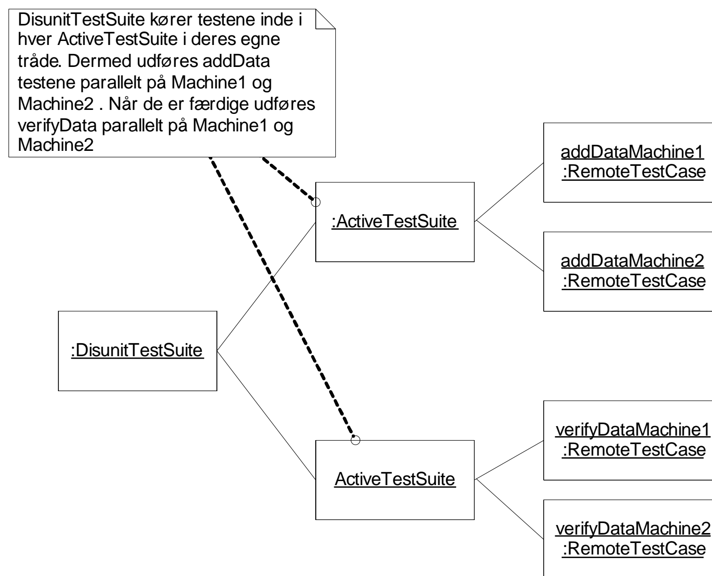

Figure 19: Component & Connector viewpoint: Objektdiagram for en DisunitTestSuite med to ActiveTestSuites, der hver indeholder to tests.

15.3 Observepoints

15.3.1 Anvendelse

Observe points kan afhjælpe:

- Controllability og observability problemer, der opstår af at ønsket data ikke kan aflæses af en DUUTs interface.
- Inter-test synkronisering og kommunikation, som ikke er hensigtsmæssigt at implementere ved hjælp af Pisces' kommunikationslag.
- Ventetid i forbindelse med asynkront input til en tester.

Der findes flere implementationer af observe points, men interfacet er det samme, som illustreret i figur 20.

