DATALOGISK INSTITUT
DET NATURVIDENSKABELIGE FAKULTET
AARHUS UNIVERSITET

Hovedopgave

Diplom i Informationsteknologi
linien i Softwarekonstruktion

Caching i JEE backends

af
Peter Bugge Jakobsen

15. juni 2010

Peter Bugge Jakobsen,
studerende

Anders Møller, 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

Indholdsfortegnelse

1 Indledning

Denne hovedopgave handler om caching af backendkald i JEE applikationer. Backendkald i den forstand at en applikation kalder en ressource udenfor sin egen kontekst, hvilket for en JEE applikation vil sige dens egen JVM. Der er altså tale om fx et kald til en database, et kald til en webservice eller en hvilken som helst anden ekstern datakilde. Opgaven tager udgangspunkt i en kompleks JEE verden, hvor caching ofte ikke anvendes eller i bedste fald anvendes ad-hoc på en manuel og ikke genbrugelig måde.

Udfordringen opdeles i tre centrale områder som er: *logging af backendkald, analyse af logs og implementering af caching.* De tre emner behandles individuelt, men udgør tilsammen grundlaget for at gøre caching af backendkald mere automatisk og genbrugelig.

1.1 Motivation

Til daglig arbejder jeg med JEE applikationer målrettet web – det vil sige alt fra små applikationer som en låneberegner til større applikationer i en netbank. Igennem dette arbejde som softwarekonstruktør har jeg haft en rækker konkrete oplevelser, hvor applikationer opførte sig ret uheldigt i forhold til svartider og ressourceforbrug. Der har været flere af disse situationer, hvor jeg har tænkt tanken: *hvordan kunne det gå så galt?*

Fælles for mange af disse udfordringer er, at brugen af frameworks i høj grad har skjult detaljerne for den enkelte udvikler. Det kunne fx være en kompliceret lifecycle model i et webframework, der gjorde, at man fik flere kald til sin backend, end man egentlig regnede med. Måske opdager man det aldrig, fordi backendkaldet til databasen har en god svartid pga. caching i databasen. Måske sker det samme backend kald et par steder i den samme applikation.

I et konkret tilfælde resulterede fire JSP Expression Language udtryk i fire kald til den samme backend. Man havde muligvis opdaget, at det ikke var hensigtsmæssigt, og derfor forsøgt sig med en ad-hoc caching af kaldet. Desværre virkede cachen ikke pga. en fejl i invalideringen. Fejlen gjorde, at cachen aldrig blev opdateret efter, at første indsættelse var blevet forældet.

Fejlen blev først opdaget tre år efter idriftsættelse af applikationen i forbindelse med en performance analyse. Her undrede en udvikler sig over, at funktionen blev kaldt hyppigere end den umiddelbart burde. Efterfølgende opfølgning har vist, at antallet af kald gik fra ca. 20.000 om dagen til mindre end 1.000. Selve funktionen performede meget godt med svartider omkring 40 millisekunder, hvilket kan forklare, hvorfor det aldrig blev opdaget. Alligevel gav en passende caching af backendkaldet en årlig besparelse på ca. DKK 100.000 pga. sparede CPU minutter på mainframe.

Forklaringerne på den slags tilfælde kan være mange. Min opfattelse er, at mængden af frameworks til rådighed for en JEE udvikler er en af de vigtigste. Ganske vist kan de være med til at spare tid og skabe ensartethed på tværs af projekter i en større udviklingsorganisation. Men de er også med til, at vi ikke behøver kende ret meget til helt fundamentale begreber indenfor IT. Efter min overbevisning er lige netop caching et helt central begreb - især når man laver større distribuerede systemer.

Ofte overlades caching dog helt til andre elementer i en typisk JEE arkitektur som vist på Illustration 1. Det kan være i frameworks, applikationsservere, netværkskomponenter eller databaseservere. Det betyder ofte, at den enkelte udvikler ikke bruger nævneværdig tid på den slags detaljer.

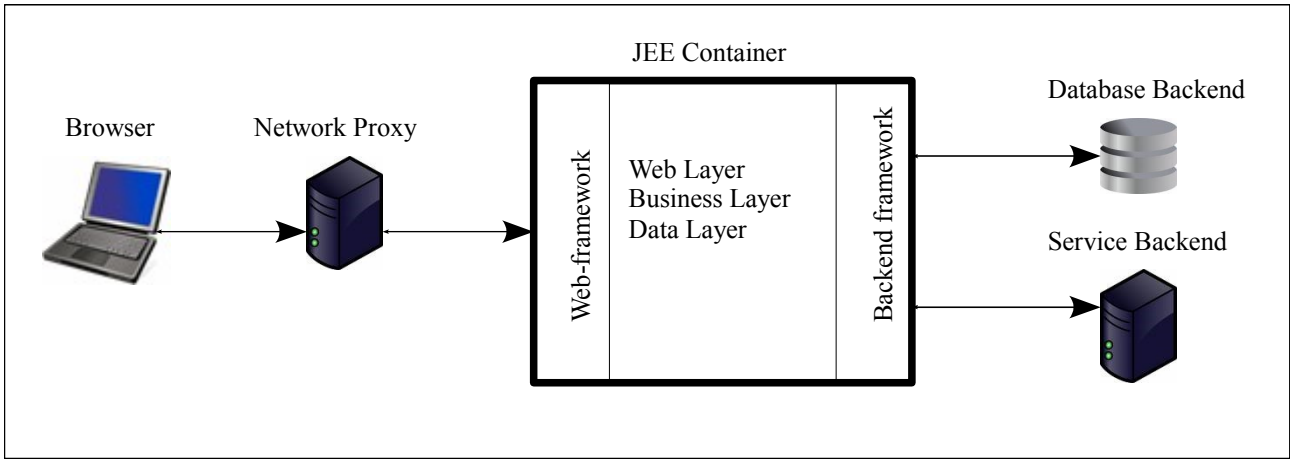


Illustration 1:Forenklet logisk arkitektur for applikationer i et JEE miljø.

For at kunne undgå lignende situationer synes jeg, at det kunne være interessant at undersøge, om man på en mere automatisk og genbrugelig måde kan finde og undgå den slags problemer. Det er specielt muligheden for at opnå bedre caching i JEE applikationers backendkald, der interesserer mig. Det vil sige færre kald til databaser eller webservice backends, hvilket i mange situationer vil kunne spare ressourcer og give bedre svartider.

En anden grund til, at jeg mener, at emnet er interessant, er at det vigtigt at kende mulighederne og teorierne bag, når man skal løse konkrete udfordringer. Derfor vil jeg også bruge denne opgave til, at sætte mig selv og læserne bedre ind i de muligheder og udfordringer, der måtte være med caching.

1.2 Problemformulering

Problemet er at der ofte ikke implementeres caching af backendkald i en JEE kontekst. En årsag til dette er efter min erfaring, at man ikke har et tilstrækkeligt overblik over de backendkald, som applikationerne rent faktisk foretager. Det vil sige, at man fx ikke kender til antal, svartider, evt. dubletter osv. Udover dette er manglende kendskab til selve elementerne i caching også en medvirkende årsag til problemet.

Den første del af udfordringen er at finde de backendkald, som kunne være kandidater til, at blive cachede. Denne del af udfordringen består af to dele: ***logging af backendkald*** *og* ***analyse af logs***. Den næste del bliver at ***implementere caching***, der passer til den givne situation.

Udfordringen med logging er, at det skal ske på en automatisk og genbrugelig måde, som giver mindst mulig arbejde for den enkelte udvikler. Det vil sige, at finde det bedste sted i en typisk JEE arkitektur, hvor der er det input og output, der skal til for, at en efterfølgende analyse giver mening.

Efterfølgende analyse af de loggede backendkald skal ligeledes finde sted så automatisk og genbrugeligt som muligt. Udfordringen her er dog også, at finde ud af, hvordan man laver en analyse, der kan finde kandidater til caching.

Sidste del af udfordringen bliver at implementere caching. Selve implementeringen skal ligeledes være så automatisk og genbrugelig som mulig. Samtidig er det målet, at koble implementeringen af caching sammen med resultaterne af den automatiske analyse.

Med udgangspunkt i ovenstående udfordringer opsummeres opgavens problemformulering i følgende hypotese:

> ***"Det er muligt, at lave caching af backendkald i JEE applikationer mere genbrugelig og automatisk end ad-hoc løsninger. Det kan gøres ved, at lave automatisering og genbrug af de tre centrale områder: logging af backendkald, analyse af logs og implementering af caching"***

1.3 Afgrænsning

Dette afsnit beskriver de væsentligste afgrænsninger, som jeg har foretaget i forhold til indhold og omfang af opgaven. Det første afsnit handler om placering af caching i forhold til en typisk JEE applikationsarkitektur. Det næste handler om logging og analyse. Endelig handler det sidste om praktisk afprøvning af prototyper.

1.3.1 Placering af caching i arkitektur

Med udgangspunkt i et ønske om at finde generelle og genbrugelige løsninger på tværs af forskellige typer af backendkald, ligger opgaven fokus omkring det generelle i datalaget i en typisk "three-tier" JEE applikationsarkitektur. Altså en arkitektur inddelt i præsentations-, forretnings- og datalag. Det vil sige, at der **ikke** fokuseres på specifikke løsninger relateret til bestemte typer af backendkald. Det kunne fx være caching i JPA (Java Persistance API) eller Hibernate, som er to eksempler på frameworks til automatisk mapping af objekter til relationelle databaser – ORM værktøjer. Eller det kunne være caching af backendkald til webservices igennem HTTP.

Årsagen til denne afgræsning er at den organisation, som jeg arbejder i, har mange forskellige typer af backends. Dette er ikke specielt unikt i lidt større IT-organisationer. Et par eksempler er JDBC, MQ, CICS og forskellige webservices baseret på HTTP. MQ er en kø-baseret teknologi fra IBM, der bruges til understøttelse af JMS kommunikation med en mainframe baseret på IBM's Z/OS. CICS er et andet IBM alternativ til understøttelse af kommunikation imellem mainframe og JEE.

Fordi der i min organisationen i forvejen findes et centralt framework til datalaget, ville det være et oplagt sted, at ligge en central caching mekanisme. Det vil gøre genbrug på tværs af forskellige backends muligt, hvilket ikke ville være tilfældet med en specifik løsning til fx database backends.

1.3.2 Logging og analyse

Logging og efterfølgende analyse heraf kunne evt. gennemføres ved anvendelse af kommercielle produkter til diagnosticering og performanceanalyse. Men det ønsker jeg ikke at tage stilling til i denne opgave. Der er to grunde til denne afgrænsning.

For det første kræver langt de fleste produkter ret mange ressourcer – fx til uddannelse, tilpasning til det konkrete miljø, tilpasning til den konkrete applikation og sidst men ikke mindst licensprisen.

Udover ressourcerne er den anden grund til afgrænsningen, at det alt andet lige vil kræve en eller anden form for integration. Ellers ville det ikke være muligt, at koble resultater sammen med en egentlig central caching mekanisme. Det er min vurdering, at det vil være en for stor opgave, at finde og analysere mulighederne for den slags værktøjer, og efterfølgende lave integration inden for rammerne af denne opgave.

1.3.3 Praktisk afprøvning af prototyper

Mine muligheder for at afprøve opgavens prototyper i praksis er begrænsede af den virkelighed, som jeg befinder mig i på min arbejdsplads. Det er primært to forhold, der gør, at jeg har fravalgt denne mulighed.

Det første er at vi p.t. anvender JEE 1.4. I skrivende stund er vores infrastrukturafdeling ved at forberede understøttelsen af JEE 5. Det betyder at al vores produktionskode p.t. er skrevet i Java 1.4. De teknikker og principper, som jeg anvender i mine prototyper, baseres på Java 1.5+. Det er fx JUnit 4+ og generics fra java 1.5+ Det vil altså kræve en opdatering af disse applikationer og deres afhængigheder for at kunne afprøve prototyperne.

Den anden faktor er et realistisk datagrundlag. For at få realistiske målinger bør der opsamles data fra kørende applikationer i produktion med et realistisk load og et realistisk brugsmønster. Det optimale ville have været, at der fandtes test scripts, som kunne afvikles for at skabe load. Desværre er det ikke tilfældet for de pågældende applikationer. Og jeg vurderer ikke, at det er realistisk, at skabe sådanne i scope af dette projekt. En anden mulighed var så at afprøve ting i et rigtig produktionsmiljø, men fordi der primært er tale om applikationer i vores online netbank, er det af hensyn til driftstabilitet og sikkerhed ikke en mulighed.

1.4 Metode

I dette afsnit vil jeg skitsere de områder, som opgaven konkret vil beskæftige sig med, samt de metoder og teknikker jeg har tænkt mig at anvende.

1.4.1 Logging af backendkald

Første udfordring bliver, at kunne implementere logging af backendkald på en automatisk og genbrugelig måde. Målet er altså en generel og automatisk løsning, som giver så lidt ”ekstra” arbejde som muligt for en udvikler. For at undersøge og sammenligne forskellige alternativer, har jeg tænkt mig at anvende arkitektur prototyper [2].

Jeg vil konkret bygge to prototyper:

- Logging i DAO (Data Access Object)[1]
- Logging via dynamic proxy [3]

Den første prototype bygges primært for at skabe rammene for den næste. Det gør den ved, at bygge et skelet af en ofte anvendt arkitektur i datalaget i en JEE applikation. Denne arkitektur indkapsler dataadgang ved at bruge et klassisk JEE DAO pattern, hvor konkret implementeringen af CRUD (create, read, update og delete) funktioner isoleres for resten af applikationen. Den næste prototype bygges for, at forsøge, at forbedre graden af automatik og genbrugelighed.

Hver enkelt prototype evalueres på, hvordan de klarer sig i forhold til primært maintainability og sekundært performance. Det er de kvalitetsattributter, som jeg har vurderet, er de vigtigste i forhold til at gøre logging mere automatisk og genbrugelig fra applikation til applikation. Evalueringen af prototyperne opsummeres i en samlet delkonklusion for begge prototyper.

1.4.2 Analyse af logs

Den næste udfordring bliver, at lave automatisk og genbrugelig analyse af de input og output, der logges. For at en sådan analyse skal give mening, har jeg først brug for at identificere, hvad analysen skal kunne finde. Det har jeg tænkt mig at gøre ved, at undersøge nogle eksisterende ad-hoc analyser. Disse er udarbejdet i forbindelse med performance analyse af en samling af kørende applikationer. Målet er at finde mønstre i analyserne, der kan finde kandidater til caching.

Efterfølgende vil jeg bygge en prototype, der kan:

- simulere et eller flere mønstre
- automatisk analysere og rapportere om kandidater, der følger disse mønstre

Prototypen vil jeg evaluere i forhold til at lave manuelle analyser af logs. Konkret vil jeg evaluere på hvilke resultater de to løsninger kan finde og hvor tidskrævende og komplekse de er.

1.4.3 Implementering af caching

Sidste udfordring er implementering af caching. Min plan kan inddeles i følgende tre underopgaver:

- undersøgelse af caching generelt
- undersøgelse af strategier i forhold til mønstre fra analysen af logs
- implementering af prototype til central caching

Med udgangspunkt i mit begrænsede kendskab til caching generelt, har jeg tænkt mig, at undersøge hvilke muligheder der findes. Og hvilke udfordringer jeg i den forbindelse skal forholde mig til. Jeg vil også undersøge hvilke strategier, der kan anvendes til caching af situationer, der følger de mønstre, jeg forventer at finde i min analyse af logs.

Endelig vil jeg forsøge at bygge en prototype, der implementerer en central caching mekanisme. Prototypen skal i første omgang kunne håndtere et eller flere af de konkrete mønstre, som jeg tidligere har identificeret.

Resultatet bliver en opsummering af mine undersøgelser af mulighederne og udfordringerne med caching generelt. Efterfølgende vil jeg evaluere prototypen på dens vigtigste arkitektturkvalitetsattributter, som jeg vurderer er maintainability og performance. Endvidere vil jeg teste hvor effektiv prototypen er, ved at koble den på simulering, som jeg bygger til analysen af logs.

2 Relateret arbejde

I mit arbejde med caching i en JEE kontekst, har jeg i stor udstrækning trukket på artikler, blogindlæg og detaljer fra forskellige konkrete implementeringer på nettet [6-17]. Det skyldes primært at den klassiske JEE litteratur ikke beskæftiger sig nævneværdigt med emnet caching. Fx forholder Martin Fowler sig kun meget generelt til det i indledningen til Patterns of Enterprise Application Architecture [4]. Ted Neward har ligeledes et generelt item i Effective Enterprise Java [5]. Og kigger man på ”Core J2EE Patters”[1] er det ligeledes meget sparsomt. Et typisk og noget fattigt eksempel fra disse bøger er være caching af lookups i JNDI via en service locator.

Arkitekturprototyper

Mit arbejde med prototyper i denne opgave bygger i høj grad på principperne præsenteret i Bardram et. al. [2]. Prototyperne omkring logging er primært af en eksplorativ karakter. De forsøger, at finde det optimale sted og den optimale måde, at introducere logging i en applikation. Dette med fokus på arkitektturkvalitetsattributter som maintainability og performance. De øvrige prototyper omkring analyse og implementering af caching er i større grad af eksperimentel karakter. De handler med andre ord mere om at skabe et ”proof of concept”, for at verificere, at ideen og arkitekturen giver mening.

Dynamic Proxy

Min brug af dynamic proxies til at implementere et interceptor pattern, for at indskyde logging, analyse og caching bygger på principperne præsenteret at Brian Goetz [3] i hans artikel om dynamic proxies.

Lignende løsninger

Andre har forsøgt at implementere mere eller mindre generelle løsninger, der adresserer den samme udfordring, som jeg har skitseret i motivationen til denne opgave. Altså udfordringen at cache imellem en backend og en browserbaseret klient.

Et eksempel er AutoWebCache [19], hvor man forsøger at cache indholdet af dynamiske hjemmesider. Løsningen adskiller sig dog fra mit oplæg på et centralt område. De ønsker at cache så tæt på klienten som muligt. Det ønsker de for at opnå besparelser i form af mindre processering i fx servlet containeren, hvor JSP'er behandles eller i applikationscontaineren, hvor EJB'er behandles. Jeg har derimod valgt at ligge min caching stik modsat, nemlig så tæt på backenden som muligt. Årsagen hertil er et ønske om, at bevare en fornuftig lagdeling i applikationsarkitekturen, med henblik på bedre genbrugelighed. Fx ville en caching af et backendkald kunne genbruges, hvis præsentationsteknologien suppleres eller udskiftes. Et par eksempeler kunne være anvendelsen af en rig Swing baseret klient eller en mobil udgave. Dette ville ikke være muligt i en løsning, som ligger sig tæt op at servlet api'et, som AutoWebCache gør.

