

UNIVERSITÀ DEGLI STUDI DI ROMA
"LA SAPIENZA"



SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
LAUREA TRIENNALE in TECNOLOGIE INFORMATICHE

TESI DI LAUREA

"Architettura Software"

"Gedap/Consoc"

RESPONSABILE INTERNO

Prof. Parisi-Presicce Francesco

STUDENTE

Federici Simone

A/A 2006/2007

Indice generale

Introduzione.....	3
Organizzazione della Relazione.....	4
Contesto.....	5
1. Il progetto Gedap.....	5
2. Il progetto Consoc.....	6
Organizzazione del lavoro per il team di sviluppo.....	7
1. La scelta "Maven".....	8
1.1. Organizzazione fisica del progetto.....	11
2. Test per la non regressione.....	12
3. Continuous Integration.....	15
4. Code Review e code Refactoring.....	15
Design dell' Architettura Software.....	18
1. La ricerca di un buon design.....	19
2. Identificazione dei Moduli del progetto.....	24
3. Sequences diagrams.....	25
3.1. Un primo sequence diagram.....	25
3.2. Sequence diagram Webapp.....	26
3.3. Sequence Diagram Use Case Controller.....	27
3.4. Sequence Diagram Manager.....	29
3.5. Sequence Store (Template Method).....	30
4. Component Diagram.....	31
4.1. Factory Component Diagram.....	31
4.2. Architectural Component Diagram.....	32
4.3. Componenti Coinvolti.....	33
5. Scelte di progettazione.....	34
5.1. Riguardo i Frameworks.....	34
5.2. I Framework di Gedap.....	35
5.3. Spring: Container IoC.....	35
5.4. Hibernate: ORM.....	37
5.5. EJB Entity.....	37
5.6. Controller Transazionale e lo stato dello Use Case.....	38
5.7. Clustering with Terracotta.....	39
5.7.1. Cosa è Terracotta DSO.....	39
Altre attività eseguite.....	43
1. View, Casi d'uso e la navigazione accessibile.....	43
2. Lavori Asincroni.....	44
Conclusioni.....	45
Ringraziamenti.....	46
Bibliografia.....	47

Introduzione

Nel Giugno 2006 mi fu lanciata una sfida. Il **Dipartimento della Funzione Pubblica** presso la **Presidenza del Consiglio** richiedeva un sistema software in grado di gestire, rilevare e censire gli **allontanamenti dal lavoro**¹ che ogni anno le Pubbliche Amministrazioni, presenti su tutto il territorio nazionale, effettuano in favore dei propri dipendenti. La sfida era creare una architettura per questo sistema in grado di evolvere e crescere nel tempo, aggiungendo nuove funzionalità nello stesso dominio e prevedendo la possibilità di includere progetti con domini e problematiche diverse. **DFP**² ha infatti diversi progetti che in futuro potrebbero avere la necessità di essere sviluppati o migrati.

Raccolta la sfida, ho cominciato questa esperienza di cui ora mi accingo a raccontare le scelte e le responsabilità da me affrontate.

1 Un allontanamento dal lavoro, conosciuto sotto il nome di '**Istituto**', è il privilegio concesso da una Pubblica Amministrazione ad un Dipendente di assentarsi dal lavoro per espletare attività che derivano da **funzioni Sindacali dirigenziali** o per espletare attività derivanti da **cariche elettive**.

2 DFP – Dipartimento della Funzione Pubblica

Organizzazione della Relazione

Durante tutta la durata del progetto, i miei compiti e ruoli sono stati molteplici. In questa relazione parlerò di due aspetti del mio lavoro che ritengo siano stati i più interessanti e formativi:

- **Organizzazione del lavoro per il team di sviluppo**

Configurazione e impostazione di un ambiente di sviluppo semplice, robusto e modulare per la gestione del progetto in tutto il suo ciclo di vita. Organizzazione delle risorse e impostazione degli strumenti per facilitare la collaborazione e lo scambio di informazione tra i membri del team.