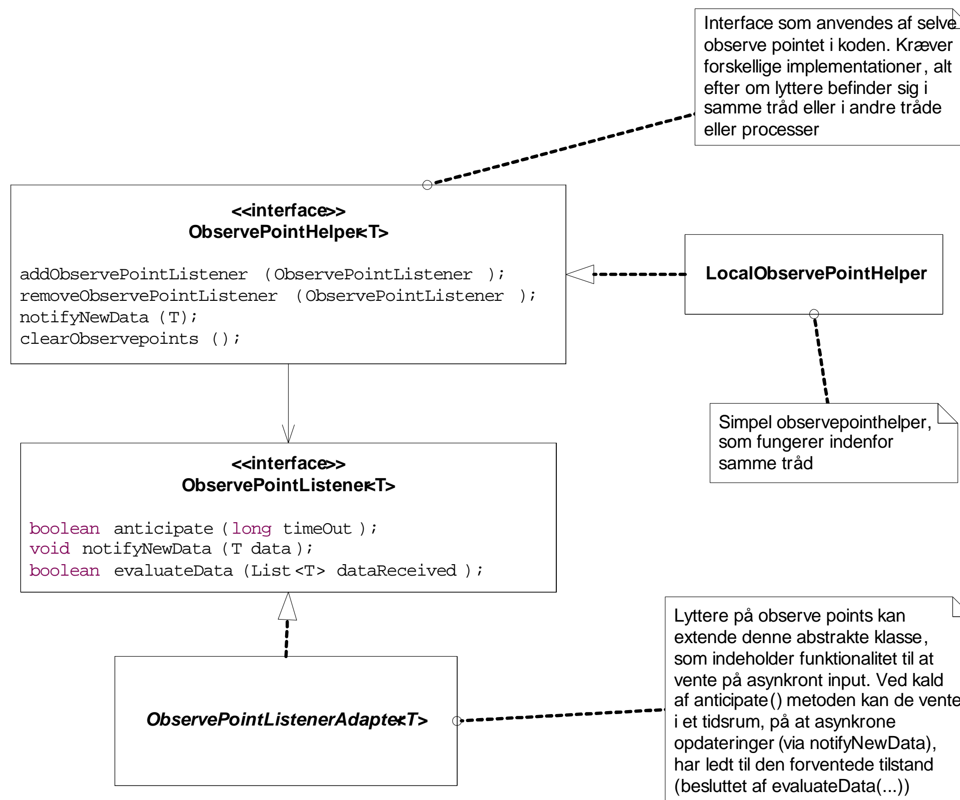

Figure 20: Module viewpoint: Klassediagram over interfaces og klasser, som anvendes til at realisere observe points i disunit

Et simpelt eksempel på en tester, som kan bruge observe point, er en tester som sender input til en DUUT og skal verificere, at inputtet bliver reflekteret i en asynkron lytter. Problemet kunne løses ved at tilføje lytteren, give input, og så vente i et tidsrum for bagefter at checke om data er som forventet. Denne metode introducerer dog spildtid, idet data kan være modtaget indenfor et meget kortere tidsrum, end testeren venter. I det enkelte tilfælde betyder det ikke meget, men samlet set, f.eks. i en kørsel af mange regressionstest, vil det løbe op i lang idle tid.

I stedet kan testeren extende ObservePointListenerAdapter og det relevante lytter interface og kalde notifyNewData fra lytterimplementationen, når data er modtaget. ObservePointListenerAdapter samler disse data op, og spørger hver gang evaluateData(), om den ønskede tilstand er nået. Derved kan testeren fortsætte umiddelbart efter, den forventede tilstand er opnået, eller rapportere et svigt, såfremt den ikke opnås.

15.3.2 Timingsmæssige udfordringer

Der kan opstå situationer, hvor lyttere på observepoints ikke er tilstede, før den interessante tilstand er opnået. Som et eksempel kan tages en DUUT, som har en form for bootcycle, hvor den initierer en række subelementer. DUUT-udvikleren har sat et observepoint ind, hver gang et subelement initieres. Såfremt testeren ikke har mulighed for at kontrollere initieringen (altså i essensen et controllability-problem), men blot sætter sig som lytter på observepointet, kan den distribuerede natur resultere i at visse eller alle subelementer er initieret, når testlytteren tilføjes.

ObservePointListenerAdapter kompenserer for sådanne tidsforskydninger ved at opsamle alle data, der er notificeret fra ObservePointHelperen i en intern tilstand. Dette kan i værste fald lede til OutOfMemory problemer, hvilket især er et problem, hvis observe pointet er placeret i produktionskode. Derfor er der mulighed for, via en system property at slå denne cache-funktionalitet fra, når design for test DUUTs kører i produktionstilstand.

En i praksis anvendelig implementation af ObservePointHelper kom med integrationen af disunit til en eksisterende platform, se afsnit 15.4.

15.4 T-Core integration

Én ting er eksempler, en anden er at anvende et distribueret unit test framework i en platform, som ikke er udviklet med henblik på at blive testet af frameworket.

15.4.1 T-Core

T-Core er Terma A/S's platform til udvikling af C3-systemer[14]. Systemerne er som regel koncentreret omkring et situationsoverblik over realtidsdata[15] i et eller andet form for operationsområde, der typisk er militært.

Funktionalitet i T-Core og systemer baseret på T-Core grupperes i T-Core komponenter. T-Core komponenter kan have en central (typisk serverplaceret) del og/eller en decentral del (klientplaceret). T-Core komponenterne kommunikerer altid med hinanden gennem den decentrale del, og den centrale del har typisk til opgave at holde på persisteret data eller fælles datamodeller. T-Core kører T-Core komponenter gennem en lifecycle proces[16]. T-Core komponenterne bringes igennem en række tilstande, hvor de har mulighed for at hente referencer til hinanden, danne kommunikationskanaler og vente på andre T-Core komponenter, før de erklærer sig selv som operationelle, dvs. at de er klar til anvendelse.