Et andet eksempel er det arbejde, som præsenteres i artiklen An Aspect Oriented Performance Analysis Environment [18]. Selvom en stor del af dette arbejde handlede om udnyttelse af AOP, og om at bygge værktøjsunderstøttelse i Eclipse, er der mange ligheder til de ting, som jeg gerne vil opnå. Fx er tilgangen til identifikation af kandidater til caching meget i stil med, hvad jeg vil forsø-

ge. De har dog valgt, at udvide ”scopet” for deres søgning til hele den applikationer, der profileres. Jeg har i stedet valgt at fokusere på DAO abstraktionen. Forskellen betyder efter min overbevisning, at min løsning vil være bedre egnet til andet end profilering, da en komplet instrumentering alt andet lige vil give et relativt voldsomt overhead i et produktionssetup. Man kunne selvfølgelig indskrænke en AOP løsning ved at benytte ”pointcuts”, der kun påvirker fx DAO abstraktionen. I forhold til implementeringen af caching er der også en væsentlig forskel. Deres fokus har været at lave simpel caching, der kan indsættes i en profileringssituation, med henblik på at påvise effekten af en caching. Deres tanke er så, at der efterfølgende skal implementeres ”rigtig” caching på det pågældende sted.

Desværre virker det ikke til, at dette ellers lovende projekt med AOP performance analyse er nået ud fra IBM's labs, hvilket er en skam, da det umiddelbart har et ret lovende potentiale.

3 Logging af backendkald

I grove træk har jeg et behov for at logge alle backendkald med henblik på efterfølgende analyse. Det handler altså om at logge input til et backendkald, samt efterfølgende output med henblik på at kunne sige noget om gentagelser, der potentielt kan caches – kandidater til caching.

Umiddelbart virker det simpelt, blot at logge alle input parametre til et metodekald og efterfølgende logge en returværdi. Der er dog et par vigtige arkitektur-kvalitetsattributter, som skal overvejes hvis løsningen skal være holdbar i praksis. Det handler om:

- Maintainability (fokus på genbrugelighed og automatik)
- Performance

Maintainability er vigtig i flere forskellige sammenhænge. Først og fremmest skal det være så let, som muligt at ændre i en applikation uden, at det giver ekstra arbejde med ny logging. Et eksempel kunne være, at udvide et backendkald med et nyt søgekriterie. Det skulle helst afføde minimalt ekstra arbejde i forhold til logging. Maintainability er også helt central i forhold til implementering af logging i eksisterende applikationer, hvor det skal være muligt at lave og ændre logging med minimale ændringer i eksisterende kode.

Performance er naturligt også en vigtig kvalitetsattribut, da logging alt andet lige vil give ringere performance end i en applikation uden logging. Det er derfor vigtigt, at logging ikke koster væsentligt for performance.

3.1 Arkitekturprototyper til logging

For at finde det optimale sted at udføre logging i en ”typisk” arkitektur i et JEE datalag, har jeg udarbejdet to forskellige arkitekturprototyper og sammenlignet dem primært på maintainability og sekundært på performance. De to prototyper er:

- Logging i DAO
- Logging via dynamic proxy

Den første prototype med logging i DAO, er samtidig udgangspunktet for den næste prototype. Målet var først at bygge et fornuftigt skelet, der kunne være udgangspunkt for den næste prototype. Samtidig var det også målet, at blive skarpere på de udfordringer, der er i forhold til maintainability i en sådan arkitektur.

Den næste prototype bygger videre på samme struktur, men forsøger at løse maintainability udfordringen bedre end den første.

3.1.1 Logging i DAO

Tanken med at placere logging per DAO, er at ansvaret bliver placeret, der hvor ansvaret for kommunikation med backends i forvejen ligger – det vil sige der, hvor input sendes ud af applikationen og svar returneres som output. Hele prototypen, inkl. de tests den er bygget på baggrund af, kan ses i Appendiks A**.**

Implementering

Prototypen indeholder et simpelt DAO interface – ***BackendDao***, der udstiller tre metoder til at finde og gemme objekter af typen ***BackendObject***:

- findObjectById(String id) – returnerer BackendObject
- findObjectsByIds(String[] ids) – returnerer List<BackendObject>
- persistObject(BackendObject object) – returnerer void

Klienter får adgang til instanser af ***BackendDao*** igennem en ***DaoFactory***, som har til ansvar at lave de konkrete instanser. Det at lave instanser indkapsles i en factory, for at skjule detaljer om prototypens logger – ***SimpleLogger***, der uddybes senere.

Et eksempel på en metode i DaoFactory, som sender den korrekte logger med:

```
public BackendDao getBackendDao() {
        return new BackendDaoImpl(logger);
}
```

I et mere realistisk eksempel ville denne factory også indkapsle forskellige implementeringer af ***BackendDao*** interfacet. Det kunne fx være en implementering med et ”mockobject”, som er et eksempel på en teknik, der gør det muligt at afkoble applikationen fra en ”rigtig” backend. Et ”mockobject” simulerer et rigtigt objekt på en kontrolleret måde. Denne teknik anvendes ofte med henblik på, at afkoble dele af applikationen fra forskellige backendsystemer, som fx en databaseserver. Afkoblingen gør det muligt at lave ”rigtige” unit tests, som ikke laver integrationstest som en sideeffekt.

Når klienter kalder en af de tre metoder på ***BackendDao***, er det op til den konkrete implementering i ***BackendDaoImpl*** at logge de forskellige input og output. Denne logging sker ved at kalde metoden log(String entry) på den logger instans, som factory'en har indskudt da DAO instansen blev lavet.

Et eksempel:

```
public BackendObject findObjectById(String id) {
        //Her ville der fx bygges en SQL forespørgsel og efterfølgende et BackendObject
        BackendObject object = new BackendObject();
        logger.log(id + object);
        return object;
}
```

SimpleLogger indkapsler selve funktionaliteten til logging, således at detaljerne skjules for resten af applikationen. Det vil sige at der fx kan skiftes imellem konkret logging til memory eller en logfil uden, at det kræver ændringer øvrige steder i applikationen. I prototypen er ***SimpleLogger*** blot implementeret med en instansvariable af typen ArrayList<String>, og udstiller udover metoden log(String entry) en boolean contains(String entry) metode. Metoden contains bruges kun til test i prototypen.

Et eksempel på et flow i prototypen kan ses på illustration 2. Her laver en klient en ***BackendDao*** instans via en DaoFactory, hvilket sammensætter en ***BackendDaoImpl*** med en ***SimpleLogger***. Efterfølgende kalder klienten findObjectById, hvilket laver et ***BackendObject*** ud fra id. Input og output logges inden det "fundne" objekt sendes tilbage til klienten.

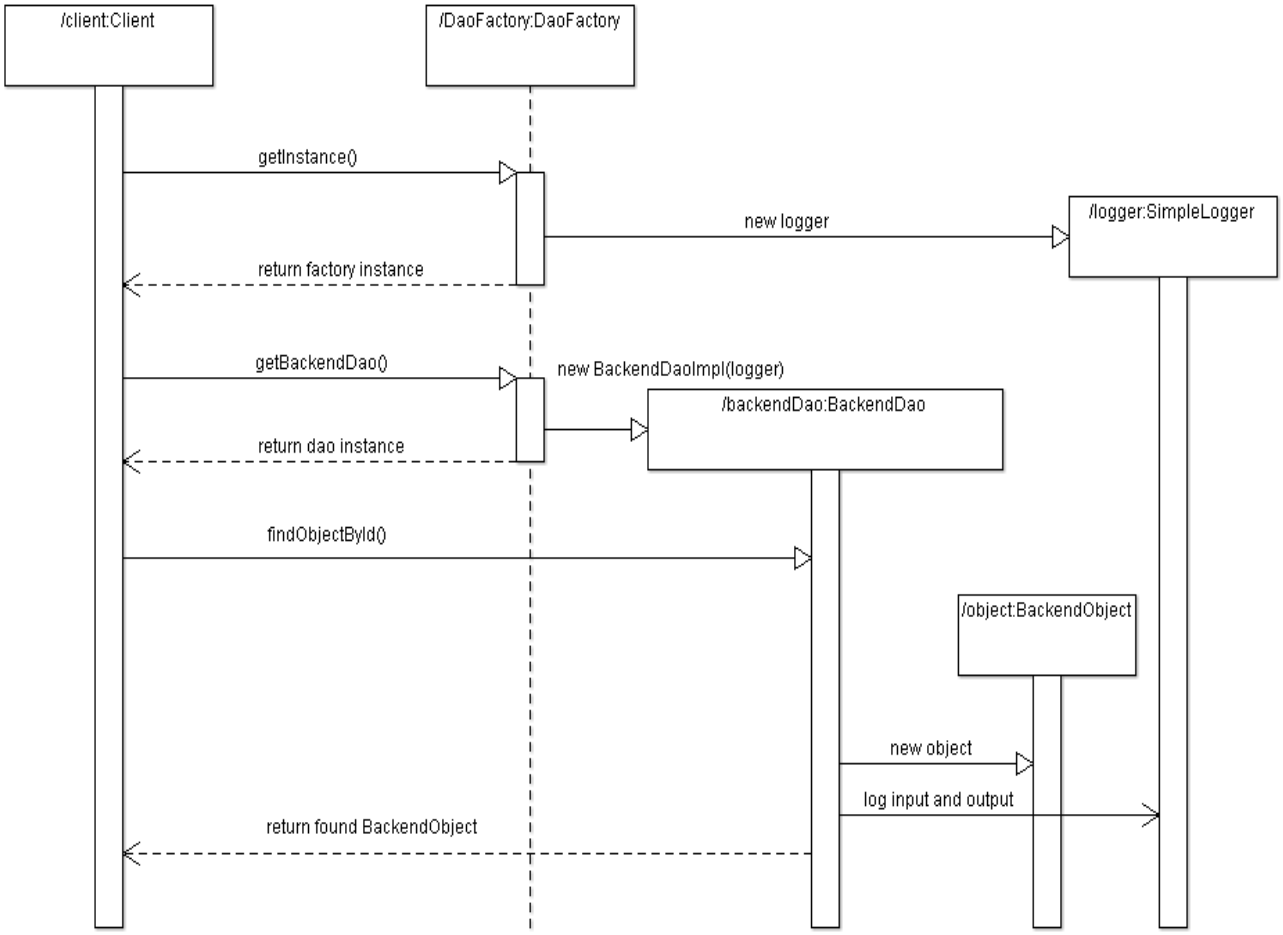


Illustration 2: Sekvensdiagram for oprettelse af DAO objekt og efterfølgende logging af kald til read metode

Det betyder altså, at hele ansvaret for, hvad og hvornår der logges, ligges i hver enkelt konkret DAO. Selve logging funktionaliteten er så yderligere isoleret i ***SimpleLogger.*** Det betyder at, hvordan der konkret logges kan ændres uden indflydelse på resten af applikationen.

Fordele

I en situation hvor man ønsker, at ændre *hvordan* logging skal foregå, virker løsningen fornuftig i forhold til maintainability, da selve logging fint er indkapslet i ***SimpleLogger***. Endvidere har vi fuld kontrol over ”*hvad og hvornår”* de konkrete metoder skal logge. Fx kan vi positivt fravælge logging af kald til ”write” metoder, der som udgangspunkt ikke giver mening at cache.

I forhold til performance er det svært at se en løsning, som kan være billigere, da det eneste overhead er selve logging funktionaliteten, som alt andet lige ikke kan være gratis.

Ulemper

Kigger man på *hvad og hvornår* der skal logges, er det umiddelbart en ulempe, at det er spredt til hver enkelt metode. Det betyder i praksis, at logging giver ekstra arbejde, hver gang man skriver en ny metode på en DAO, som har brug for caching. At vi har fuld kontrol over, hvad der logges, betyder at vi har risiko for, at det bliver svært/umuligt at lave en genbrugelig analyse efterfølgende.

At alle metoder selv har ansvaret betyder ligeledes, at en implementering af logging i en eksisterende applikation ville kræve tilføjelser i samtlige DAO klasser. Det vil endvidere betyde risiko for, at man introducere nye fejl.

Et andet minus ved løsningen er, at der ingenting findes i arkitekturen, der tvinger en udvikler til at implementere en passende logging. Det vil sige, at man i en maintainability situation, hvor der fx udvikles en ny ”findByName” metode, er risiko for, at der ingen logging implementeres. Det kan muligvis løses igennem code reviews eller lignende, men det vil så skulle ske for samtlige metoder.

Konklusion

Opsummeret giver løsningen ikke et tilfredsstillende niveau for maintainability, da man har ekstra arbejde ved hver tilføjelse. Samtidig giver implementering i eksisterende applikationer også ekstra arbejde og risiko for fejl. Herudover er der ingen lette muligheder for kontrol af implementering. Dog giver løsningen ingen performance udfordringer.

3.1.2 Logging via dynamic proxy

Ideen med den næste prototype er, at flytte ansvaret for logging ud af de enkelte metoder i en DAO. Tanken er at fange alle kald til en DAO. For hvert kald logges al input og al output imellem klienten og selve DAO instansen. Logging ”indskydes” altså imellem klienten og DAO instansen uden, at nogle af de to parter påvirkes. Hele prototypen, inkl. de tests den er bygget på baggrund af, kan ses i Appendiks B**.**

Implementering

For at indskyde logging imellem klienten og DAO instansen, har jeg valgt at implementere en dynamic proxy [3]. Tanken bag en dynamic proxy er, at den laves på runtime via Java's reflection API. Det vil sige, at man ud fra en klasse eller et interface får en proxy, som kan ”spille” den pågældende

klasse/interface i forhold til en klient. Klienten kommunikerer så med en proxy, der har mulighed for at opsnappe alle kald igennem en central ”invoke” metode. I praksis bindes den dynamiske proxy sammen med en såkaldt InvocationHandler på det tidspunkt, hvor den laves. Det er så den konkrete InvokationHandler, der får kald til proxy instansen igennem metoden ”invoke”.

I forhold til sekvensen i illustration 2 i afsnit 3.1.1 ændres der ikke på oprettelsen af en DAO set fra klientens side. I stedet for blot at returnere en ***BackendDaoImpl*** indsættes følgende ekstra metode:

```
public BackendDao getBackendDao() {
      return getBackendDaoProxy(new BackendDaoImpl());
}
private BackendDao getBackendDaoProxy(final BackendDao dao) {
      return (BackendDao) Proxy.newProxyInstance(
            dao.getClass().getClassLoader(),
            new Class[] { BackendDao.class },
            new LoggingInvocationHandler<BackendDao>(dao, logger));
}
```

Det vigtige her er den ***LoggingInvocationHandler***, der laves og sendes med til factorymetoden newProxyInstance. Her tager Java's reflection API kontrollen, og sørger for, at alle kald til vores proxy resulterer i et kald til metoden invoke. ***LoggingInvocationHandler*** har invoke metoden, da den implementerer interfacet java.lang.reflect.InvocationHandler.

En ekstra detalje er, at jeg har valgt at lave ***LoggingInvocationHandler*** generisk. Den erklæres her med typen ***<BackendDao>.*** I praksis kunne man bruge den sammen med en hvilken som helst anden proxy. Den samme ***LoggingInvocationHandler*** kan altså bruges til forskellige DAO typer.

Det næste skridt i prototypen er at erstatte al logging med håndtering i invoke, der kaldes hver gang fx metoden findObjectById(String id) kaldes. Metoden invoke har følgende signatur:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Invoke modtager altså selve proxy objektet, metoden der blev kaldt og dens parametre - ”method” og ”args” udnytter jeg til at bygge følgende del af en tekst til logging:

caching.dao.BackendDaoImpl.findObjectById: objectId ->

Herefter sendes kaldet videre til det rigtige objekt, som ***LoggingInvocationHandler*** modtager i sin constructor. Det sker på følgende måde:

```
Object ret = method.invoke(underlying, args);
```

Underlying er i dette tilfælde en instans af ***BackendDaoImpl***. Resultatet af metodekaldet gemmes i variablen ***ret,*** der nu tilføjes til teksten, der skal logges. Det giver fx en logentry som denne:

caching.dao.BackendDaoImpl.findObjectById: objectId -> caching.dao.BackendObject @360be0

Formatet er *klassenavn.metodenavn: param1, .. paramN -> returnobject*

I denne prototype er både parametre og returnobject blot logget via deres nedarvede toString metode fra java.lang.Object. Hvorvidt, det er nok, er ikke relevant for denne prototype, men det undersøges senere i afsnit 4 om analyse af logs. Resultatet er at alle konkrete DAO instanser ikke længere forholder sig til logging, som det var tilfældet i den første prototype med logging i DAO.

Fordele

Den helt store fordel ved denne prototype er, at ansvaret for logging er flyttet fra enkelte metoder i konkrete DAO implementationer til ét centralt sted i en InvocationHandler. For maintainability betyder det fx, at der intet ekstra arbejde er ved at tilføje en ny metode i en DAO. Ligeledes skal der heller ikke rettes i eksisterende applikationer, hvis man ønsker at tilføje logging.

I forhold til performance må man sige, at prisen for at kalde igennem en proxy er minimal. Jeg har lavet et meget simpelt forsøg, hvor jeg har indsat et kald til System.currentTimeMillis() før og efter et kald til findObjectById. Det har jeg gjort i begge prototyper, og resultatet var i den første med logging i DAO ikke målbart – dvs kaldet tog 0 millisekunder. I prototypen med en dynamic proxy tog samme logging 1 millisekund de fleste gange jeg prøvede. Ca. hver 10 forsøg gav dog et resultat på 0 millisekunder. Derfor er min konklusion, at det ikke får nogen nævneværdig indflydelse på performance.

Ulemper

I forhold til maintainability skal der ved hver ny DAO tilføjes en proxy. Det betyder, at der i eksisterende applikationer skal rettes i DAO factories, således at de laver proxies. Et alternativ kunne dog være at introducere et nyt interface, som samtlige DAO'er implementerer. Dette ville kunne gøre det muligt, at bruge én metode til oprettelse af proxy og InvocationHandler. Faktisk er det sådan prototypen allerede indirekte gør. BackendDao interfacet skal blot være generelt for alle DAO'er.

Der opstår et behov for, at man opfinder en filtrering af den generelle logging, hvis man ikke ønsker at alle ”write” kald skal logges. Dog kan den slags kald i nogle tilfælde være interessante alligevel, hvis man fx tilføjede ”execution time” til loggen. Det ville give mulighed for at analysere svartider på alle backendkald.

Konklusion

Overordnet er det en meget acceptabel løsning, hvor den største ulempe er for maintainability i forbindelse med implementering i en eksisterende applikation. Her vil der dog kun skulle rettes i de eksisterende DAO factories, hvilket må siges at være et ret begrænset i omfang. Til gengæld er ansvaret for logging pænt isoleret til ét sted og helt skjult for både klient og DAO implementeringer. Endelig er der ikke nogen nævneværdig performance omkostning i forhold til introduktion af logging.