Applikationsudvikleren får gennem T-Core stillet en række faciliteter til rådighed, som er relevante i forhold til udviklingen af denne type systemer:

[14] C3: Command, Control and Communications

[15] T-Core understøtter ikke ægte realtid. Betegnelsen dækker over, at data typisk er interessante i et forholdsvist kort tidsrum efter tidspunktet, de blev skabt.

[16] T-Core komponenternes boot-proces, hvor en forudbestemt rækkefølge af metodekald på komponenterne udføres, kaldes i daglig tale for "lifecycle". Begrebet anvendes også som udsagnsord, som i f.eks. "komponenten lifecycles" og tillægsord, som i "den lifecyclede komponent".

- Basale typer og omregningsmetoder.
- Transparent distribution af data gennem decentrale og centrale komponenter.
- Indbygget fail-over funktionalitet[17] mellem servere.
- Fællesudviklede standardkomponenter til domænespecifik funktionalitet.

15.4.2 Disunit i T-Core

Disunit blev introduceret i T-Core som en central komponent, der er placeret på såvel servere som klienter i et T-Core system. Testklasser (klasser som nedarver fra JUnits TestCase) bliver betragtede som lifecyclede komponenter, der ejes af den enkelte disunit komponent, som også bringer dem igennem lifecycle.

Ideen var, at en disunit T-Core testsuite i en tråd skulle starte T-Core systemet, som var konfigureret med disunit komponenten, der skulle lifecycle sine testcases og dermed udføre dem. Dermed bliver T-Core startet op i en RemoteTest, som testcontrolleren beder RemoteTestRunnerAgent om at udføre. Den venter så på, at disunit T-Core komponenten er lifecyclet og operationel, og dermed kan TestCases udføres. Denne struktur svarer til en overordnet lagdelt arkitektur, som afbilledet i figur 21.

[17] Redundante servere kan stå klar til at overtage requests fra den primære server, såfremt den svigter.

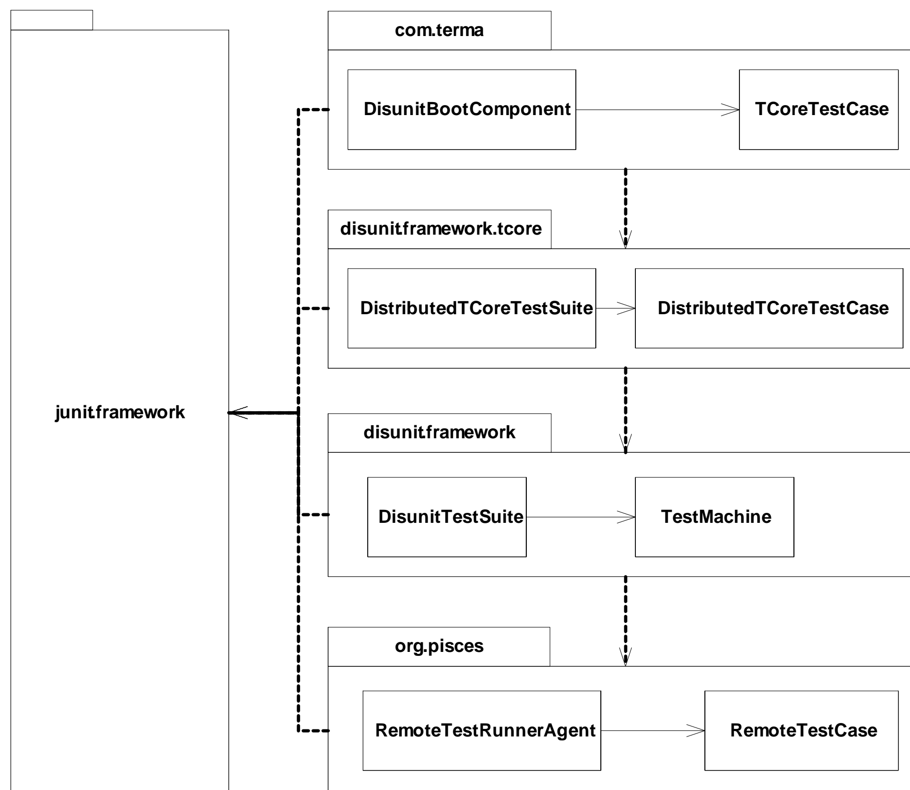

Figure 21: Module view: Package diagram og delvis klassediagram over et lagdelt view på disunits integration i T-Core

En udvikler af distribuerede unit tests i T-Core kan altså forvente, at test bliver bragt i et test harness, hvor T-Core er operationelt og oppe at køre, og dermed kan udvikleren tilgå de ønskede T-Core komponenters interface og operere på systemet.

15.4.3 Principper i T-Core disunit

En disunit test i T-Core adskiller sig fra en JUnit Test Case på en række punkter. Desuden er der en række forudsætninger og antagelser i forbindelsen med kørslen af T-Core tests:

- En test defineres altid af en DistributedTCoreTestSuite
- Testcontrolleren er DistributedTCoreTestSuite, som starter T-Core op på de maskiner, som er blevet skabt i testen.
- De enkelte maskiner sørger - via T-Core - for at lifecycle T-Core komponenter og de DistributedTCoreTestCases, som findes.

- Testcontrolleren venter indtil alle T-Core klienter er operationelle
- De enkelte metoder i DistributedTCoreTestCases køres. Mellem hvert kald laves en ny instans af test klassen (som i JUnit), hvorfor alle TestCases, der har member field, skal erklære dem static.
- Testcontrolleren samler resultater og lukker T-Core systemerne ned.

Hele forløbet svarer på testcontrolleren til at køre en JUnit testsuite (DistributedTCoreTestSuite).

15.4.4 DistributedObservePointHelper

T-Core indeholder en komponent, som gør det muligt at lave en distribueret data model, der kan lyttes på i de enkelte T-Core klienter. Lyttere, som ikke har været med fra begyndelsen, får et tilstandskald når de tilføjer deres lyttere, så de kan indhente data, de ikke nåede at få tidligere.

Ved hjælp af denne distribuerede data model komponent var det muligt at implementere en ObservePointHelper, som fungerer på tværs af TestMachines (i kontrast til LocalObservePointHelper fra eksemplerne, som kun fungerede indenfor en enkelt JVM). Denne ObservePointHelper, som blev døbt DistributedObservePointHelper, anvendes til inter-test kommunikation. Det anvendes ikke til at oprette PCO'er i T-Core med henblik på design-for-test og at give testere mulighed for at lytte på interne tilstande, da man ikke ønskede at gøre T-Core afhængigt af et testframework.

DistributedObservePointHelper giver test case'ne mulighed for at kommunikere indbyrdes med information om, hvilke data de har fået fra T-Core systemet. Dermed kan man opnå en større grad af observability og controllability i sine tests, da en tester er i stand til at modtage beskeder om, hvad andre testere har modtaget som output til et givet input. Desuden medvirker de til at gøre determinismen nemmere at opnå, da synkronisering mellem tests kan foretages i DistributedObservePointHelper i tilgift til de synkroniseringsmekanismer som følger af opbygningen af testsuiten (se evt. afsnit 15.2.4).

Den store ulempe ved DistributedObservePointHelper er, at disunit gøres afhængigt af T-Core komponenter. Med andre ord stoler man på, at T-Core komponenten der tilbyder den distribuerede datamodel funktionalitet fungerer, og man inddrager den i princippet som en del af test frameworket. I praksis har det ikke været et problem, fordi denne komponent i forvejen er grundigt gennemprøvet i både test og produktions, men principielt set baserer man sig på noget andet end test frameworket og en eventuel defekt i den distribuerede data model kunne give anledning til store problemer, da da test udvikleren sandsynligvis ville tro, at et svigt skyldtes en defekt i den komponent, han reelt tester.

15.4.5 Non-funktionelle tests

Disunit bliver også anvendt til stabilitetstests, hvor T-Core bliver startet op på flere fysiske maskiner (real distributed) og kører tilfældige tests, som stresser

systemet over dage af gangen . Emnet gennemgås ikke nærmere men nævnes, idet scenariet afslørede endnu et brugsmønster af disunit. I stabilitetstest bliver disunit startet af T-Core i stedet for omvendt. Dette betyder i praksis at DistributedTCoreTestSuite mister noget af sit ansvar med at starte T-Core op og kræver tilstedeværelsen af en variation af klassen, som bliver kaldt RealDistributedTCoreTestSuite.