3.2 Delkonklusion på logging

I Tabel 1: Opsummering af logging prototyper har jeg opstillet de vigtigste målepunkter for de to prototyper. Umiddelbart er der én væsentlig forskel på det to løsninger, som er mængden af arbejde i forbindelse med implementering i en eksisterende applikation. Her vil en løsning, hvor der indsættes logging i hver enkelt DAO, simpelthen blive for omfattende. Det ville kræve rettelser alt for mange steder, og dermed også introducere alt for stor risiko for fejl. Derimod er virker løsningen med en dynamic proxy ganske overbevisende og helt klart noget, som jeg vil arbejde videre med i de øvrige prototyper.

Prototype/Kvalitet	**Ny funktion/Ny DAO (maintainability)**	**Implemtering i eksisterende app. (maintainability)**	**Performance**
Logging i DAO	Acceptabelt – en til en	Uacceptabelt – en til en	Acceptabelt - minimum
Logging via dynamic proxy	Acceptabelt – en per DAO/factory	Acceptabelt – en per DAO/factory	Acceptabelt – tæt på ingen påvirkning

Tabel 1: Opsummering af logging prototyper

4 Analyse af logs

I dette afsnit vil jeg forsøge at bygge en prototype, der kan analysere input og output med henblik på at finde kandidater til caching. For at bygge en analyse der giver mening, har jeg dog først brug for at finde ud af, hvilke mønstre disse kandidater kunne følger. Det vil jeg gøre ved at beskrive nogle ad hoc analyser, som jeg har foretaget af en samling kørende applikationer.

4.1 Ad hoc analyser

Analyserne blev udarbejdet i forbindelse med et projekt, hvor målet primært var oprydning og vedligeholdelse i en hel række kørende applikationer. Et af emnerne for projektet var performance. Vi havde reelt ikke overblik over, hvordan vores applikationer performede, udover hvad vi kunne opleve igennem brugertest. Vi forsøgte at arbejde med automatiserede brugertests igennem Selenium[20], som er et udbredt testing framework til webapplikationer. Her lykkedes det os at identificere en række steder med dårlige svartider. Vi fik hurtigt mistanke til, at vores backendkald til mainframe via MQ kunne være en af synderne.

For at blive helt skarpe på de faktiske tal og for at identificere de konkrete kald, besluttede vi os for at indsætte log statements i forbindelse med hvert kald. Vi indsatte logging i vores framework til MQ kommunikation, som bliver brugt af langt størstedelen af de applikationer, som vores projekt beskæftigede sig med. I første omgang var det på følgende form:

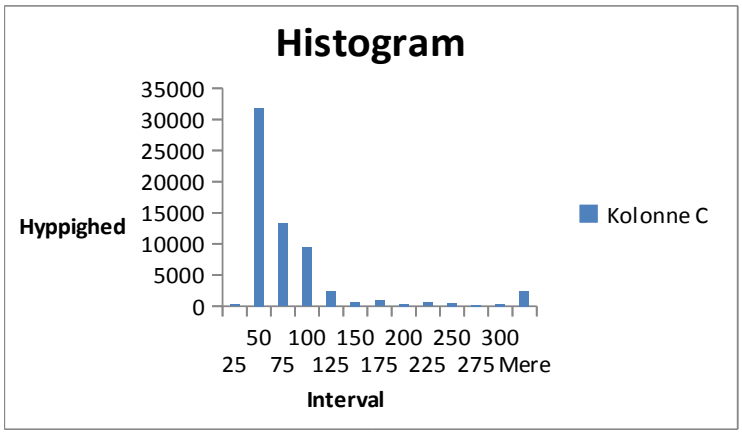
```
jyskeprodukter_da1642-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** INFO  2010-01-
14 03:42:20,044 (AbstractJMSDAO.java:220) - getSeveralRecords Time elapsed for;
hentProduktoversigt; 47; milliseconds; BrugerId 0046317001
```

Ovenstående log entry er genereret via Log4J[21], som er et hyppigt anvendt framework til logging i Java. Den første del frem til ":" er navnet på den logfil, som den konkrete entry er fra. Filnavnet er medtaget, fordi ovenstående logstatement er udtrukket fra logfiler på fire forskellige servere. Informationen om server er relevant i forhold til kontrol af loadbalancering og performance fra server til server. Herefter følger tre standard elementer fra Log4J: loglevel, et timestamp og Java klassen der er udskrevet fra. Endelig kommer vores "egen" tekst, som består af en generel tekst, funktionsnavn, tid i millisekunder og endelig brugerid.

For at analysere de konkrete funktionskald importerede vi alle entries for en given funktion til MS Excel, hvor jeg byggede en template til import og efterfølgende statistisk analyse af resultaterne. Et eksempel resume af statistiske nøgletal fra denne template kan ses i Tabel 2. Udover de rå nøgletal blev der også genereret et histogram, der viser fordelingen af observerede svartider i intervaller. Et eksempel på et histogram kan ses i Illustration 3. Se Appendiks C – Forklaring til statistiske nøgletal for yderligere detaljer omkring de enkelte nøgletal og deres sammenhænge. Disse statistiske analyser dannede grundlag for vores efterfølgende analyser og fejlrettelser i projektet.

Middel	91,24
Standardafvigelse	200,78
Median	47
Skævhed	15,7
Maks	8906
Min	15
Antal	63238
Antal watch (BrugerID=0046317001)	7276 (11,51%)
Antal jb-brugere	8675 (13,72%)
Periode start: 14-01-2010	Dage: 33,00
Periode slut: 17-02-2010	

Tabel 2:Eksempel på resume af statistiske nøgletal for en funktion



Interval	*Hyppighed*
25	237
50	31830
75	13300
100	9409
125	2454
150	634
175	901
200	351
225	606
250	516
275	200
300	379
Mere	2421
	63238

Illustration 3:Eksempel på histogram over svartider fra ad hoc performance analyse

4.1.1 Identificerede mønstre

Vi fandt ret hurtigt ud af, at vi ikke generelt havde dårlige svartider per kald. Derimod dukkede der andre interessante mønstre op, som jeg vil beskrive i det følgende afsnit.

Mange kald fra "watch-bruger"

Da vi i Jyske Bank anvender et overvågningskoncept, hvor der med faste intervaller skyldes mod en "watch" servlet, rammes en del af vores funktionskald også ret heftigt fra denne kant. Fx kan man se i Tabel 2, at watch faktisk står for 11,51% af alle kald. Det betyder, at vi er nødt til at frasortere

disse kald, hvis vi vil kigge på performance for vores slutbrugere. Det betyder også, at vi fx ikke bør cache denne brugers kald – da et af formålene med watch er at teste afhængigheder til forskellige backend systemer. Der er altså tale om en central overvågning, der ved fejl rapporterer til vores døgnbemandede overvågning i vores driftsselskab JN DATA.

I praksis betød vores observationer, af hvor stort load watch lavede på vores systemer, at vi generelt fik ryddet op i mængden af forskellige funktioner, der blev kaldt i en watch situation. Der blev ofte kaldt samtlige funktioner i en applikationer, selvom de i praksis testede afhængigheden til det samme backendsystem – i dette tilfælde vores MQ kommunikation til Cobol programmer på mainframe. Vi fik dermed reduceret vores ressourceforbrug til watch betragteligt, blot ved at få det gjort synligt.

Men observationen betyder, at en analyse bør kunne "frasortere" uønskede input/output i de tilfælde, hvor der er tale om en "speciel" bruger som fx watch. Dette er især vigtigt i forhold til en evt. caching, som ikke må ske for denne specielle bruger.

Flere kald til samme funktion på "samme" tid

Hvis den samme funktion kaldes flere gange på samme tid, kan der ofte være tale om en fejl eller en dårlig implementering. Et eksempel på en sådan fejl/implementering kunne være følgende JSP kode:

```
<table class="KundeInfoTable">
 <tr><td class="KundeInfo"><%=dk.jyskebank.budget.kontekst.KundeUtil.getStilling(session)%></td></tr>
 <tr><td class="KundeInfo"><%=dk.jyskebank.budget.kontekst.KundeUtil.getNavn(session)%></td></tr>
 <tr><td class="KundeInfo"><%=dk.jyskebank.budget.kontekst.KundeUtil.getGade(session)%></td></tr>
 <tr><td class="KundeInfo"><%=dk.jyskebank.budget.kontekst.KundeUtil.getPostnrBy(session)%></td></tr>
</table>
```

Som udgangspunkt er der ingenting galt med JSP-koden, det der derimod går galt er at KundeUtil ikke nødvendigvis var designet til dette brugsmønster. Den gemmer nemlig ikke tilstand, men kalder den samme backendfunktion for hver af de fire linier. Resultatet kan ses i det følgende eksempel fra logs, hvor hentCKNavn kaldes fire gange:

```
jyskeprodukter_da1642-cp1_jb.ind.ebk.bm.ear-V1-7-23.log.4:** INFO 2009-12-20 00:04:02,439 (?:?) -
getOneRecord: Time elapsed for; hentCKNavn; 78; milliseconds
jyskeprodukter_da1642-cp1_jb.ind.ebk.bm.ear-V1-7-23.log.4:** INFO 2009-12-20 00:04:02,502 (?:?) -
getOneRecord: Time elapsed for; hentCKNavn; 47; milliseconds
jyskeprodukter_da1642-cp1_jb.ind.ebk.bm.ear-V1-7-23.log.4:** INFO 2009-12-20 00:04:02,564 (?:?) -
getOneRecord: Time elapsed for; hentCKNavn; 31; milliseconds
jyskeprodukter_da1642-cp1_jb.ind.ebk.bm.ear-V1-7-23.log.4:** INFO 2009-12-20 00:04:02,642 (?:?) -
getOneRecord: Time elapsed for; hentCKNavn; 62; milliseconds
```

Hvis ikke man logger alle backend kald, kan det være svært at få øje på den slags problemer. Der er trods alt kun tale om 78+47+31+62=218 millisekunder i total svartid. Faktisk var det ovenstående eksempel, der fik os til at udvide vores logging med et brugerid. Det skete fordi, man ikke umiddelbart har nogen chance for at vide, om ovenstående er et udtryk for en bruger eller bare flere samtidige.

Flere kald til samme funktion for samme bruger indenfor samme ”session”

Efter at have indsat brugerid i vores logging, begyndte vi at finde flere eksempler på gentagne kald fra den samme bruger. I modsætning til det tilfælde, hvor kaldene sker indenfor samme sekund, sker disse kald typisk indenfor samme ”session”. Session forstået som den tid en given bruger er logget ind i en applikation. Et eksempel kan ses i de efterfølgende logstatements:

```
jyskeprodukter_da1642-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** INFO  2010-01-14 08:12:00,190
(AbstractJMSDAO.java:220) - getSeveralRecords Time elapsed for; hentProduktoversigt; 47; milliseconds;
BrugerId xxxxxxxxxx
jyskeprodukter_da1642-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** INFO  2010-01-14 08:13:38,002
(AbstractJMSDAO.java:220) - getSeveralRecords Time elapsed for; hentProduktoversigt; 63; milliseconds;
BrugerId xxxxxxxxxx
jyskeprodukter_da1642-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** INFO  2010-01-14 08:15:06,923
(AbstractJMSDAO.java:220) - getSeveralRecords Time elapsed for; hentProduktoversigt; 63; milliseconds;
BrugerId xxxxxxxxxx
```

Her er der tre kald til funktionen ”hentProduktoversigt” indenfor ca. 3 minutter. Udfordringen med at finde dette mønster, er at der sagtens kan være adskillige andre, der kalder samme funktion i løbet af de tre minutter. Ved at tilføje brugerid til vores logging, fik vi altså mulighed for at filtrere per bruger, hvilket gjorde det lettere at finde dette mønster.

Øvrige observationer

Umiddelbart viste vores analyser også et par andre tilfælde af dårlig performance. Fx fandt vi ud af at der omkring årsafslutning og kvartalsafslutning kan være store udfordringer med performance. Vi observerede også, at der generelt kan være dårligere performance på mindre brugte funktioner, hvilket skyldes manglende caching på backendsiden. Det kunne fx være caching af Cobol programmer, initialisering af datastores, caching på DB2 osv.

4.2 Prototype til log analyse

Umiddelbart viste vores analyser altså at vores største udfordring var gentagne kald, og ikke fx langsomme kald. Derfor bliver det primære mål for min prototype til log analyse, at kunne finde gentagne funktionskald, hvilket dækker over to af de observerede mønstre.

- **Flere kald til samme funktion på ”samme” tid**
- **Flere kald til samme funktion for samme bruger indenfor samme ”session”**

Det grundlæggende problem er altså det samme, blot med en variation i hvor hurtigt der gentages. Kildekoden til prototypen kan i sin helhed ses i Appendiks D – Kildekode til log analyse og central caching, I det efterfølgende afsnit skitseres prototypens vigtigste elementer og mine erfaringer med at bygge den.

Implementering

Først har jeg bygget en LogAnalyzer, der ad hoc kan analysere en given log, med henblik på følgende:

- finde unikke funktionskald
- finde antal forekomster af et unikt kald
- beregne gennemsnitligt svartid for et kald

For at gøre det muligt for en LogAnalyzer, at analysere på en log, var det nødvendigt for mig at gøre loggingen fra min prototype i afsnit 3 mere detaljeret. I stedet for blot at logge en String, har jeg brug for flere detaljer. Disse detaljer blev løbende indarbejdet i prototypen med TDD[23], hvilket vil sige, at jeg løbende skrev tests efterfulgt af kode og refactoring. Altså hele tiden små iterationer af følgende skridt: ***skriv test->se rød test->implementer->se grøn test->refactor->se grøn test***. På den måde fik jeg skabt min nye logger – InvocationLogger, samt dens entries – InvocationLogEntry.

En InvocationLogEntry tager udgangspunkt i at registrere et metodekald. Den indeholder følgende instansvariabler:

```
private final String invocationString;
private final String fullyQualifiedMethodName;
private final Object[] args;
private final Object returnValue;
private final long executionTimeInMilliseconds;
private final long creationTime;
```

Instantiering sker via createEntry, som er en statisk factory metode[24]. Der laves et immutabelt objekt[24]. Det giver frihed til at lade en InvocationLogger udstille entries til interesserede klasser, som fx en LogAnalyzer uden risiko for opdateringer. Hvis ikke disse entries er immutable, kan to interesserede klasser skabe problemer for hinanden ved opdateringer. Instantieringen pakkes ind via en statisk factory metode, hvilket giver mulighed for afskære nedarv ved at beskytte constructoren. Dette gøres for at bevare immutabiliteten af InvocationLogEntry objekter.

InvocationLogEntries skabes i min DaoInvocationHandler, som er en videreudvikling af min LoggingInvocationHandler fra afsnit 3. Hver gang en DAO kaldes udføres følgende kode:

```
public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {

      long before = System.currentTimeMillis();
      Object ret = method.invoke(underlying, args);
      long after = System.currentTimeMillis();
      String fullyQualifiedMethodName = underlying.getClass().getName()
      +"."+method.getName();
      logger.log(InvocationLogEntry.createEntry(fullyQualifiedMethodName,
                  args, ret, after-before));
      return ret;
}
```

Der opsamles et timestamp inden selve kaldet udføres og et efter, hvilket giver mulighed for at skabe en svartid for metodekaldet. Metodenavn, argumenter, returværdi og svartid bruges herefter til at skabe en InvocationLogEntry, der tilføjes til min InvocationLogger.

LogAnalyser klassen beskrives bedst igennem dens test, som indeholder følgende:

canReportAverageResponseTimeForMultipleInvocationsOfOneMethod()
canReportNumberOfEntries()
canReportNumberOfOccurencesForAnEntry()
canReportNumberOfOccurencesIfNoEntryWasFound()
canReportNumberOfUniqueEntries()
canReportUniqueEntriesByOccurence()

Selve implementeringen er ret enkel. LogAnalyser instatieres med den logger, som man ønsker at undersøge. Herefter er det inspektion af InvocationLogEntry objekter i loggeren via iteration og sammenligning vha. equals. InvocationLogEntries opfattes som equal, hvis de har samme invocationString. Det vil sige metodenavn, argumenter og returnværdi. Svartid og oprettelsestidspunkt holdes ude, hvilket giver mulighed for at tælle unikke entries og for at beregne en gennesnitlig svartid.

Et eksempel på en konkret implementation i LogAnalyser er de unikke entries:

```
public Set<InvocationLogEntry> reportUniqueInvocationEntries() {
	Iterator<InvocationLogEntry> iterator = logger.getIterator();
	HashSet<InvocationLogEntry> uniqueEntries = new
		HashSet<InvocationLogEntry>();
	while (iterator.hasNext()) {
		InvocationLogEntry entry=(InvocationLogEntry) iterator.next();
		uniqueEntries.add(entry);
	}
	return uniqueEntries;
}
```

Her udnytter jeg java.util.Set, der igennem add sikre at kun unikke InvocationLogEntry objekter tilføjes. Det sker via equals. De øvrige metoder kan ses i Appendiks D.

DuplicateInvocationReporter

Efterfølgende har jeg udvidet min analyse med en DuplicateInvocationReporter, som automatisk kan rapportere om gentagne funktionskald. Reporteren er knyttet sammen med loggen via et klassisk observer pattern[28].

Jeg har således udvidet min InvocationLogger til at være Observable, hvilket betyder at den arver fra java.util.Observable. Jeg har valgt den indbyggede model i Java, hvilket ikke er optimalt i alle situationer. Ulempen er fx at InvocationLogger får en tæt kobling til Observable, hvilket ikke er en ”kerne” funktionalitet for en Logger. Den bliver endvidere afskåret fra yderligere arv, fra andre klasser. Disse udfordringer kunne løses ved at vælger en implementering, der benytter delegering frem for arv. Jeg har dog valgt at ”betale” prisen i min prototype, for at komme hurtigere i mål.