16 Disunit 2.0 arkitektur

16.1 Kommunikationslag

Som tidligere nævnt, er kommunikationen i disunit 2.0 bygget op som lag. Nederst ligger JGroups, som tager sig af netværkskommunikationen, hvor der kommunikeres i JGroups Messages. Communication laget fungerer et Facade pattern op mod JGroups og tilbyder kommunikation i Serializable, og ovenpå ligger Machines abstraktionen, som kommunikerer i RunnableMessages i en simpel request/respond protokol mellem en MachineController (test controller) og Machines (en (test) maskine). Denne arkitektur, som kan betrages som en peer-to-peer kommunikations model[18], er illustereret i figur 22.

[18] Kendt fra bl.a. OSI Reference Model

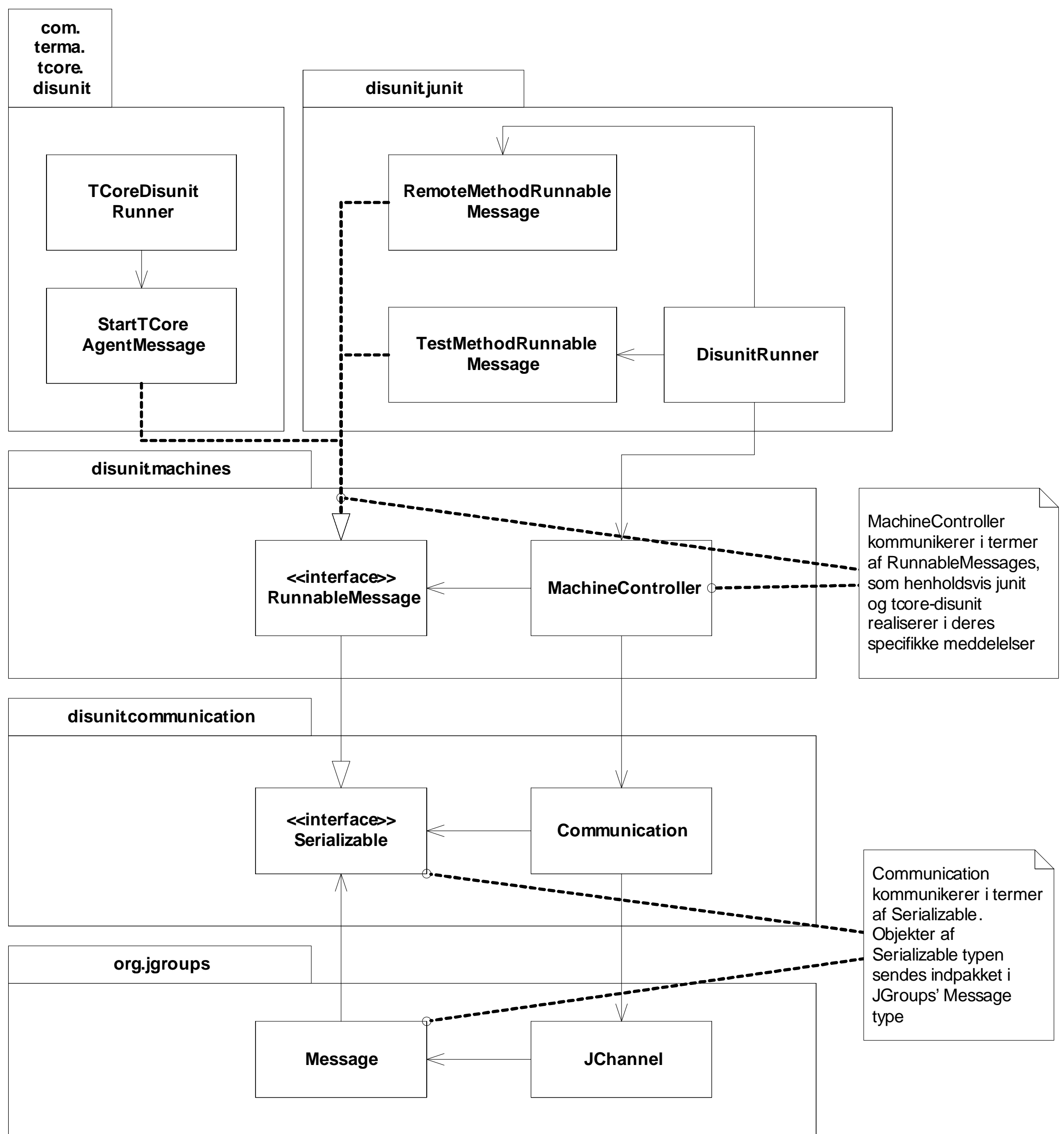

Figure 22: Klassediagram, som illustrerer lagdelingen i disunit 2.0

En RunnableMessage er en besked med et objekt, som implementerer Runnable interfacet. Sådan en besked sendes via en MachineController til en Machine, som udfører koden i objektets run() metode. Når koden er udført returneres objektet, som derfor kan indlejre eventuelle data til brug på afsendersiden. Dermed tilbyder MachineController sine klienter udførelse af arbitrær kode på de Machines,

som den har i sin gruppe. Figur 23 viser et runtime view på kommunikationen.