For at teste implementeringen af min Observable, har jeg valgt at benytte jMock[25]. Ved at anvende et mocking framework, som jMock, får jeg mulighed for at beskrive og teste samspillet imellem de involverede objekter. Testen i min InvocationLogger se således ud:

```
@Test
public void updatesObserversWhenLoggingAnEntry() {
	Mockery context = new Mockery();
	final Observer observer = context.mock(Observer.class);

	final InvocationLogger logger = InvocationLogger.getInstance();
	logger.addObserver(observer);
	final InvocationLogEntry entry =
	InvocationLogEntry.createEntry("invocationString", 0);
	// expectations
	context.checking(new Expectations() {{
		oneOf (observer).update(logger, entry);
	}});
```

```
        // execute
        logger.log(entry);
        // verify
        context.assertIsSatisfied();
    }
```

Testen laver først de objekter, der skal spille sammen: en Observer, en InvocationLogger og en InvocationLogEntry. Observeren tilføjes på loggeren. Herefter beskrives forventningerne, som er at update bliver kaldt præcis én gang med loggeren og vores entry. Efterfølgende sker selve kaldet til logger.log(entry). Som det sidste skridt kaldes assertIsSatisfied, hvilket tjekker den opsatte forventning. Hvis ikke update kaldes med de rigtige objekter præcis én gang fejler testen. Selve implementeringen i InvocationLogger er forholdsvis simpel:

```
    public void log(InvocationLogEntry entry) {
        entries.add(entry);
        setChanged();
        notifyObservers(entry);
        clearChanged();
    }
```

Selvom ser simpelt ud, udstillede testen meget godt, at jeg personligt har arbejdet for meget med webapplikationer og standard frameworks i en lang periode. Det tog nemlig et par forsøg inden testen blev grøn, da jeg lige skulle ”huske”, hvad der skulle til i Java's observer implementering.

På modtager siden gav det følgende update metode i DuplicateInvocationReporter:

```
    @Override
    public void update(Observable o, Object arg) {
        String invocationString = ((InvocationLogEntry)
                                        arg).getInvocationString();
        if (uniqueEntries.containsKey(invocationString)) {
            Integer newCount = uniqueEntries.get(invocationString) + 1;
            uniqueEntries.put( invocationString, newCount);
            checkThresholdAndReportIfExceeded(newCount, invocationString);
        } else {
            uniqueEntries.put(invocationString, 1);
        }
    }
```

Metoden viser hvordan DuplicateInvocationReporter gemmer en reference til hver unik invocationString og en tæller for hver observation. Udover at tælle antallet af unikke invocations, rapporteres der også hvis tælleren overskrider et ”threshold”. Rapporteringen sker til System.out i prototypen og threshold er default sat til 1.

CallSimulator

For at kunne teste DuplicateInvocationReporter og senere se effekten af caching, har jeg bygget en CallSimulator, der kan ”afspille” en logfil. Simulationen er hårdt koblet til én bestemt metode, der kaldes for hver ny linie i logfilen. Jeg har valgt findCustomerById på CustomerDao.

Den logfil jeg har valgt til min test, er et uddrag af en produktionslog fra et kørende system. Loggen er manipuleret i forhold til de brugerids, der normalt findes heri da det er rigtige kunder. Men udover brugerids, er loggen et realistisk brugsmønster for funktionen ”hentProduktoversigt”. Funktionen anvendes til at hente en liste af produkter for en bestemt kunde. Produkterne ændres sjældent, da der er tale om ”rettigheder” til områder i netbanken, der typisk har en levetid på 6 måneder til flere år.

Loggen består af i alt 360 linier, som er sammensat af hvilken logfil den er udtrukket af, dato og tid, funktionen der er kaldt, tiden det tog i millisekunder og det brugerid der kaldet. Fx ser et par af linierne således ud:

```
jyskeprodukter_da1642-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:20:23,63 hentProduktoversigt     250   5441
jyskeprodukter_da1643-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:21:45,80 hentProduktoversigt     359   4786
jyskeprodukter_da1643-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:22:12,39 hentProduktoversigt     63    4786
jyskeprodukter_da1643-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:22:14,44 hentProduktoversigt     62    8151
jyskeprodukter_da1644-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:22:37,47 hentProduktoversigt     94    4493
jyskeprodukter_da1644-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:22:58,78 hentProduktoversigt     453   685
jyskeprodukter_da1642-cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:** 14-
01-10 09:24:59,90 hentProduktoversigt     47    4493
```

CallSimulator instantieres med et fil objekt og en CustomerDao, hvorefter metoden playLog kalder findCustomerById for hver linie i filen. Selve playLog metoden blev ret enkel:

```
public void playLog() throws Exception {
      String currentLine = "";
      while ((currentLine = readLine()) != null) {
            dao.findCustomerById(getUserFromLine(currentLine));
      }
}
```

Det enkle design og implementering i CallSimulator, skyldes efter min bedste overbevisning, at jeg fik bygget nogle gode tests, inden jeg skrev en eneste linje kode. Altså et godt resultat, som følge en disciplineret anvendelse af TDD. Den centrale test er der, hvor jeg beskriver samspillet imellem CallSimulator og kald til findCustomerById.

```
@Test
public void willCallDaoAsManyTimesAsLogHasEntriesWhenStarted() throws
Exception {
      Mockery context = new Mockery();
      final CustomerDao dao = context.mock(CustomerDao.class);
      final CallSimulator simulator = new CallSimulator(new
            File(LOGFILE_NAME), dao);
      // expectations
      context.checking(new Expectations() {{
            exactly(360).of
            (dao).findCustomerById(with(any(String.class)));
      }});
      // execute
      simulator.playLog();
      // verify
      context.assertIsSatisfied();
}
```

Igen har jeg valgt at bruger jMock til at teste samspillet. Først laves der en mock til min CustomerDao, som efterfølgende bruges til at lave en ny CallSimulator. Derefter opsættes forventningen til at findCustomerById kaldes præcis 360 gange. Selve playLog kaldes, og der verificeres til sidst. Det var denne test, der gjorde mig i stand til at refactor playLog, til den enkle udformning den har nu. I starten havde jeg fx en String mere i spil, til at holde styr på den sidste valide linje i loggen. Resten af TestCallSimulator kan ses i Appendiks D.

Som nævnt tidligere, er formålet med CallSimulator at teste DuplicateInvocationReporter, hvilket jeg har gjort med følgende test:

```
@Test
public void reportsDuplicateInvocationsWhenCallSimulatorCalls() throws
Exception {
      DaoFactory daoFactory = DaoFactory.getInstance();
      CallSimulator simulator = new CallSimulator(new
                        File(LOGFILE_NAME) ,daoFactory.getCustomerDao());
      simulator.playLog();
      LogAnalyzer analyzer = new LogAnalyzer(daoFactory.getLogger());
      int numberOfEntries = analyzer.reportNumberOfEntries();
      assertEquals("There should be 360 entries", 360, numberOfEntries);
      System.out.println("Number of unique entries:
      "+analyzer.reportNumberOfUniqueEntries()+" out of total entries:
      "+numberOfEntries);
}
```

Umiddelbart er den første del, indtil LogAnalyzer instantieres, nok til at teste. Resultatet er en udskrift til System.out, hver gang findCustomerById kaldes med samme parameter som den tidligere er kaldt med. De tre sidste ser således ud:

```
Duplicated invocation - count at: 2 for invocationString:
caching.dao.CustomerDaoRandomImpl.findCustomerById: [6921] ->
caching.dao.Customer@196861
Duplicated invocation - count at: 2 for invocationString:
caching.dao.CustomerDaoRandomImpl.findCustomerById: [7970] ->
caching.dao.Customer@19dd5a
Duplicated invocation - count at: 3 for invocationString:
caching.dao.CustomerDaoRandomImpl.findCustomerById: [7970] ->
caching.dao.Customer@19dd5a
```

LogAnalyzer er efterfølgende anvendt, for at verificere at der blev kaldt 360 gange, samt for at udskrive antallet af unikke kald:

```
Number of unique entries: 206 out of total entries: 360
```

Det betyder altså, at der i den anvendte log findes 360 – 206 = **154 redundante kald**. Disse kald vil jeg forsøge at reducere i afsnit 5, når jeg implementerer en prototype til central caching.

4.3 Evaluering af log analyse

Der er stor forskel på de ad hoc analyser, som jeg har beskrevet først i afsnittet og den prototype som jeg har implementeret. For det første er præcisionen væsentlig større i forhold til at sammenligne kald. De manuelle analyser kiggede udelukkende på funktionsnavn og den bruger der kaldte. I min prototype tager jeg både højde for input og output, hvilket giver et mere rigtigt billede af om de pågældende kald ret faktisk er identiske. Det ville være vanskeligt, at skabe den præcision med en ad hoc analyse, medmindre man først generaliserede logningen til et mere detaljeret niveau.

En anden væsentlig forskel er at DuplicateInvocationReporter er fokuseret imod identiske kald, hvilket vil sige at den ikke har fokus på svartid. Det betyder at man kan ”nøjes” med at kigge på identiske kald, frem for at skulle uddrage det mønster fra en stor mængde af kald. Fx vil kald der kun kommer én gang ikke bliver opsamlet og rapporteret.

Implementeringen i prototypen er dog ikke ”klar” til produktionsbrug, uden at man først er opmærksom på de krav, som den stiller. Den væsentligste udfordring er at loggeren i min prototype er stateful, hvilket vil sige at den vokser over tid. Reporteren vokser tilsvarende over tid, dog kun med referencer til entries i loggen. Jeg har valgt ikke at løse denne udfordring i min prototype, da jeg ikke anser det som et uløseligt problem. En simpel løsning kunne være en begrænsning på størrelse eller tid i loggen.

Implementeringen af invocationString i InvocationLogEntry er bygget ved, at lave en String af argumenterne og returværdien for et metodekald. Det er primært gjort for at have ”læsevenlige” invocationStrings, men er ikke ideelt. Dog virker equals på sådanne String objekter fint i en situation, hvor der anvendes simple datatyper og objekter uden overskrivning af toString. Fx ville et input af typen Kunde, med en toString metode, der kun udskrev navnet på kunden, introducere en mulighed for fejl. Det ville ske, hvis man fx havde to kunder ved navn ”Hans Andersen”, med forskellige CPR-numre. Et fix, som ville kunne fjerne denne kobling til toString, er at anvende hashCode for alle argumenter i invocationString.

Alt andet lige, stiller implementeringen krav til at equals og hashCode er implementeret korrekt, hvilket dog ikke bør være et problem. Grunden til dette er at de begge er en forudsætning for at bruge Java's collection API optimalt. [27]. Fx ville java.util.HashMap være ret ineffektiv uden en korrekt hashCode implementering. Desuden findes der i moderne IDE'er som fx Eclipse og IntelliJ hurtig autogenerering af begge.

Til trods for de nævnte implementeringsdetaljer, er det min opfattelse at prototypen giver et godt grundlag for at lave bedre analyser, end det er muligt ad hoc. Bedre både i forhold til større præcision, men også i forhold til den begrænsede tid det ville tage at implementere. Implementeringstiden vil alt andet lige være begrænset til én gang for rapporteringsdelen, og én gang per applikation for at indsætte opsamlingen, via en dynamic proxy i en DaoFactory.

5 Implementering af caching

I dette afsnit vil jeg først undersøge, hvad caching generelt er. Jeg vil starte med generelle begreber og efterfølgende undersøge de konkrete muligheder i en JEE kontekst. Inden jeg til sidst beskriver en prototype til en central caching mekanisme, vil jeg beskrive relevante caching strategier i forhold til de konkrete mønstre, som jeg ønsker at understøtte.

5.1 Undersøgelse af caching generelt

De fleste beskrivelser af caching stater med en definition af, hvad "caching" egentlig er. På Wikipedia[6] defineres en cache fx som: "en komponent, der forbedrer performance ved transparent, at gemme data så fremtidige forespørgsler på disse data kan besvares hurtigere". Andre steder [8] defineres det fx som: "Caching er det at holde andres data, så du ikke behøver at spørge efter det senere". Grunden til at der ikke findes en "præcis" definition, skal nok findes i det faktum, at "caching" som begreb, er brugt i rigtige mange forskellige sammenhænge alene inden for IT. I den ene ende finder man fx low-level caching i en klassisk "von neumann arkitektur"[9], hvor CPU'en bruger RAM som cache. I den anden ende finder man fx caching af HTTP requests i en browser via HTTP headers[10]. En interessant fællesnævner, for de mange forskellige definitioner af caching, er ordets oprindelse i det franske sprog, hvis betydning er "et gemmested" eller "at skjule".

Definition

Efter min overbevisning passer definitionen fra Wikipedia bedst til den kontekst, jeg befinder mig i. Det er en kontekst af caching i forbindelse med applikationer, som er skrevet i et objektorienteret sprog som Java og afviklet i en typisk distribueret arkitektur. Altså en webbaseret applikationsarkitektur, hvor klienten er en browser, applikationen findes på en webserver og data findes på en databaseserver eller et andet eksternt sted. Det centrale er altså, at der er tale om:

> "*en **komponent**, der forbedrer **performance** ved **transparent**, at **gemme data** så fremtidige forespørgsler på disse data kan besvares hurtigere*"

Hit/Miss

Konceptuelt gemmer man data via et "id", som også anvendes til at finde data igen. Når man spørger en cache om data med et id, er der to mulige udfald. Det første udfald er, at der ikke findes et resultat, hvilket kaldes et "**cache miss**". Det andet udfald er, at man finder data, hvilket kaldes et "**cache hit**". Disse to begreber bruges til at udregne henholdsvis "**hit ratio"** og **"miss ratio"**, altså forholdet imellem det totale antal forespørgsler og antallet af hits eller misses. Disse tal er interessante, hvis man ønsker at undersøge, hvor effektiv en cache er.

Caching strategier

Den helt afgørende faktor i forhold til hit/miss ratio'erne for en cache, er dens strategi i forhold til udskiftning af indhold. Fx ville data, der aldrig blev udskiftet, have en hit ratio gående imod 100%. Et element, der udskiftes inden, det blev læst anden gang, ville have en miss ratio på 100%. Teoretisk set er den optimale algoritme den, hvor det data, der bliver behov for længst ude i fremtiden, udskiftes først. Denne algoritme kaldes den clairvoyante algoritme eller Belady's algoritme[6].

I praksis findes der en række generelle algoritmer og en hel del variationer heraf. Jeg vil ikke nævne alle, men forsøge at skitsere de væsentligste, der danner grundlag for de fleste variationer.

FIFO

Den mest simple er FIFO - ”first in, first out”, hvilket vil sige, at den ældste ryger først ud. Dette implementeres typisk med en ”linked list” med fast størrelse, hvor der tilføjes nye elementer i den ene ende, mens de ældste simpelthen får lov at ”ryge” ud i den anden ende. Algoritmen er billig, da man hverken har behov for at holde styr på alder, antal læsninger eller behov for omrokering ved indsættelse. FIFO er dog generelt ikke anvendt hyppigt i dens simple form, da den ikke er særlig effektiv, hvis der fx er forskel på, hvor tit de forskellige elementer læses. Hvis der fx kun var plads til 10 elementer, og et af disse elementer blev læst meget hyppigere end de øvrige, ville den ryge ud af cachen, selvom den måske skulle bruges før de øvrige. Der findes dog variationer, som fx ”Second Chance” og ”Clock”[15], der forsøger at rette op på dette.

LRU

Hvis man derimod begynder at holde styr på, hvornår et element sidst blev læst, bevæger man sig over i en kategori af algoritmer, der går under navnet LRU - ”Least recently used”. I sin helt rå form smider den de elementer væk, der blev tilgået for længst tid siden. Det implementeres typisk via en ”linked list”, hvor man ved hver læsning flytter elementet om ”foran” i listen. På den måde skubbes de elementer, der aldrig læses ud i enden, mens de der læses genplaceres. LRU findes i en række varianter, og må nok siges, at være den hyppigst anvendte type indenfor caching i webapplikationer.

LFU

I stedet for at hold styr på hvornår et element blev læst, kan man holde styr på, hvor tit det blev læst. Denne type kaldes LFU - ”Least frequently used”, hvilket betyder, at man smider det element væk, der er læst færrest gange. Man tæller altså op for hver læsning af et element. Generelt er LFU ikke specielt effektiv ved fx skiftende forhold, hvor visse elementer læses mere i bestemte perioder. Dette kan fx være tilfældet, hvis der er en ”bruger” i den anden ende.

Den ”rigtige” algoritme

Udover de nævnte FIFO, LRU og LFU findes der et hav af kombinationer, der er mere eller mindre avancerede. Fælles for alle algoritmer er det dog, at den ”optimale” ikke findes i en kompleks verden. Dette betyder at man kun ved hjælp af statistik og efterfølgende tilpasning kan komme tæt på. Det er naturligvis muligt at lave en optimal caching, hvis man arbejder med ”små” mængder data, som sjældent opdateres og kan caches permanent. Et eksempel kunne være informationskurser på valuta fra nationalbanken, som kun opdateres én gang dagligt. De ville med fordel kunne hentes ind i en cache på forhånd, og efterfølgende give en optimal hit ratio. Den slags caching kaldes også for ”primed” i modsætning til det mere normale ”on demand”, hvor man indsætter i cachen ved første miss. [14]

Avancerede forhold

Udover de mere håndgribelige emner som hit/miss ratio og caching strategier, findes der også avancerede forhold, som jeg blot vil nævne for helhedens skyld. Den første problemstilling er udfordringen, der opstår, når en cache bliver for stor til at være en ren ”memory” baseret cache. I en Java verden vil det være når, der ikke længere er nok heap(hukommelse). I en sådan situation har man brug for en strategi, for at gemme hele eller dele af cachen på en harddisk eller et andet medie, hvilket ikke er en triviel problemstilling. En anden udfordring kan være at dele en cache på tværs af flere servere. Her skal der fx findes en strategi for propagerring af invalideringer fra en server til de

øvrige. I begge tilfælde bliver det alt andet lige mere komplekst, at gennemskue den reelle omkostning i forhold til den gevinst, man opnår ved caching. I nogle tilfælde kan fordelen måske helt forsvinde eller bliver til en reel omkostning.

Caching i Java

I et objektorienteret sprog som Java er det naturligt, at caching har fokus på objekter, hvilket betyder løsninger, der kan cache hele objekter. Som det tit er tilfældet med generelle problemstillinger i Java giver det anledning til en JSR – Java Specification Request. Disse ”standardiseringsønsker” behandles igennem JCP – Java Community Process, som i grove træk er en struktureret process, der har til formå at sikre solide specifikationer med bred opbakning og interesse i industrien. Helt naturligt blev der også lavet en JSR til objekt caching i Java. Den kom til at hedde JSR 107: JCACHE - Java Temporary Caching API. Desværre var opbakningen ikke stor nok, hvilket kan skyldes en række faktorer. Et svar på et blogindlæg[11] hentyder, at der simpelthen ikke var stor nok interesse i at arbejde ”gratis” for Oracle, som kom med det oprindelige oplæg til specifikationen. Det betyder at JSR 107 i dag har status inaktiv. Dog finders der en række forskellige implementeringer, som mere eller mindre følger det oprindelige API udkast[8].

Umiddelbart er en af de mest udbredte løsninger Ehcache[12], som samtidig også virker til at være en af de absolut mest modnede løsning, der understøtter langt flest avancerede features. Det vil sige mere end ”blot” en midlertidig cache i hukommelsen. Af andre løsninger, der også supportere JSR 107 delvist, kan Apache JCS, Cache4J og Google AppEngine nævnes. Kigger man lidt længere væk fra standarderne, finder man også object cache løsninger fra nogle af de store applikationsserver udbydere. IBM tilbyder DynaCache[17] på deres WebSphere platform og mens JBoss tilbyder Infinispan[22].

5.2 Caching strategier i forhold til mønstre

I dette afsnit vil jeg forsøge at holde de mest interessante mønstre op imod de mest interessante caching strategier. I afsnit 4.1.1 fandt jeg frem til, at de følgende to mønstre er de mest interessante for den type af applikationer, jeg beskæftiger mig med:

- **Flere kald til samme funktion på ”samme” tid**
- **Flere kald til samme funktion for samme bruger indenfor samme ”session”**

En væsentlig detalje er at levetiden for de data, der hentes typisk er væsentlig længere end en typiske ”session”. Det skyldes det faktum, at rigtig mange data i en netbank opdateres i forbindelse med natlige kørsler. Eksempler kunne være rentetilskrivning, indbetalinger, hævninger osv. Det kunne lede tanken hen på at, den ideelle opsætning ville være en **primed cache**. Altså, hvor der fx læses ind i cachen en gang om dagen umiddelbart efter de natlige kørsler. Det ville betyder, at der aldrig ville være et miss i cachen. Desværre ville det også betyde at langt størstedelen af cachen aldrig ville blive brugt – da langt fra alle kunder logger ind dagligt. Faktisk ville det nok også være umuligt at finde nok hukommelse eller diskplads til at gemme alle mulige funktionskald for samtlige ca. 200.000 netbankkunder.

Umiddelbart ville en løsning med FIFO kunne dække over behovet. For at være optimal ville det kræve en størrelse præcis stor nok til at rumme en entry per ”funktion” per session. På den måde

ville ”gamle” entries ryge ud først, når en session dør. Det kræver dog to ting – nemlig at sessioner har ”ens” levetid og at antallet af dem er forholdsvis ens. Desværre er sessioner afhængige af brugere, som desværre er forskellige. Det vil sige at antallet svinger og længden varierer. Selvfølgelig kan man opsamle statistik, der kan hjælpe, således at man kan komme tæt på. Spørgsmålet er bare om en ”stor” nok FIFO i praksis ville være for stor til at rumme alt.

LFU virker ikke specielt hensigtsmæssig i forhold til det faktum, at der er en sammenhæng imellem hvor meget en bruger fx klikker rundt i fx netbanken og antallet af læsninger i cachen. Det ville betyder, at en entry kunne bliver liggende rigtig lang tid, hvis en bruger havde været meget aktiv – faktisk nærmest permanent, indtil nok andre kunder ”overgik” antal klik. En kombination med en ”time-to-live” på entries ville kunne ”redde” situationen. På den måde ville en hyppigt læst entry forsvinde, hvis den blev inaktiv.

I forhold til LFU virker LRU mere oplagt, da den ville tage højde for inaktivitet som en del af dens virkemåde. Entries, der ikke bliver læst, vil simpelthen ryge ud når størrelsen er brugt. Umiddelbart virker den også mest oplagt til de to mønstre, hvor den samme funktion kaldes flere gange på kort tid. Det afhænger dog alt sammen af størrelsen på cachen.

I princippet burde det være ligegyldigt hvilken strategi, der anvendes, hvis cachen er stor til at rumme alle identiske kald for alle brugere indenfor en sessions levetid. Om det er praktisk muligt afhænger af størrelsen på de objekter, der caches og antallet af sessioner. Jeg vil i forbindelse med test af min prototype undersøge de tre forskellige strategier nærmere.

5.3 Prototype til central caching

I dette afsnit vil jeg beskrive den prototype til central caching, som jeg har implementeret. Formålet med prototypen er at indsætte en central caching, der kan logge alle metodekald til en DAO. Kildekoden kan ses i Appendiks D, hvor der endvidere findes et link til et komplet Eclipse projekt, der udover kildekoden indeholder anvendte frameworks.

Jeg har valgt benytte et caching framework til kernefunktionaliteten af en object cache. Det har jeg gjort af tre grunde. For det første er mit primære problem, hvor i arkitekturen jeg kan indsætte en cache, og hvad der skal caches for, at det bliver effektivt. For det andet ville det begrænse, den mængde af strategier, jeg ville kunne afprøve, hvis jeg selv skulle implementere dem fra bunden. For det tredje var det en oplagt mulighed for at få afprøvet et af de forskellige Java caching frameworks på markedet.

Mit valg af caching framework faldt på Ehcache[12], med den begrundelse at det virker som det bedste bud, hvis man vil have en moden løsning, som ikke er bundet til en bestemt JEE applikationsserver. Umiddelbart er det også den løsning, der får den bedste kritik i forhold til funktionalitet og performance på de blogindlæg og websites, jeg har læst om caching[13-15].

Learning tests af Ehcache

For at komme hurtigt og effektivt i gang med at anvende Ehcache byggede jeg nogle simple "learning tests"[26]. Det vil sige tests, der har til formål at afprøve, hvordan frameworket virker i praksis. Konkret fik jeg afprøvet forskellige måder at konfigurere cachen på, forskellige måder at instantiere cachen på og forskellige udgaver af cache interfaces. Fx fik jeg også testet den JSR107 cache implementering, som Ehcache tilbyder. Jeg endte fx ud med en test, der henter en cache fra en konfigurationsfil:

```
@Test
public void testCreateEhcacheViaManager() {
	CacheManager.create();
	CacheManager manager = CacheManager.getInstance();
	System.out.println(Arrays.toString(manager.getCacheNames()));;

	Ehcache cache = manager.getCache("sampleCache2");
	assertNotNull(cache);
	cache.put(new Element("key", "value"));
	assertEquals("Should be value1 that we get with
	key","value",cache.get("key").getValue());
}
```

En anden test afprøvede at lave og konfigurere cachen programmatisk:

```
@Test
public void testCreateFIFOCacheProgramatically() throws
                                                  InterruptedException {
	//create and add to manager
	CacheManager singletonManager = CacheManager.create();
	Cache testCache = new Cache(new CacheConfiguration("testCache",
	4).overflowToDisk(false).memoryStoreEvictionPolicy
	(MemoryStoreEvictionPolicy.FIFO));

	singletonManager.addCache(testCache);
	//get from manager and try to put
	//Cache testCache = singletonManager.getCache("testCache");
	testCache.put(new Element("key1", "value1"));
	Thread.sleep(100);
	testCache.put(new Element("key2", "value2"));
	testCache.put(new Element("key3", "value3"));
	testCache.put(new Element("key4", "value4"));
	testCache.put(new Element("key5", "value5"));
	System.out.println(testCache.getKeys());
	assertNull("Should not be value1 that we get with key1 - FIFO should
	have kicked it out",testCache.get("key1"));
}
```

Faktisk lærte denne test mig en del om Ehcache. Min FIFO test fejlede nemlig, selvom den umiddelbart ikke burde. Størrelse er sat til 4, og jeg putter 5 nye elementer i cachen. Efterfølgende tester jeg, at det første element ikke findes i cachen. Men det gjorde det altså! Faktisk røg det element med "key2" ud først. Ved at kigge lidt på kildekoden til Ehcache, blev jeg klar over, at der holdes styr på tid på elementer. Det fik mig til at eksperimentere med Thread.sleep, hvilket gav bonus med det samme. Ved at vente 100 millisekunder imellem første og andet element, fik jeg den ønskede opførsel – det første element røg ud først, som det burde med FIFO. Jeg har ikke gravet videre i årsagen, men jeg mistænker brugen af millisekunder i implementeringen, hvilket umiddelbart ikke er præcis nok. I en situation hvor der er "delay" ved fx IO, ville det næppe blive et reelt problem. Jeg laver trods alt 5 put kald på så tæt på samme tid, som muligt i en test uden flere tråde.

Implementering af cache

Et andet godt princip, når man tager et framework i brug, er at indkapsle det og dermed afskærme resten af applikationen[26]. Det har jeg gjort ved at lave klassen InvocationCache, som kun udstiller to metoder:

```
public void put(Invocation invocation, Object result)
public Object get(Invocation invocation)
```

Internt bruges en instans af net.sf.ehcache.Cache. Den skabes programmatisk og har en fast defineret størrelse samt cachingstrategi. Jeg har valgt ikke at ligge konfigurationen i en fil, da jeg ønskede at holde prototypen så simpel som mulig – og hurtig at ændre i. Endvidere har min "learning test", vist at det ikke er nogen stor opgave. Hele implementeringen er forholdsvis enkel og kan ses i Appendiks D. Fx ser get således ud:

```
public Object get(Invocation invocation) {
      Element element = invocationCache.get(invocation);
      return element != null ? element.getValue() : null;
}
```

Ovenstående er et godt eksempel på hvordan resten af applikationen afkobles for kendskabet til Element, som er en Ehcache specifik klasse – det er Object der returneres og ikke Element.

Klassen Invocation er en simpel indkapsling af den metode, der blev kaldt, samt de argumenter den blev kaldt med. Det vil sige det java.lang.reflect.Method objekt og det Object array, som jeg har modtaget i min DaoInvocationHandler:

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
                                                                  Throwable {
      Invocation invocation = new Invocation(method, args);
      Object ret = cache.get(invocation);
      if(ret == null) {
            ret = invokeAndLogInvocation(method, args);
        cache.put(invocation, ret);
        return ret;
      } else {
            return ret;
      }
}
```

Jeg har udnyttet min dynamic proxy, til at indskyde caching central. Det vil sige, at jeg tjekker cachen først, og returnerer resultatet derfra hvis det findes. Er cachen tom, foretager den det oprindelige kald, samtidig med at der stadig logges.

Test af cache med CallSimulator

Udover de Junit tests, som jeg har skrevet i forbindelse med udviklingen af prototypen, har jeg også foretaget en test af forskellige strategier. Til det formål har jeg anvendt min CallSimulator, som jeg beskrev i afsnit 4.2. Det vil sige, at jeg har simuleret i alt 360 backendkald, hvoraf 206 var unikke. Konkret har jeg skiftet følgende to indstillinger i min InvocationCache:

```
private static final MemoryStoreEvictionPolicy EVICTION_POLICY =
                  MemoryStoreEvictionPolicy.LRU;
private static final int CACHE_SIZE = 50;
```

Jeg har anvendt tre forskellige strategier: LIFO, LRU og LFU. Resultaterne er opsummeret i Tabel 3. Resultaterne viser meget godt, at alle tre strategier er lige effektive, hvis bare de får en stor nok størrelse.

Entries(206 unikke, 360 total)

Size	LIFO	LRU	LFU
1	330	330	330
50	252	249	256
100	220	218	228
181	208	208	208
182	206	208	206
188	206	206	206

Tabel 3: Opsummering af test med CallSimulator

En anden interessant observation, er at gevinsten fra 1-50 i størrelse, er væsentlig større end fra 50-100. Fx giver de første 49 en reduktion på 81 kald med LRU, mens de næste 50 ”kun” giver 31. Når resultatet kommer tættere på det optimale på 206, skal der en større og større cache til at gøre en forskel.

Umiddelbart virker det til, at LRU er en anelse bedre end de to øvrige. Ikke overraskende er LFU mindst effektiv, da antal læsninger ikke burde være afgørende for, om der kommer flere i det givne mønster.

I øvrigt er det væsentligt at bemærke, at resultaterne ikke er statiske. Det vil sige at der forekommer små variationer fra kørsel til kørsel. Dette skyldes givet den måde, hvorpå jeg afvikler min test uden forsinkelser. Billedet ville muligvis være anderledes, hvis jeg havde implementeret en realistisk forsinkelse i min CallSimulator imellem hvert kald. Variationerne var dog kun ca. et par kald +/-, men det samme overordnede mønster som tabellen viser.

5.4 Opsamling og evaluering af caching prototype

I afsnit 5.1 har jeg beskrevet de vigtigste begreber i forbindelse med caching, samt givet mit bud på den mest dækkende definition af caching:

> ”*en **komponent**, der forbedrer **performance** ved **transparent**, at **gemme data** så fremtidige forespørgsler på disse data kan besvares hurtigere”*

Efterfølgende har jeg forsøgt at kortlægge situationen omkring object caching i en Java verden, hvor standardiseringen igennem JSR 107 ikke fik fodfæste.

I afsnit 5.2 har jeg forsøgt at give et bud på, hvordan de forskellige caching strategier FIFO, LFU og LRU vil passe til de identificerede mønstre. Umiddelbart virker LRU eller FIFO, som de bedste kandidater til den type gentagne kald.

Evaluering af prototype – Maintainability

Ved at anvende en dynamic proxy til indsættelse af en central caching, har jeg opnået de samme fordele og ulemper, som beskrevet i slutningen af afsnit 3.1.2 Logging via dynamic proxy. Det betyder altså, at arkitekturen sikre god maintainability i forbindelse med fx implementering i eksisterende applikationer. Den primære grund til, at dynamic proxy sikre en god maintainability, er dens evne til at afskærme. Hverken klienten, der kalder en DAO, eller selve DAO implementeringen kender til dens eksistens.

Et alternativ til dynamic proxy kunne have været en løsning baseret på Aspect-oriented Programming – AOP. Her kunne man have isoleret kendskabet til caching i konfiguration af aspecter, hvilket helt ville fjerne kendskabet til caching fra den egentlige applikation. Jeg har dog bevist fravalgt AOP, da det i min arkitektur kun ville påvirke ét sted. Det drejer sig om det sted, hvor jeg indsætter mig proxy, hvilket sker et centralt sted i min DaoFactory. Min vurdering var derfor, at genvinsten ville være minimal. Især set i forhold til den ekstra tid, det ville tage at lære og forstå resten af kompleksiteten omkring et AOP framework som fx AspectJ.

En anden central maintainability kvalitet ved prototypen er indkapslingen af Ehcache, som er det framework, jeg har valgt at bruge. Ved at afskærme resten af applikationen fra selve Ehcache, vil en evt. udskiftning af det bagvedliggende caching framework have minimal påvirkning på resten. Det kræver selvfølgelig en ekstra klasse, der samtidig bliver den eneste klasse, der ville skulle skrives om ved en udskiftning.

Evaluering af prototype – performance

I forhold til performance har jeg ikke gennemført en egentlig grundig test af selve cachingdelen. Dog har jeg målt på eksekveringstiderne i forbindelse med afviklingen af min CallSimulator. I den forbindelse har jeg konstateret, at der er en pris ved initialisering af cachen. Umiddelbart koster det ca. 1.2-1.3 sekunder første gang en cache laves og anvendes. Derimod tager det kun 32 millisekunder at køre CallSimulator med 360 gets og ca 250 puts (en cache på 50). Det må siges, at være et minimalt overhead i forhold til, at et ”normalt” backendkald i min verden tager ca 50 millisekunder.

Min vurdering af performance i prototypen, er at den er acceptable, da en initialiseringspris er acceptabel, da den med stor sandsynlighed ville kunne tages ved opstart af applikationen. Hvis prisen betales ved opstart, vil det ikke få nogen indflydelse på svartiden for brugeren. Mht. til performance på put og get virker observationerne fra prototypen acceptabel. Situationen kan naturligvis ændre sig ved større datamængder og flere requests. I en sådan situation er det dog min vurdering af netop indkapslingen af caching frameworket, gør det muligt at adressere performance centralt uden påvirkning af resten af applikationen.

Evaluering af prototype – effektivitet i forhold til identiske kald

I forhold til caching af identiske metodekald, der enten kommer umiddelbart efter hinanden eller kort tid efter i samme session, viste prototypen sig at være ganske effektiv. En relativ lille cache på 50 i størrelse, gav en reduktion på 104-111 kald varierende imellem de tre testede strategier (som illustreret i tabel 3: Opsummering af test med CallSimulator).

Resultaterne er efter min overbevisning pæne, men er selvfølgelig skabt under de forudsætninger, der er gjort for prototypen. Det ville naturligvis have være optimalt med en afprøvning på en rigtig applikation, men det har jeg som nævnt i afsnit 1.3.3 valgt at afgrænse pga. praktiske og tidsmæssige udfordringer. Umiddelbart vurderer jeg dog, at mønsteret afprøvet med min CallSimulator, giver en god indikation af, at det ville være en investering værd.

En vigtig begrænsning i prototypen er, at kunne ”filtrere” skrivninger – det giver umiddelbart kun mening at cache læsninger. Jeg har ikke prioriteret at løse denne udfordring, men har dog overvejet et par løsningsmuligheder. En måde kunne være en navnestandard for metoder, hvor man automatisk frasorterer alle metoder, der starter med ”update”, ”insert” osv. En anden kunne være at frasortere alle metoder med returtypen void, som per definition ikke giver mulighed for caching af resultatet.

6 Konklusion

Målet med opgaven var to-delt, nemlig dels at lave caching på en mere genbrugelig og mere automatisk måde end ad-hoc løsninger. I forhold til genbrugelighed, er der for mig ingen tvivl om, at brugen af en dynamic proxy til at indsætte både logging og caching, er et godt match. Dels er der tale om en solid indkapsling, der sikre god maintainability, og dels er performanceprisen ved Java's reflection minimal. Indkapslingen er især vigtig, da det sikre minimal påvirkning af både nye og eksisterende applikationer.

På analysedelen, er løsningen i prototypen på ingen måde dækkende i forhold til en ad-hoc analyse. Det skyldes primært, at der i prototypen udelukkende er fokuseret på at finde to af flere mønstre. Til gengæld er DuplicateInvocationReporter i prototypen både mere genbrugelig og fokuseret til at finde gentagne backendkald end en tilsvarende ad-hoc analyse, som ville kræve en del manuel opsamling og bearbejdning. Til trods for at prototypen ikke er modnet til produktionsbrug, da det ville kræve bedre håndtering af fx hukommelsesforbrug, mener jeg, at prototypen er et godt udgangspunkt for at implementere flere typer af analyser. Det kunne fx være opsamling af svartider eller lign.

I forhold til at gøre løsningen automatisk, kan man sige at både logging, analyse og caching sker automatisk. Logging fordi alle backendkald simpelthen som udgangspunkt logges. I analysedelen er den eneste automatiske del DuplicateInvocationReporter. I forhold til den centrale caching, er automatikken igen på et simpelt niveau, da alt som udgangspunkt caches. Hvis jeg skulle have være nået helt i dybden med automatikken, kunne det optimale være at gøre nogle forsøg med automatisk tilpasning af cachingstrategierne. Fx skift imellem FIFO og LRU baseret på statistik i cachen. Det har jeg ikke haft tid i denne opgave. Muligvis har jeg heller ikke en god nok simulation til, at det ville give mening. Alt andet lige ville den slags kræve flere eksperimenter for at samle et bedre datagrundlag.

Med det udgangspunkt at målet var at gøre det bedre end ad-hoc situationen, mener jeg, at prototyperne i opgaven giver nogle gode bud på solide løsninger, der ville kunne opnå det mål i en egentlig implementering. Endvidere er der også skabt et fundament for at løse andre mønstre end de to omkring identiske kald.

En oplagt udvidelsesmulighed ville være at finde ”næsten” identiske kald. Det kunne fx være en situation, hvor et forkert design har gjort, at man returnerer et domæne objekt bestående af to dele, der opdateres uafhængigt af hinanden. Et eksempel kunne være et konto objekt, der både indeholder saldo, kontonummer, kontonavn og rentesats. I sådan en situation, ville 75% værdierne være ens i langt de fleste kald, da kun saldo ville bliver opdateret hyppigt. Eksemplet er simpelt, men i en verden styret af store komplekse datamodeller, er det bestemt ikke unormalt med store objekter, der sendes rundt i systemerne uden hensyntagen til den slags ”spild”. Alt andet lige ville en sådan analyse være svær at gennemføre ved ad-hoc logging og analyse. Det ville til gengæld ikke være en alt for stor opgave, at udvide prototyperne i denne opgave til at omfatte en sådan rapportering.

7 Referencer

[1] Deepak Alur, John Crupi and Dan Malks. Core J2EE Patters: Best Practices and Design Strategies Second Edition 2003, side 462-495.

[2] Jakob Eyvind Bardram, Henrik Bærbak Christensen and Klaus Marius Hansen. Architectural Prototyping: An Approach for Grounding Architectural Design and Learning 2004

[3] Brian Goetz. Java theory and practice: Decorating with dynamic proxies 2005 (http://www.ibm.com/developerworks/java/library/j-jtp08305.html)

[4] Martin Fowler, Patternes of Enterprise Application Architecture 2003, side 6-9

[5] Ted Neward, Effective Enterprise Java 2005, Item 8 side 59-67

[6] Cache og Cache strategier på Wikipedia.org, http://en.wikipedia.org/wiki/Cache og http://en.wikipedia.org/wiki/Cache_algorithms

[7] JSR 107 home, http://jcp.org/en/jsr/detail?id=107, oprindeligt bidrag fra Oracle: http://jcp.org/aboutJava/communityprocess/jsr/cacheFS.pdf

[8] JSR 107 Java Caching API, https://jsr-107-interest.dev.java.net/

[9] Von Neumann arkitektur, http://en.wikipedia.org/wiki/Von_Neumann_architecture

[10] Web Cache, http://en.wikipedia.org/wiki/Web_cache

[11] Sylvian Wallez, blogindlæg om JSR 107 september 2009, http://bluxte.net/musings/2009/09/10/what-happened-jcache-aka-jsr-107

[12] Ehcache, http://ehcache.org/

[13] Srini Penchikala, J2EE object-caching frameworks, Improve performance in Web portal applications by caching objects By, JavaWorld.com, april 2005 http://www.javaworld.com/javaworld/jw-05-2004/jw-0531-cache.html

[14] Abhijit Gadkari, Caching in the Distributed Environment, The Architecture Journal MSDN oktober 2008, http://msdn.microsoft.com/en-us/library/dd129907.aspx

[15] Ahmed Ali, blogindlæg februar 2009 http://javalandscape.blogspot.com/2009/01/cachingcaching-algorithms-and-caching.html

[16] Java Caching System, http://jakarta.apache.org/jcs/index.html

[17] Mastering DynaCache in WebSphere Commerce – IBM Redbook http://www.redbooks.ibm.com/abstracts/sg247393.html

[18] Jonathan Davies et. al., An Aspect Oriented Performance Analysis Environment

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.8114&rep=rep1&type=pdf

[19] Sara Bouchenak et. al., Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution http://sardes.inrialpes.fr/~bouchena/publications/Middleware06.pdf

[20] Selenium web testing framework, http://seleniumhq.org/

[21] Log4J logging framework, http://logging.apache.org/log4j/

[22] InfiniSpan JBoss caching framework, http://www.jboss.org/infinispan

[23] TDD, http://en.wikipedia.org/wiki/Test-driven_development

[24] Joshua Bloch, Effictive Java Second Edition 2008, Item 1 og Item 15.
[25] jMock letvægts Java mocking framework, http://www.jmock.org/
[26] Robert C. Martin, Clean Code 2009, Chapter 8: Boundaries
[27] Joshua Bloch, Effictive Java Second Edition 2008, Item 8-10.
[28] Alan Shalloway, Design Patterns Explained 2002, kapitel 17.

8 Appendiks A – Kildekode til logging i DAO

I dette appendiks findes alt kildekode til prototypen i afsnit 3.1.1 Logging i DAO. Test klasserne vises først, da de danner grundlag for selve implementeringen.