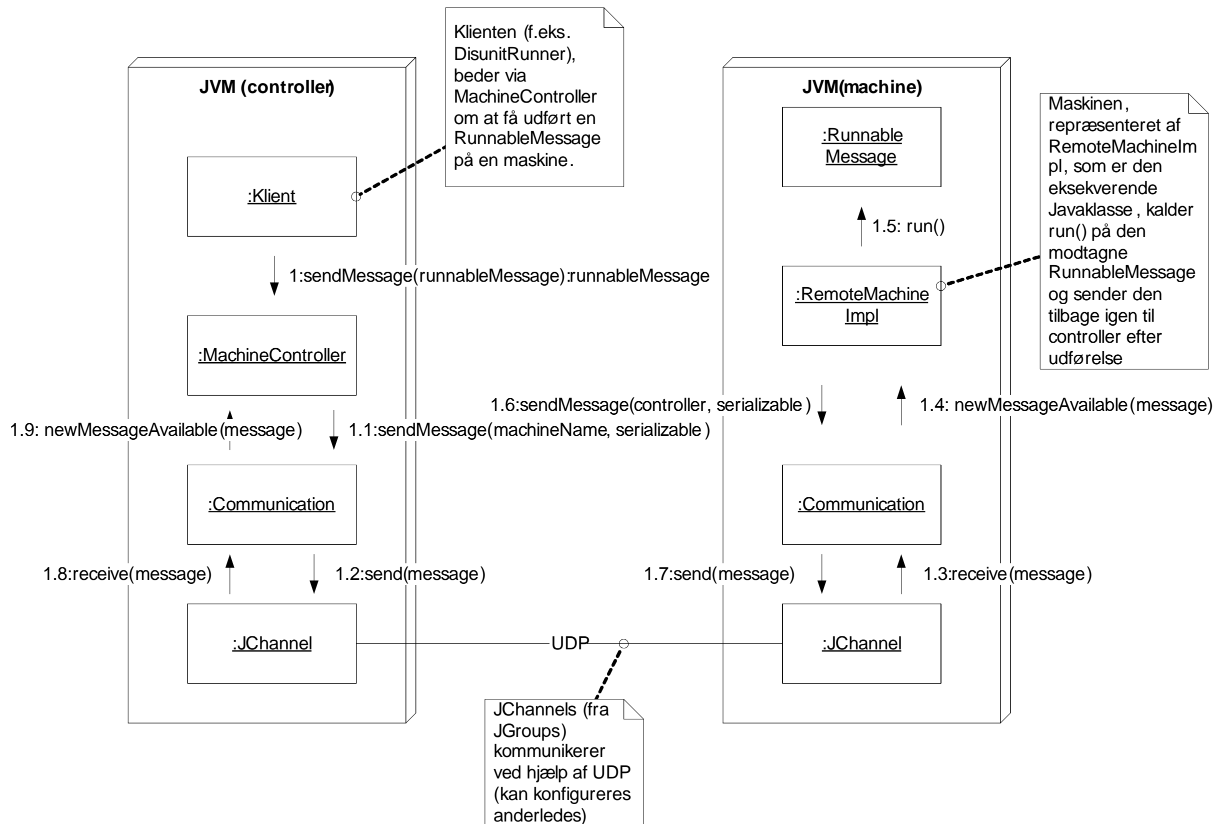

Figure 23: Communication diagram, som viser kommunikation mellem controller og en machine ved transmission og udførelse af en RemoteRunnableMessage

16.2 JUnit - DisunitRunner

DisunitRunner er den centrale del af disunit 2.0s JUnit 4.x-integration. JUnit bruger Runners til at køre test med, og ud fra en såkaldt Builder bestemmer den hvilken konkret implementation af Runner, der passer til en given test. Ved at annotere en test klasse med @RunWith(DisunitRunner.class) annotation får disunit 2.0 således et hook ind i JUnit frameworket gennem DisunitRunner.

DisunitRunner bygger en test op ud fra de disunit 2.0 (og JUnit) annotations, som anvendes i den enkelte testklasse, og som blev beskrevet i afsnit 10.1. Den anvender JUnit's Statement-konstruktion. En Statement er noget funktionalitet, som udføres på et senere tidspunkt, dvs. under runtime af testkørslen. Disunit nedarver fra Statement og danner et Composite pattern, hvor DisunitStatements kan indeholdes i DisunitStatements. Klassediagrammet i figur 24

illustrerer arkitekturen i disunit 2.0s JUnit integration.

Ved hjælp af dette Composite pattern er det muligt at opbygge sammensatte strukturer af tests, der kører i sekvens og samtidigt. Et eksempel på en objektgraf, dannet af DisunitRunner, kan ses i figur 25.