```
package caching.dao;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import java.util.List;
import org.junit.Test;
import caching.logging.SimpleLogger;
public class TestBackendDao {
	private BackendDao dao;
	private SimpleLogger logger;
	public TestBackendDao() {
		logger = SimpleLogger.getInstance();
		dao = DaoFactory.getInstance(logger).getBackendDao();
	}
	@Test
	public void canFindAListOfBackendObjectsByStringId() {
		List<BackendObject> backendObjects = dao.findObjectsById("someid");
		assertNotNull("List of backendobjects null", backendObjects);
	}
	@Test
	public void canFindABackendObjectByStringId() {
		BackendObject backendObject = dao.findObjectById("someid");
		assertNotNull("BackendObject null", backendObject);
	}
	@Test
	public void canPersistABackendObject() {
		BackendObject backendObject = new BackendObject();
		dao.persistObject(backendObject);
		// should assert that you can get the persisted object via the find
		// methods - but not relevant for this prototype
	}
	@Test
```

```
	/** Output is simply logged via toString
	 */
	public void logsInputParamsAndOutputForIdQuery() {
		BackendObject backendObject = dao.findObjectById("anidtologasinputparam");
		long before = System.currentTimeMillis();
		dao.findObjectById("objectId");
		long after = System.currentTimeMillis();
		System.out.println("with dynamic logging: "+(after-before));
		assertTrue("Could not find logged input/output entry in logger",
		logger.contains("anidtologasinputparam" + backendObject));
	}
	@Test
	public void logsInputParamsForListQuery() {
		dao.findObjectsById("anotherid");
		assertTrue("Could not find logged id(anotherid) in logger",
	logger.contains("anotherid"));
	}
}

package caching.dao;
import java.util.List;
public interface BackendDao {
	public List<BackendObject> findObjectsById(String id);
	public BackendObject findObjectById(String id);
	public void persistObject(BackendObject input);
}

package caching.dao;
import java.util.ArrayList;
import java.util.List;
import caching.logging.SimpleLogger;
public class BackendDaoImpl implements BackendDao {
	private final SimpleLogger logger;
	public BackendDaoImpl(SimpleLogger logger) {
		this.logger = logger;
	}
```

```
	@Override
	public BackendObject findObjectById(String id) {
		BackendObject object = new BackendObject();
		logger.log(id + object);
		return object;
	}
	@Override
	public List<BackendObject> findObjectsById(String id) {
		logger.log(id);
		return new ArrayList<BackendObject>();
	}
	@Override
	public void persistObject(BackendObject input) {
	}
}

package caching.dao;
public class BackendObject {
}

package caching.dao;
import org.junit.Test;
import static org.junit.Assert.assertNotNull;
public class TestDaoFactory {
	@Test
	public void canCreateDao() {
		DaoFactory factory = DaoFactory.getInstance();
		BackendDao dao = factory.getBackendDao();
		assertNotNull(dao);
	}
}

package caching.dao;
import caching.logging.SimpleLogger;
public class DaoFactory {
```

```
	private static SimpleLogger logger;
	public static DaoFactory getInstance() {
		return new DaoFactory();
	}
	public BackendDao getBackendDao() {
		return new BackendDaoImpl(logger);
	}
	public static DaoFactory getInstance(SimpleLogger logger) {
		DaoFactory.logger = logger;
		return new DaoFactory();
	}
}
```

```
package caching.logging;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import org.junit.Test;
public class TestSimpleLogger {
	@Test
	public void canLogAString() {
		SimpleLogger logger = SimpleLogger.getInstance();
		logger.log("log entry");
		assertTrue("Logger doesn't contain new entry: log entry", logger.contains("log
		entry"));
	}
	@Test
	public void canCreateALogger() {
		SimpleLogger logger = SimpleLogger.getInstance();
		assertNotNull("Failed to get SimpleLogger instance - got null", logger);
	}
}
```

```
package caching.logging;
import java.util.ArrayList;
import java.util.List;
```

```
public class SimpleLogger {
	private List<String> entries = new ArrayList<String>();
	public static SimpleLogger getInstance() {
		return new SimpleLogger();
	}
	public void log(String string) {
		entries.add(string);
	}
	public boolean contains(String string) {
		return entries.contains(string);
	}
}
```

9 Appendiks B – Kildekode til logging via dynamic proxy

I dette appendiks findes alt kildekode til prototypen i afsnit 3.1.2 Logging via dynamic proxy. Test klasserne vises først, da de danner grundlag for selve implementeringen. Enkelte klasser er identiske med klasserne i appendiks A, hvorfor der henvises dertil når det er tilfældet.

```
package caching.dao;
import static org.junit.Assert.assertNotNull;
import java.util.List;
import org.junit.Test;
import caching.logging.SimpleLogger;
public class TestBackendDao {
	private BackendDao dao;
	private SimpleLogger logger;
	public TestBackendDao() {
		logger = SimpleLogger.getInstance();
		dao = DaoFactory.getInstance(logger).getBackendDao();
	}
	@Test
	public void canFindAListOfBackendObjectsByStringId() {
		List<BackendObject> backendObjects = dao.findObjectsByIds(new String[]
		{"someid"});
		assertNotNull("List of backendobjects null", backendObjects);
	}
```

```
	@Test
	public void canFindABackendObjectByStringId() {
		BackendObject backendObject = dao.findObjectById("someid");
		assertNotNull("BackendObject null", backendObject);
	}
	@Test
	public void canPersistABackendObject() {
		BackendObject backendObject = new BackendObject();
		dao.persistObject(backendObject);
		//should assert that you can get the persisted object via the find methods - but not relevant for this prototype
	}
}
```

BackendDao interface – se Appendiks A

```
package caching.dao;
import java.util.ArrayList;
import java.util.List;
public class BackendDaoImpl implements BackendDao {
	@Override
	public BackendObject findObjectById(String id) {
		BackendObject object = new BackendObject();
		return object;
	}
	@Override
	public List<BackendObject> findObjectsByIds(String[] ids) {
		return new ArrayList<BackendObject>();
	}
	@Override
	public void persistObject(BackendObject input) {
	}
}
```

BackendObject (tomt dummy object til test)– se Appendiks A

```
package caching.dao;
import java.lang.reflect.Proxy;
import org.junit.Test;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
public class TestDaoFactory {
	@Test
	public void canCreateDao() {
		DaoFactory factory = DaoFactory.getInstance();
		BackendDao dao = factory.getBackendDao();
		assertNotNull(dao);
	}
	@Test
	public void canCreateDynamicProxyDao() {
		DaoFactory factory = DaoFactory.getInstance();
		BackendDao dao = factory.getBackendDao();
		assertNotNull(dao);
		assertTrue("dao not instanceof Proxy", dao instanceof Proxy);
	}
}
```

```
package caching.dao;
import java.lang.reflect.Proxy;
import caching.logging.LoggingInvocationHandler;
import caching.logging.SimpleLogger;
public class DaoFactory {
	private static SimpleLogger logger;
	public static DaoFactory getInstance() {
		return new DaoFactory();
	}
	public BackendDao getBackendDao() {
		return getBackendDaoProxy(new BackendDaoImpl());
	}
	private BackendDao getBackendDaoProxy(final BackendDao dao) {
```

```
                return (BackendDao) Proxy.newProxyInstance( dao.getClass().getClassLoader(),
                new Class[] { BackendDao.class }, new
                LoggingInvocationHandler<BackendDao>(dao, logger));
        }
        public static DaoFactory getInstance(SimpleLogger logger) {
                DaoFactory.logger = logger;
                return new DaoFactory();
        }
}

package caching.logging;
import static org.junit.Assert.assertTrue;
import org.junit.Test;
import caching.dao.BackendDao;
import caching.dao.BackendObject;
import caching.dao.DaoFactory;
public class TestLoggingInvocationHandler {
        @Test
        public void canLogMethodCallOnUnderlyingObject() {
                SimpleLogger logger = new SimpleLogger();
                BackendDao dao = DaoFactory.getInstance(logger).getBackendDao();
                long before = System.currentTimeMillis();
                BackendObject object = dao.findObjectById("objectId");
                long after = System.currentTimeMillis();
                System.out.println("with dynamic logging: "+(after-before));
                assertTrue("Logger missing log statement",
                logger.contains("caching.dao.BackendDaoImpl.findObjectById: objectId ->
                "+object));
        }
}

package caching.logging;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class LoggingInvocationHandler<T> implements InvocationHandler {
```

```
	private final SimpleLogger logger;
	private final T underlying;
	public LoggingInvocationHandler(T underlying, SimpleLogger simpleLogger) {
		this.underlying = underlying;
		this.logger = simpleLogger;
	}
	@Override
	public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
		StringBuffer sb = new StringBuffer();
		sb.append(underlying.getClass().getName()+".");
		sb.append(method.getName()+ ": ");
		 for (int i=0; args != null && i<args.length; i++) {
			 if (i != 0) {
				sb.append(", ");
			}
			sb.append(args[i]);
		}
		 Object ret = method.invoke(underlying, args);
		 if (ret != null) {
			sb.append(" -> ");
			sb.append(ret);
		}
		System.out.println("About to add to log: "+sb.toString());
		logger.log(sb.toString());
		return ret;
	}
}
```

TestSimpleLogger og **SimpleLogger** – se Appendiks A

10 Appendiks C – Forklaring til statistiske nøgletal

De følgende forklaringer til forskellige statistiske nøgletal er baseret på observationer gjort i forbindelse med performance analyse i mit daglige arbejde, samt viden indhentet fra forskellige kilder som wikipedia.org og den indbyggede hjælp til statistiske funktioner i MS Excel. Der er på ingen måde lagt vægt på korrekte matematiske definitioner, da dette ikke vurderes relevant i kontekst af dette arbejde.

10.1 Middelværdi

Middelværdi er en af de mest almindeligt anvendte målinger når man ser på data. Beregningen er enkel: Sum divideret med antallet af observationer.

Det meste af tiden kan dette være en god indikation af, hvilket ”niveau” du kigger på. Men middelværdi kan ikke stå alene i vores særlige verden. Det er på grund af karakteren af vores data i deres rå form. Disse indeholder ofte ekstreme observationer, som har stor indflydelse den gennemsnitlige beregning. Forestil dig fx, at du har 95 observationer på 40-50 ms og 5 observationer tæt på 10000ms (som kan være en timeout grænse). Dette ville give en middelværdi på 542,75 (95 x 45 + 5 x 10.000) / 100. Denne middelværdi skjuler effektivt både "normale" svartider på 40-50 ms og problemer med timeout 5% af tiden. Derfor bør man som minimum kombinere middelværdien med en median.

10.2 Median

Medianen er værdien af observationen i midten af data. Og når man ser på tallene er det i de sorterede værdier. Lad os sige, du har disse observationer: [2, 6, 7, 4, 3]. Dette ville give en median på 4, fordi sorteret er det [2, 3, 4, 6, 7]. Hvis antallet af observationer er et lige bliver medianen beregnet som gennemsnittet af de to midterste observationer. For eksempel ville [2, 3, 4, 5] give en median på 3,5.

Hvis man ser på det eksempel, der er nævnt ved beskrivelsen af middelværdi, vil medianen være et tal mellem 40 og 50. Og når man sammenligner denne med middelværdien på 542,75 fremgår det, at noget er galt.

Derfor kan medianen i kombination med en middelværdi give dig en indikation af, hvordan "jævnt" fordelt dine observationer. Hvis de er omkring det samme, har du en god indikation af, hvor stor spredning der er. Hvis ikke du har brug for at kigge på nogle af de andre målinger for at se hvordan "dårlige" din fordeling er, og hvad det skyldes.

10.3 Standardafvigelse

Standardafvigelsen er en relativ værdi, der beskriver hvor stor er variationen i observationer er i forhold til middelværdi. For eksempel (oversat wiki), er den gennemsnitlige højde for voksne mænd i USA er omkring 178 cm, med en standardafvigelse på omkring 8 cm. Det betyder, at de fleste mænd (ca. 68 procent, under forudsætning af en normalfordeling) har en højde indenfor 8 cm af den gennemsnitlige højde (170-185 cm) og næsten alle mænd (ca. 95%) har en højde indenfor 15 cm af den gennemsnitlige (163-193 cm) – 2 standardafvigelser.

Vær forsigtig dog forsigtig med forhastede konklusioner omkring standardafvigelse, for som dette eksempel forsøger at påpege er "de fleste" (68%) og "næsten alle" (95%) antagelser er baseret på en normalfordeling. I vores eksempel fra middelværdi og median er standardafvigelsen 2180,59 – hvilket siger, at "de fleste" er inden for 2180,59+542,75, og "næsten alle" er inden for 2x2180.59 +542,75. Det giver ikke megen mening, fordi i virkeligheden er næsten alle (95%) er under 50.

Når du bruger din standardafvigelse til at forklare noget om, hvad "de fleste" eller "næsten alle" observationer er, så sørg for også at se på skævhed - da dette vil hjælpe med at beskrive, hvordan din distribution er i forhold til din middelværdi. Som et alternativ kunne man skille sig af med de "unormale" observationer - for eksempel top 5% og bund 5%, og derefter behandle disse separat. I vores eksempel ville dette give et rimeligt resultat.

10.4 Skævhed

Skævhed er en relativ værdi der beskrivelser, hvor asymmetriske observationerne er fordelt rundt om en middelværdi. En positiv værdi angiver, at du har en fordeling "højere" end din middelværdi. En negativ værdi angiver en "lavere" observationer. Så jo større tallet er, jo større er din "hale". Når man ser på observationer med en høj skævhed er det bedst at kombinere analysen med et histogram der viser, hvilke intervaller observationerne findes i. Så se skævhed som en indikation af, hvor "normal" din distribution er.

10.5 Min/Maks., antal og periode

Disse målinger er ret ligetil, men værd at nævne, da der kan være vigtige aspekter til data "skjulte" her. En ting at se på, er antal i kombination med den periode, hvilket vil give dig en god indikation af, hvor ofte funktionen bruges. Ser man på dette og sammenligner med dine forventninger kan man finde tilfælde af spildte ressourcer og potentielle kandidater til mere caching.

Hvis du har data med ekstreme værdier, (fx 0 eller 10000) vil dette være synligt i dine min og maks. værdier, og du bør undersøge disse yderligere. Det kan også være meget nyttigt at se på din periode og tager månedsafslutning, kvartalsafslutning og årsskifte i betragtning, da disse kan have en stor indvirkning på dine svartider.

11 Appendiks D – Kildekode til log analyse og central caching

I dette appendiks findes alt kildekode til prototyperne i afsnit 4.2 og 5.3. Test klasserne vises først, da de danner grundlag for selve implementeringen. Som et alternativ kan kildekode inkl. diverse 3rd party jars hentes her: http://labanos.dk/files/prototypes.zip (zip-filen indeholder en export fra Eclipse, der med fordel kan åbnes via den tilsvarende import feature).

```
package caching.cache;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertNotNull;

import org.junit.Test;


/**Cannot test with Method being anything but null, as mocking a final bootstrapped java class

*is not possible with jmock. Will instead be tested indirectly when used with the InvocationHandler.

*/
```

```
public class TestInvocation {
	@Test
	public void canCreateInvocationFromMethodAndArgs() {
		Invocation invocation = new Invocation(null , new Object[] {});
		assertNotNull(invocation);
	}
	@Test
	public void knowsIfTwoInvocationsAreEqual() {
		Invocation invocationA = new Invocation(null , new Object[] {});
		Invocation invocationB = new Invocation(null , new Object[] {});
		assertEquals(invocationA, invocationB);
	}
	@Test
	public void knowsIfTwoInvocationsWithNullsAreEqual() {
		Invocation invocationA = new Invocation(null , null);
		Invocation invocationB = new Invocation(null , null);
		assertEquals(invocationA, invocationB);
	}
	@Test
	public void givesSameHashCodeForTwoEqualInvocations() {
		Invocation invocationA = new Invocation(null , new Object[] {});
		Invocation invocationB = new Invocation(null , new Object[] {});
		assertEquals(invocationA.hashCode(), invocationB.hashCode());
	}
	@Test
	public void knowsTheSameInvocationToBeEquals() {
		Invocation invocationA = new Invocation(null , new Object[] {});
		assertEquals(invocationA, invocationA);
	}
	@Test
	public void knowsANullInvocationIsNotEqualToAInvocation() {
		Invocation invocationA = new Invocation(null , new Object[] {});
		assertFalse(invocationA.equals(null));
	}
	@Test
```

```
	public void knowsAStringIsNotEqualToAInvocation() {
		Invocation invocationA = new Invocation(null , new Object[] {});
		assertFalse(invocationA.equals(""));
	}
	@Test
	public void knowsNotEqualsIfDifferentArgs() {
		Invocation invocationA = new Invocation(null , new Object[] {});
		Invocation invocationB = new Invocation(null , new Object[] {""});
		assertFalse(invocationA.equals(invocationB));
	}
}
package caching.cache;
import java.io.Serializable;
import java.lang.reflect.Method;
import java.util.Arrays;

public class Invocation implements Serializable {
	private static final long serialVersionUID = 59186452010707170006L;
	private final Method method;
	private final Object[] objects;
	public Invocation(Method method, Object[] objects) {
		this.method = method;
		this.objects = objects;
	}
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((method == null) ? 0 : method.hashCode());
		result = prime * result + Arrays.hashCode(objects);
		return result;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
```

```
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Invocation other = (Invocation) obj;
            if (method == null) {
                if (other.method != null)
                    return false;
            } else if (!method.equals(other.method))
                return false;
            if (!Arrays.equals(objects, other.objects))
                return false;
            return true;
        }
}
package caching.cache;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import org.junit.Test;

public class TestInvocationCache {
        @Test
        public void canCreateSingletonInvocationCache() {
            InvocationCache cache = InvocationCache.getInstance();
            assertNotNull("Cache shouldnt be null!", cache);
        }
        @Test
        public void canPutAndGetInvocationAndResultInCache() {
            InvocationCache cache = InvocationCache.getInstance();
            Invocation invocation = new Invocation(null , new Object[] {});
            Object result = "";
            cache.put(invocation, result);
            assertEquals("Should get the same result", result, cache.get(invocation));
        }
```

```
}
package caching.cache;
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;
import net.sf.ehcache.config.CacheConfiguration;
import net.sf.ehcache.store.MemoryStoreEvictionPolicy;

public class InvocationCache {
        private static final MemoryStoreEvictionPolicy EVICTION_POLICY =
                                                        MemoryStoreEvictionPolicy.LRU;
        private static final int CACHE_SIZE = 50;
        private static final InvocationCache instance = new InvocationCache();
        private Cache invocationCache;

        public static InvocationCache getInstance() {
                return instance;
        }
        private InvocationCache() {
                CacheManager singletonManager = CacheManager.create();
                invocationCache = new Cache(new CacheConfiguration("invocationCache",
                CACHE_SIZE).overflowToDisk(false).
                memoryStoreEvictionPolicy(EVICTION_POLICY));
                singletonManager.addCache(invocationCache);
        }
        public void put(Invocation invocation, Object result) {
                invocationCache.put(new Element(invocation, result));
        }
        public Object get(Invocation invocation) {
                Element element = invocationCache.get(invocation);
                return element != null ? element.getValue() : null;
        }
}
```

Learning tests af Ehcache:

```
package caching.cache;
import static org.junit.Assert.assertEquals;
```

```
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
import java.util.Arrays;
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Ehcache;
import net.sf.ehcache.Element;
import net.sf.ehcache.config.CacheConfiguration;
import net.sf.ehcache.store.MemoryStoreEvictionPolicy;
import org.junit.Test;

public class TestJCache {
	@Test
	public void testCreateEhcacheViaManager() {
		CacheManager.create();
		CacheManager manager = CacheManager.getInstance();
		System.out.println(Arrays.toString(manager.getCacheNames()));;
		Ehcache cache = manager.getCache("sampleCache2");
		assertNotNull(cache);
		cache.put(new Element("key", "value"));
		assertEquals("Should be value1 that we get with
				key","value",cache.get("key").getValue());
	}
	@Test
	public void testCreateFIFOCacheProgramatically() throws InterruptedException {
		//create and add to manager
		CacheManager singletonManager = CacheManager.create();
		Cache testCache = new Cache(new CacheConfiguration("testCache", 4).
		overflowToDisk(false).memoryStoreEvictionPolicy(
		MemoryStoreEvictionPolicy.FIFO));
		singletonManager.addCache(testCache);
		//get from manager and try to put
		//Cache testCache = singletonManager.getCache("testCache");
		testCache.put(new Element("key1", "value1"));
		Thread.sleep(100);
		testCache.put(new Element("key2", "value2"));
```

```
        testCache.put(new Element("key3", "value3"));
        testCache.put(new Element("key4", "value4"));
        testCache.put(new Element("key5", "value5"));
        System.out.println(testCache.getKeys());
        assertNull("Should not be value1 that we get with key1 - FIFO should have kicked it
        out",testCache.get("key1"));
    }
    @Test
    public void testCreateLFUCacheProgramatically() throws InterruptedException {
        //create and add to manager
        CacheManager singletonManager = CacheManager.create();
        Cache lfuCache = new Cache(new CacheConfiguration("testLFUCache", 3).
        overflowToDisk(false).memoryStoreEvictionPolicy(
        MemoryStoreEvictionPolicy.LFU));
        singletonManager.addCache(lfuCache);
        //get from manager and try to put
        Cache test = singletonManager.getCache("testLFUCache");
        test.put(new Element("key1", "value1"));
        test.put(new Element("key2", "value2"));
        test.put(new Element("key3", "value3"));
        test.put(new Element("key4", "value4"));
        test.put(new Element("key5", "value5"));
        System.out.println(test.getKeys());
        assertNull("Should not be value1 that we get with key1 - LFU should have kicked it
        out",test.get("key1"));
        test.get("key4");
        test.get("key4");
        test.get("key3");
        test.put(new Element("key6", "value6"));
        System.out.println(test.getKeys());
        assertNull("Should not be any value with key5 - LFU should have kicked it
        out",test.get("key5"));
    }
}
package caching.dao;
import static org.junit.Assert.assertEquals;
```

```
import java.io.File;
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.junit.Test;
import caching.logging.LogAnalyzer;

public class TestCallSimulator {
	private static final String LOGFILE_NAME =
	"C:/Users/Ejer/workspace/caching.prototypes_log_analyser/data/
	hentProduktoversigtLog.txt";
	private static final String FIRST_LINE_OF_LOG = "jyskeprodukter_da1643-
		cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:**	14-01-10	07:59:04,96
		hentProduktoversigt	63	8480";
	private static final String LAST_LINE_OF_LOG = "jyskeprodukter_da1643-
		cp1_jb.ind.ebk.produktovs.ear-V2-6-4.log:**	14-01-10	15:07:05,89
		hentProduktoversigt	110	2872";
	@Test
	public void canReadLineFromFile() throws Exception {
		CallSimulator simulator = new CallSimulator(new File(LOGFILE_NAME));
		assertEquals("readline did not give expected result", FIRST_LINE_OF_LOG,
		simulator.readLine());
	}
	@Test
	public void canReadAllLinesInFile() throws Exception {
		CallSimulator simulator = new CallSimulator(new File(LOGFILE_NAME));
		int numberOfLines = 0;
		while (simulator.readLine() != null) {
			numberOfLines++;
		}
		assertEquals("should be 360 lines in file",360,numberOfLines);
	}
	@Test
	public void canGetUserFromLine() throws Exception {
		CallSimulator simulator = new CallSimulator(new File(LOGFILE_NAME));
		assertEquals("first line should contain user 8480", "8480",
		simulator.getUserFromLine(FIRST_LINE_OF_LOG));
		assertEquals("last line should contain user 2872", "2872",
		simulator.getUserFromLine(LAST_LINE_OF_LOG));
```

```
	}
	@Test
	public void canGetTimeInMillisecondsFromLine() throws Exception {
		CallSimulator simulator = new CallSimulator(new File(LOGFILE_NAME));
		assertEquals("first line should contain time 1263452344096", 1263452344096L,
	simulator.getTimeInMillisFromLine(FIRST_LINE_OF_LOG));
	}
	@Test
	public void willCallDaoAsManyTimesAsLogHasEntriesWhenStarted() throws Exception {
		Mockery context = new Mockery();
		final CustomerDao dao = context.mock(CustomerDao.class);
		final CallSimulator simulator=new CallSimulator(new
File(LOGFILE_NAME),dao);
		// expectations
		context.checking(new Expectations() {{
			exactly(360).of (dao).findCustomerById(with(any(String.class)));
		}});
		// execute
		simulator.playLog();
		// verify
		context.assertIsSatisfied();
	}
	@Test
	public void reportsDuplicateInvocationsWhenCallSimulatorCalls() throws Exception {
		DaoFactory daoFactory = DaoFactory.getInstance();
		CallSimulator simulator = new CallSimulator(new File(LOGFILE_NAME)
		,daoFactory.getCustomerDao());
		simulator.playLog();
		LogAnalyzer analyzer = new LogAnalyzer(daoFactory.getLogger());
		int numberOfEntries = analyzer.reportNumberOfEntries();
		//removed now that we cache assertEquals("There should be 360 entries", 360,
		//numberOfEntries);
		System.out.println("Number of unique entries: "+analyzer.
	reportNumberOfUniqueEntries()+" out of total entries: "+numberOfEntries);
	}
}
package caching.dao;
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class CallSimulator {
	private BufferedReader input;
	private final CustomerDao dao;

	public CallSimulator(File file, CustomerDao dao) throws FileNotFoundException {
		this.dao = dao;
		input = new BufferedReader(new FileReader(file));
	}
	public CallSimulator(File file) throws FileNotFoundException {
		this(file, null);
	}
	String readLine() throws IOException {
		return input.readLine();
	}
	String getUserFromLine(String logLine) {
		return logLine.substring(logLine.lastIndexOf('	')).trim();
	}
	long getTimeInMillisFromLine(String firstLineOfLog) throws Exception {
		String dateString = firstLineOfLog.substring(firstLineOfLog.lastIndexOf('*')+1,
		firstLineOfLog.indexOf(',') + 4).trim();
		SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-
		yyHH:mm:ss,SSS");
		Date firstDate = formatter.parse(dateString);
		return firstDate.getTime();
	}
	public void playLog() throws Exception {
		String currentLine = "";
```

```
                while ((currentLine = readLine()) != null) {
                        dao.findCustomerById(getUserFromLine(currentLine));
                }
        }
}
package caching.dao;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNotSame;
import static org.junit.Assert.assertTrue;
import org.junit.Before;
import org.junit.Test;

public class TestCustomer {
        private Customer customerA;
        private Customer customerAa;
        private Customer customerB;
        private Customer customerC;
        @Before
        public void setUp() {
                customerA = new Customer("a");
                customerAa = new Customer("a");
                customerB = new Customer("b");
                customerC = new Customer(null);
        }
        @Test
        public void knowsIfACustomerIsEqualToAnotherCustomer() {
                assertTrue( customerA.equals(customerA));
                assertTrue( customerA.equals(customerAa));
                assertTrue( customerAa.equals(customerA));
                assertFalse( customerA.equals(customerB));
                assertFalse( customerB.equals(customerA));
                assertFalse( customerA.equals(null));
```

```
		assertFalse( customerA.equals(""));
		assertFalse( customerC.equals(customerA));
	}
	@Test
	public void knowsAProperHashCode() {
		assertNotNull(customerA.hashCode());
		assertEquals( customerA.hashCode(), customerAa.hashCode());
		assertNotSame( customerA.hashCode(), customerB.hashCode());
		assertNotSame( customerC.hashCode(), customerA.hashCode());
	}
}
package caching.dao;
import java.io.Serializable;

public class Customer implements Serializable {
	private static final long serialVersionUID = -8342637242909019129L;
	public final String id;
	public Customer(final String id) {
		this.id = id;
	}
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((id == null) ? 0 : id.hashCode());
		return result;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
```

```
                Customer other = (Customer) obj;
                if (id == null) {
                        if (other.id != null)
                                return false;
                } else if (!id.equals(other.id))
                        return false;
                return true;
        }
}
package caching.dao;
import static org.junit.Assert.assertNotNull;
import java.util.List;
import org.junit.Test;

public class TestCustomerDao {
        private CustomerDao dao;
        public TestCustomerDao() {
                dao = DaoFactory.getInstance().getCustomerDao();
        }
        @Test
        public void canFindAListOfCustomersByStringIds() {
                List<Customer> customers = dao.findCustomersByIds(new String[] {"someid"});
                assertNotNull("List of customers null", customers);
        }
        @Test
        public void canFindACustomerByStringId() {
                Customer customer = dao.findCustomerById("someid");
                assertNotNull("Customer null", customer);
        }
        @Test
        public void canStoreACustomer() {
                Customer customer = new Customer("1");
                dao.storeCustomer(customer);
                //should assert that you can get the persisted object via the find methods - but not
        //relevant for this prototype
```

```
	}
}
package caching.dao;
import java.util.List;

public interface CustomerDao {
	public List<Customer> findCustomersByIds(String[] ids);
	public Customer findCustomerById(String id);
	public void storeCustomer(Customer input);
}
package caching.dao;
import java.util.ArrayList;
import java.util.List;

public class CustomerDaoRandomImpl implements CustomerDao {
	@Override
	public Customer findCustomerById(String id) {
		Customer object = new Customer(id);
		return object;
	}
	@Override
	public List<Customer> findCustomersByIds(String[] ids) {
		return new ArrayList<Customer>();
	}
	@Override
	public void storeCustomer(Customer input) {
		// TODO not relevant for prototype
	}
}
package caching.dao;
import java.lang.reflect.Proxy;
import org.junit.Test;
import caching.logging.InvocationLogger;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
```

```
import static org.junit.Assert.assertTrue;

public class TestDaoFactory {
	@Test
	public void canCreateDao() {
		DaoFactory factory = DaoFactory.getInstance();
		CustomerDao dao = factory.getCustomerDao();
		assertNotNull(dao);
	}
	@Test
	public void canCreateDynamicProxyDao() {
		DaoFactory factory = DaoFactory.getInstance();
		CustomerDao dao = factory.getCustomerDao();
		assertNotNull(dao);
		assertTrue("dao not instanceof Proxy", dao instanceof Proxy);
	}
	@Test
	public void canProvideReferenceToLoggerWhenCreatedWithout() {
		DaoFactory factory = DaoFactory.getInstance();
		assertNotNull(factory.getLogger());
	}
	@Test
	public void canProvideReferenceToInjectedLogger() {
		InvocationLogger logger = InvocationLogger.getInstance();
		DaoFactory factory = DaoFactory.getInstance(logger);
		assertNotNull(factory.getLogger());
		assertEquals("Logger should be equal", logger, factory.getLogger());
	}
}
package caching.dao;
import java.lang.reflect.Proxy;
import caching.logging.DuplicateInvocationReporter;
import caching.logging.InvocationLogger;
import caching.logging.DaoInvocationHandler;
```