Figure 24: Composite pattern i disunit 2.0's DisunitStatement opbygning

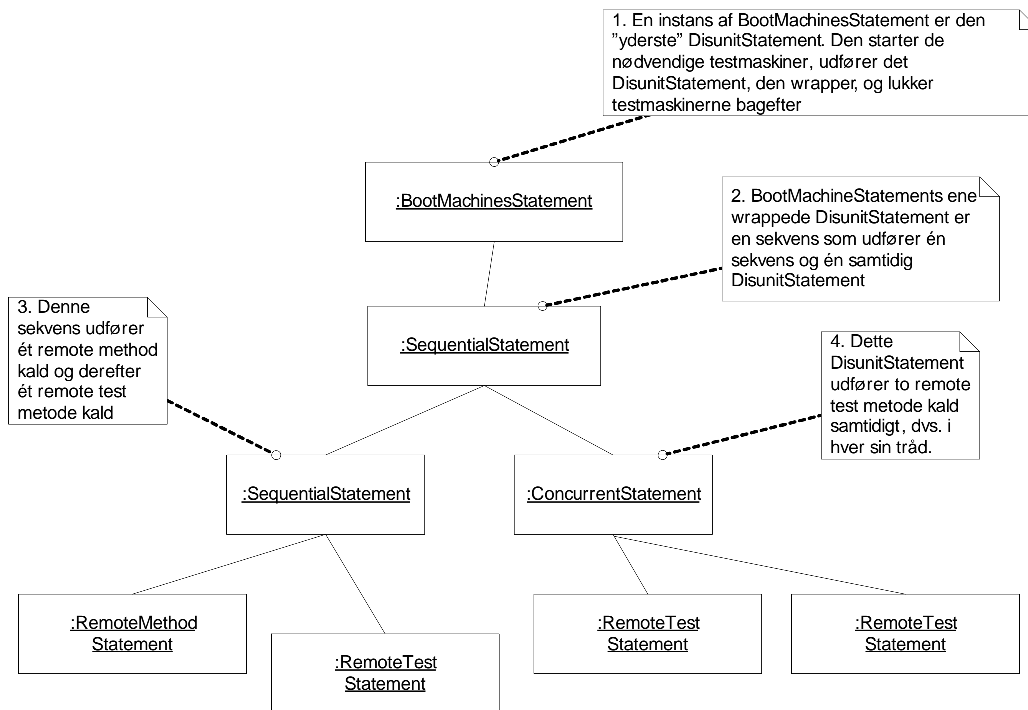

Figure 25: Instansdiagram over en teststruktur af DisunitStatements, opbygget af DisunitRunner ud fra en test klasse

16.3 T-Core disunit

T-Core integrationen til disunit 2.0 går delvist gennem JUnit delen. I DisunitRunner fra JUnit delen blev der lavet et hook til nedarvende klasser (Template Method pattern), som har mulighed for at wrappe de DisunitStatements, som DisunitRunner laver med egne DisunitStatements. Således indføres en @TCoreAgents(@TCoreAgent[]) annotation, som definerer T-Core agenter (svarer til JVM'er). Ved hjælp af Template Method pattern, wrappes DisunitStatements med StartTCoreAgentStatement, som søger for at starte agenterne på de enkelte maskiner.