```
public class DaoFactory {
	private static InvocationLogger logger = InvocationLogger.getInstance();
	static {logger.addObserver(new DuplicateInvocationReporter(1));}
	public static DaoFactory getInstance() {
		return new DaoFactory();
	}
	public CustomerDao getCustomerDao() {
		return getCustomerDaoProxy(new CustomerDaoRandomImpl());
	}
	private CustomerDao getCustomerDaoProxy(final CustomerDao dao) {
		return (CustomerDao) Proxy.newProxyInstance(
			dao.getClass().getClassLoader(), new Class[] { CustomerDao.class },
			new DaoInvocationHandler<CustomerDao>(dao, logger)
		);
	}
	public static DaoFactory getInstance(InvocationLogger logger) {
		DaoFactory.logger = logger;
		return new DaoFactory();
	}
	public InvocationLogger getLogger() {
		return logger;
	}
}
package caching.logging;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import java.lang.reflect.Method;
import org.junit.Test;
import caching.cache.Invocation;
import caching.cache.InvocationCache;
import caching.dao.CustomerDao;
import caching.dao.Customer;
import caching.dao.DaoFactory;

public class TestDaoInvocationHandler {
```

```
	@Test
	public void canLogMethodCallOnUnderlyingObject() {
		InvocationLogger logger = InvocationLogger.getInstance() ;
		CustomerDao dao = DaoFactory.getInstance(logger).getCustomerDao();
		Customer object = dao.findCustomerById("objectId");
		InvocationLogEntry entry =
		InvocationLogEntry.createEntry("caching.dao.CustomerDaoRandomImpl.
		findCustomerById: [objectId] -> "+object, 0);
		assertTrue("Logger missing log statement", logger.contains(entry));
	}
	@Test
	public void  canCacheAllDaoInvocations() throws Exception {
		CustomerDao dao = DaoFactory.getInstance().getCustomerDao();
		Customer object = dao.findCustomerById("objectId");
		InvocationCache cache = InvocationCache.getInstance();
		Method method = CustomerDao.class.getDeclaredMethod("findCustomerById",
new		Class[]{String.class});
		Object objectFromCache = cache.get(new Invocation(method, new Object[]
		{"objectId"}));
		assertEquals("Original object should be equal to object from Cache", object,
		objectFromCache);
	}
}
package caching.logging;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import caching.cache.Invocation;
import caching.cache.InvocationCache;

public class DaoInvocationHandler<T> implements InvocationHandler {
	private final InvocationCache cache = InvocationCache.getInstance();
	private final InvocationLogger logger;
	private final T underlying;

	public DaoInvocationHandler(T underlying, InvocationLogger logger) {
```

```
            this.underlying = underlying;
            this.logger = logger;
        }
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            Invocation invocation = new Invocation(method, args);
            Object ret = cache.get(invocation);
            if(ret == null) {
                ret = invokeAndLogInvocation(method, args);
          cache.put(invocation, ret);
          return ret;
            } else {
                return ret;
            }
        }
        private Object invokeAndLogInvocation(Method method, Object[] args)    throws
                                        IllegalAccessException, InvocationTargetException {
            long before = System.currentTimeMillis();
            Object ret = method.invoke(underlying, args);
            long after = System.currentTimeMillis();
            String fullyQualifiedMethodName = underlying.getClass().getName()
            +"."+method.getName();
            logger.log(InvocationLogEntry.createEntry(fullyQualifiedMethodName, args, ret,
        after-before));
            return ret;
        }
}
package caching.logging;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class TestDuplicateInvocationReporter {
        @Test
        public void reportsDuplicateInvocationsWhenThresholdExceeded() {
            InvocationLogger logger = InvocationLogger.getInstance();
```

```
                DuplicateInvocationReporter observer = new DuplicateInvocationReporter(1);
                logger.addObserver(observer);
                InvocationLogEntry entry1 = InvocationLogEntry.createEntry("someinvocation", 1);
                InvocationLogEntry entry2 = InvocationLogEntry.createEntry("anotherentry", 1);
                logger.log(entry1);
                logger.log(entry2);
                assertFalse("Shouldnt be any dupilicated entries yet",
                observer.hasDuplicatedEntries());
                logger.log(entry1);
                assertTrue("Should have dupilicated entries now", observer.hasDuplicatedEntries());
        }
}
package caching.logging;
import java.util.HashMap;
import java.util.Map;
import java.util.Observable;
import java.util.Observer;

public class DuplicateInvocationReporter implements Observer {
        private Map<String, Integer> uniqueEntries = new HashMap<String, Integer>();
        private final int duplicationThreshold;
        private boolean hasDuplicatedEntries = false;

        public DuplicateInvocationReporter(int duplicationThreshold) {
                this.duplicationThreshold = duplicationThreshold;
        }
        @Override
        public void update(Observable o, Object arg) {
                String invocationString = ((InvocationLogEntry) arg).getInvocationString();
                if (uniqueEntries.containsKey(invocationString)) {
                        Integer newCount = uniqueEntries.get(invocationString) + 1;
                        uniqueEntries.put( invocationString, newCount);
                        checkThresholdAndReportIfExceeded(newCount, invocationString);
                } else {
                        uniqueEntries.put(invocationString, 1);
```

```
            }
        }
        private void checkThresholdAndReportIfExceeded(Integer newCount, String
                                                                    invocationString) {
            if (newCount > duplicationThreshold) {
                hasDuplicatedEntries = true;
                reportInvocationString(newCount, invocationString);
            }
        }
        private void reportInvocationString(Integer newCount, String invocationString) {
            System.out.println("Duplicated invocation - count at: "+newCount +" for
            invocationString: "+invocationString);
        }
        public boolean hasDuplicatedEntries() {
            return hasDuplicatedEntries;
        }
}
package caching.logging;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import org.junit.Test;

public class TestInvocationLogEntry {
        @Test
        public void canCreateInvocationLogEntry() {
            InvocationLogEntry entry = InvocationLogEntry.createEntry("invocation string",
        1000);
            assertNotNull("createEntry returned null!", entry);
            assertEquals("execution time should be 1000", 1000L,
            entry.getExecutionTimeInMilliseconds());
        }
        @Test
        public void knowsIfTwoInstancesAreEqual() {
            InvocationLogEntry entry1 = InvocationLogEntry.createEntry("invocation string",
            1000);
            InvocationLogEntry entry2 = InvocationLogEntry.createEntry("invocation string",
            1000);
```

```
        assertEquals("Entry 1 and entry 2 should be equal", true, entry1.equals(entry2));
        assertEquals("Entry 1 and entry 1 should be equal", true, entry1.equals(entry1));
}
@Test
public void knowsIfTwoInstancesAreNotEqual() {
        InvocationLogEntry entry1 = InvocationLogEntry.createEntry("invocation string",
        1000);
        InvocationLogEntry entry2 = InvocationLogEntry.createEntry("another string",
        1000);
        assertEquals("Entry 1 and entry 2 should not be equal", false, entry1.equals(entry2));
}
@Test
public void knowsThatNullIsNeverEqualToAnotherInstance() {
        InvocationLogEntry entry1 = InvocationLogEntry.createEntry("invocation string",
        1000);
        assertEquals("Entry 1 and null not be equal", false, entry1.equals(null));
}
@Test
public void knowsThatAnInstanceIsNotEqualToADifferentObject() {
        InvocationLogEntry entry1 = InvocationLogEntry.createEntry("invocation string",
        1000);
        assertEquals("Entry 1 and a String object should not be equal", false,
        entry1.equals("a string object"));
}
@Test
public void knowsThatAnInstanceIsNotEqualIfTheInvocationstringIsNull() {
        InvocationLogEntry entry1 = InvocationLogEntry.createEntry(null, 1000);
        InvocationLogEntry entry2 = InvocationLogEntry.createEntry("another string",
        1000);
        assertEquals("Entry 1 and entry should not be equal", false, entry1.equals(entry2));
}
@Test
public void knowsAHashCodeWhenInvocationStringIsNull() {
        InvocationLogEntry entry1 = InvocationLogEntry.createEntry(null, 1000);
        assertNotNull("Entry 1 should have a hashCode when invocationString is null",
        entry1.hashCode());
}
```

```
}
package caching.logging;
import java.util.Arrays;

public class InvocationLogEntry {
	private final String invocationString;
	private final String fullyQualifiedMethodName;
	private final Object[] args;
	private final Object returnValue;
	private final long executionTimeInMilliseconds;
	private final long creationTime;

	protected static InvocationLogEntry createEntry(final String invocationString, final long
			executionTimeInMilliseconds) {
		return new InvocationLogEntry(invocationString, executionTimeInMilliseconds);
	}
	public static InvocationLogEntry createEntry(final String fullyQualifiedMethodName,
			Object[] args, Object returnValue, long executionTimeInMilliseconds) {
		return new InvocationLogEntry(fullyQualifiedMethodName, args, returnValue,
			executionTimeInMilliseconds);
	}
	private InvocationLogEntry(final String fullyQualifiedMethodName, Object[] args, Object
			returnValue, long executionTimeInMilliseconds) {
		this.fullyQualifiedMethodName = fullyQualifiedMethodName;
		this.args = args;
		this.returnValue = returnValue;
		this.executionTimeInMilliseconds = executionTimeInMilliseconds;
		this.invocationString = fullyQualifiedMethodName+": "+Arrays.toString(args)+" ->
		"+returnValue;
		creationTime = System.currentTimeMillis();
	}
	private InvocationLogEntry(final String invocationString, final long
			executionTimeInMilliseconds) {
		this.invocationString = invocationString;
		this.executionTimeInMilliseconds = executionTimeInMilliseconds;
		creationTime = System.currentTimeMillis();
		returnValue = null;
```

```
        args = null;
        fullyQualifiedMethodName = null;
}
public String getInvocationString() {
        return invocationString;
}
public long getExecutionTimeInMilliseconds() {
        return executionTimeInMilliseconds;
}
public String getFullyQualifiedMethodName() {
        return fullyQualifiedMethodName;
}
@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime
                        * result
                        + ((invocationString == null) ? 0 : invocationString.hashCode());
        return result;
}
@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        InvocationLogEntry other = (InvocationLogEntry) obj;
        if (invocationString == null) {
                if (other.invocationString != null)
                        return false;
        } else if (!invocationString.equals(other.invocationString))
                return false;
```

```
            return true;
        }
        @Override
        public String toString() {
            return "InvocationLogEntry [args=" + Arrays.toString(args)
                    + ", creationTime=" + creationTime
                    + ", executionTimeInMilliseconds="
                    + executionTimeInMilliseconds + ", fullyQualifiedMethodName="
                    + fullyQualifiedMethodName + ", invocationString="
                    + invocationString + ", returnValue=" + returnValue + "]";
        }
}
package caching.logging;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import java.util.Iterator;
import java.util.Observer;
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.junit.Test;

public class TestInvocationLogger {
        @Test
        public void canLogAnInvocationLogEntry() {
            InvocationLogger logger = InvocationLogger.getInstance();
            InvocationLogEntry entry = InvocationLogEntry.createEntry("log entry", 1000L);
            logger.log(entry);
            assertTrue("Logger doesn't contain new entry: log entry",logger.contains(entry));
        }
        @Test
        public void canCreateALogger() {
            InvocationLogger logger = InvocationLogger.getInstance();
            assertNotNull("Failed to get InvocationLogger instance - got null", logger);
        }
```

```
@Test
public void knowsHowManyEntriesItHas() {
	InvocationLogger logger = InvocationLogger.getInstance();
	logger.log(InvocationLogEntry.createEntry("log entry1", 1000L));
	assertEquals("Logger expected to contain 1 entry", 1, logger.numberOfEntries());
	logger.log(InvocationLogEntry.createEntry("log entry2", 1000L));
	assertEquals("Logger expected to contain 2 entries", 2, logger.numberOfEntries());
}
@Test
public void canProvideIteratorForEntries() {
	InvocationLogger logger = InvocationLogger.getInstance();
	logger.log(InvocationLogEntry.createEntry("entry1", 1000L));
	logger.log(InvocationLogEntry.createEntry("entry2", 1000L));
	Iterator<InvocationLogEntry> iterator = logger.getIterator();
	String allEntries = "";
	while (iterator.hasNext()) {
		allEntries += iterator.next().getInvocationString();
	}
	assertEquals("AllEntries expected to be \"entry1entry2\"","entry1entry2",allEntries);
}
@Test
public void updatesObserversWhenLoggingAnEntry() {
	Mockery context = new Mockery();
	final Observer observer = context.mock(Observer.class);
	final InvocationLogger logger = InvocationLogger.getInstance();
	logger.addObserver(observer);
	final InvocationLogEntry entry =
	InvocationLogEntry.createEntry("invocationString", 0);
	// expectations
	context.checking(new Expectations() {{
		oneOf (observer).update(logger, entry);
	}});
	// execute
	logger.log(entry);
	// verify
```

```
                context.assertIsSatisfied();
        }
}
package caching.logging;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observable;

public class InvocationLogger extends Observable {
        private ArrayList<InvocationLogEntry> entries;
        public static InvocationLogger getInstance() {
                return new InvocationLogger();
        }
        private InvocationLogger() {
                entries = new ArrayList<InvocationLogEntry>();
        }
        public void log(InvocationLogEntry entry) {
                entries.add(entry);
                setChanged();
                notifyObservers(entry);
                clearChanged();
        }
        public boolean contains(InvocationLogEntry entry) {
                return entries.contains(entry);
        }
        public int numberOfEntries() {
                return entries.size();
        }
        public Iterator<InvocationLogEntry> getIterator() {
                return entries.iterator();
        }
}
package caching.logging;
import static org.junit.Assert.assertEquals;
import java.util.Iterator;
```

```
import java.util.Set;
import org.junit.Test;
import caching.dao.CustomerDao;
import caching.dao.DaoFactory;

public class TestLogAnalyser {
	@Test
	public void canReportNumberOfEntries() {
		InvocationLogger logger = createInvocationLoggerWithTenTestEntries();
		LogAnalyzer analyzer = new LogAnalyzer(logger);
		assertEquals("Expected 10 entries",10, analyzer.reportNumberOfEntries());
	}
	private InvocationLogger createInvocationLoggerWithTenTestEntries() {
		InvocationLogger logger = InvocationLogger.getInstance();
		for (int i = 1; i <= 10; i++) {
			logger.log(InvocationLogEntry.createEntry("entry"+i, 0));
		}
		return logger;
	}
	@Test
	public void canReportNumberOfUniqueEntries() {
		LogAnalyzer analyzer = createLogAnalyzerWithTwoUniqueEntries();
		assertEquals("Expected to find 2 unique entries", 2,
	analyzer.reportNumberOfUniqueEntries());
	}
	@Test
	public void canReportNumberOfOccurencesForAnEntry() {
		LogAnalyzer analyzer = createLogAnalyzerWithTwoUniqueEntries();
		assertEquals("Expected to get a count of 2 for entry1",2,
		analyzer.reportNumberOfOccurencesForEntry(InvocationLogEntry.createEntry(
		"entry1", 0)));
		assertEquals("Expected to get a count of 1 for entry2",1,
		analyzer.reportNumberOfOccurencesForEntry(InvocationLogEntry.createEntry(
		"entry2", 0)));
	}
	private LogAnalyzer createLogAnalyzerWithTwoUniqueEntries() {
```

```
        InvocationLogger logger = InvocationLogger.getInstance();
        logger.log(InvocationLogEntry.createEntry("entry1", 0));
        logger.log(InvocationLogEntry.createEntry("entry2", 0));
        logger.log(InvocationLogEntry.createEntry("entry1", 0));
        LogAnalyzer analyzer = new LogAnalyzer(logger);
        return analyzer;
}
@Test
public void canReportNumberOfOccurencesIfNoEntryWasFound() {
        InvocationLogger logger = InvocationLogger.getInstance();
        LogAnalyzer analyzer = new LogAnalyzer(logger);
        assertEquals("Expected to get a count of 0 for entry1",0,
        analyzer.reportNumberOfOccurencesForEntry(InvocationLogEntry.createEntry(
        "entry1", 0)));
}
@Test
public void canReportAverageResponseTimeForMultipleInvocationsOfOneMethod() {
        InvocationLogger logger = InvocationLogger.getInstance();
        InvocationLogEntry entry1 = InvocationLogEntry.createEntry("methodname", new
        Object[]{"arg1"} , "retrunval1", 1);
        InvocationLogEntry entry2 = InvocationLogEntry.createEntry("methodname", new
        Object[]{"arg2"} , "retrunval2", 2);
        InvocationLogEntry entry3 = InvocationLogEntry.createEntry("methodname", new
        Object[]{"arg3"} , "retrunval3", 5);
        InvocationLogEntry entry4 = InvocationLogEntry.createEntry("methodname2", new
        Object[]{"arg4"} , "retrunval5", 5);
        logger.log(entry1);
        logger.log(entry2);
        logger.log(entry3);
        logger.log(entry4);
        LogAnalyzer analyzer = new LogAnalyzer(logger);
        assertEquals("Average for methodname should be 2", 2.66D,
        analyzer.reportAverageExceutionTimeByMethodName("methodname"), 0.01D);
}
@Test
public void canReportUniqueEntriesByOccurence() {
        InvocationLogger logger = InvocationLogger.getInstance();
```

```
                CustomerDao dao = DaoFactory.getInstance(logger).getCustomerDao();
                dao.findCustomerById("customerid1");
                dao.findCustomerById("customerid2");
                dao.findCustomerById("customerid3");
                dao.findCustomerById("customerid3");
                dao.findCustomerById("customerid4");
                dao.findCustomerById("customerid2");
                dao.findCustomerById("customerid5");
                dao.findCustomerById("customerid3");

                LogAnalyzer analyzer = new LogAnalyzer(logger);
                assertEquals(5,analyzer.reportNumberOfEntries());
                assertEquals(5,analyzer.reportNumberOfUniqueEntries());
                Set<InvocationLogEntry> uniqueEntries =
                analyzer.reportUniqueInvocationEntries();
                assertEquals(5,uniqueEntries.size());
                Iterator<InvocationLogEntry> iterator = uniqueEntries.iterator();
                while (iterator.hasNext()) {
                        InvocationLogEntry invocationLogEntry = (InvocationLogEntry)
                        iterator.next();
                        int numberOfOccurences =
                        analyzer.reportNumberOfOccurencesForEntry(invocationLogEntry);
                        System.out.println(invocationLogEntry +" occured "+numberOfOccurences);
                }
        }
}
package caching.logging;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class LogAnalyzer {
        private final InvocationLogger logger;
        public LogAnalyzer(final InvocationLogger logger) {
                this.logger = logger;
        }
```

```
public int reportNumberOfEntries() {
	return logger.numberOfEntries();
}
public int reportNumberOfUniqueEntries() {
	return reportUniqueInvocationEntries().size();
}
public int reportNumberOfOccurencesForEntry(InvocationLogEntry entry) {
	if(logger.contains(entry)) {
		return countEntries(entry);
	} else {
		return 0;
	}
}
private int countEntries(InvocationLogEntry entry) {
	Iterator<InvocationLogEntry> iterator = logger.getIterator();
	int count = 0;
	while (iterator.hasNext()) {
		InvocationLogEntry logEntry = (InvocationLogEntry) iterator.next();
		if (entry.equals(logEntry)) {
			count++;
		}
	}
	return count;
}
public double reportAverageExceutionTimeByMethodName(String
                                                    fullyQualifiedMethodName) {
	Iterator<InvocationLogEntry> iterator = logger.getIterator();
	long totalExecutionTime = 0;
	int count = 0;
	while (iterator.hasNext()) {
		InvocationLogEntry logEntry = (InvocationLogEntry) iterator.next();
		if (fullyQualifiedMethodName.equals(
			logEntry.getFullyQualifiedMethodName())) {
			totalExecutionTime+=logEntry.getExecutionTimeInMilliseconds();
			count++;
```

```
                }
            }
            return (totalExecutionTime*1D) / (count*1D);
        }
        public Set<InvocationLogEntry> reportUniqueInvocationEntries() {
            Iterator<InvocationLogEntry> iterator = logger.getIterator();
            HashSet<InvocationLogEntry> uniqueEntries = new
            HashSet<InvocationLogEntry>();
            while (iterator.hasNext()) {
                InvocationLogEntry entry = (InvocationLogEntry) iterator.next();
                uniqueEntries.add(entry);
            }
            return uniqueEntries;
        }
}
```