T-Core integrationen anvender også MachineController direkte til at udføre RemoteRunnableMessages som starter T-Core agenter op i maskinerne, som disunit 2.0 har startet. Et statisk view af T-Core disunit er vist i figur 26.

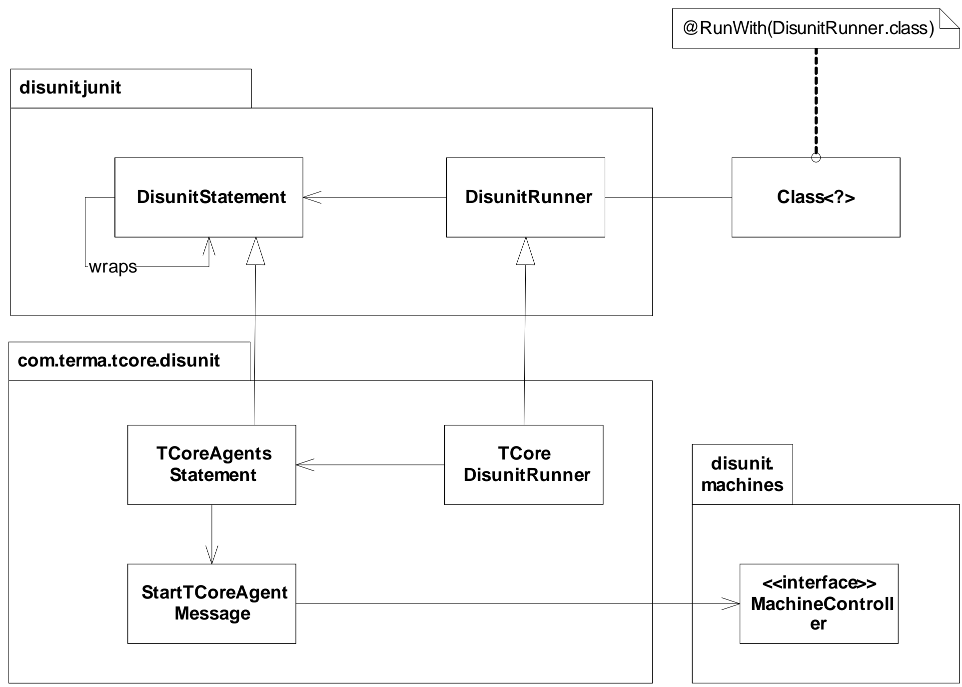

Figure 26: Statisk view af T-Core disunit integrationen

16.4 Observe points

Der er ikke sket ændringer i observe point arkitekturen siden første version af disunit. Implementationen af observe points er dog ændret til at bruge JGroups i stedet for at bruge T-Cores faciliteter. Derved er testkommunikationen fuldstændig uafhængig af T-Core.