- **Design dell'Architettura Software**

Scelte architettoniche e di design basate su tre principi fondamentali: comunicazione, semplicità e flessibilità. In questa parte della relazione, mostrerò le scelte più significative e importanti adottate durante il progetto.

Contesto

Gedap/Consoc è uno dei progetti che la ditta K-Tech srl ha sviluppato per il Dipartimento della Funzione Pubblica presso la Presidenza del Consiglio, è in produzione attualmente (secondo quarto 2008) con la versione 3.1. Cominciato dalla K-Tech srl nell'ottobre 2006, ha subito successive estensioni, alcune tuttora in fase di sviluppo.

L'architettura è stata ideata ed implementata inizialmente all'interno del progetto Gedap, il primo ad essere stato commissionato, successivamente è stata utilizzata per implementare il servizio Consoc.

Di seguito ho riportato una semplice introduzione del dominio per semplificare e dare un significato agli oggetti che incontreremo durante la relazione. Nell' **OOD**³ gli oggetti del modello di business prendono vita a partire da **entità reali** che esistono nella realtà del cliente è per questo indispensabile fornire al lettore gli strumenti per comprendere al meglio il lavoro svolto.

1. *Il progetto Gedap*

Il **Dipartimento Funzione Pubblica** è il ministero che, fra le altre attività, opera una funzione di controllo sulle attività della **Pubblica Amministrazione**. In particolare **Gedap** è il sistema del Dipartimento della Funzione Pubblica che si occupa di rilevare e monitorare la concessione da parte delle Pubbliche Amministrazioni dell'**Istituto**.

L'**Istituto** è il privilegio di assentarsi dal posto di lavoro concesso ad un **Dipendente Pubblico** per l'espletamento di attività sindacali (**Istituto Sindacale**) o per l'espletamento delle attività che derivano da una carica elettiva (**Istituto Non Sindacale**).

Gli **istituti non sindacali** sono rilevati dal Dipartimento della Funzione Pubblica solo a scopo statistico, e su di loro non viene effettuato alcun tipo di controllo.

Quando a chiedere l'istituto è un'**Associazione Sindacale** per conto di un Dipendente ad

3 OOD - Object Oriented Design

una Pubblica Amministrazione, allora si parla d'**Istituto Sindacale**.

La concessione dell'**Istituto Sindacale** è regolamentata da un accordo o nei **Contratto Collettivo Nazionale Quadro** e nei **Contratto Collettivo Nazionale di Lavoro** che intercorre tra l'**ARAN**⁴, che rappresenta lo **Stato**, e le **Associazioni Sindacali**.

L'**ARAN** svolge ogni attività relativa alla negoziazione e definizione dei contratti collettivi del personale dei vari comparti del pubblico impiego, ivi compresa l'interpretazione autentica delle clausole contrattuali e la disciplina delle relazioni sindacali nelle amministrazioni pubbliche.

Nel **Contratto Nazionale Sindacale** che ha durata biennale, vengono stabilite quali **Associazioni Sindacali** sono rappresentative presso **gruppi prestabiliti** di Pubbliche Amministrazioni. In questo stesso contratto, se il tipo particolare d'istituto lo prevede, vengono stabilite anche delle **quote** assegnate ad una o più **sigle sindacali** nell'ambito di un particolare **tipo di qualifica** (dirigenti, non dirigenti e medici dirigenti). Sulla base del **Contratto Nazionale**, il Dipartimento della Funzione Pubblica è tenuto ad effettuare dei controlli e, nel caso della **rilevazione d'illeciti**, a contestare alle Pubbliche Amministrazioni l'istituto concesso e giudicato illecito.

Inoltre il Dipartimento della Funzione Pubblica è tenuto a produrre una **Relazione Annuale** da presentare alla **Presidenza del Consiglio** nella quale riportare la rilevazione fatta ed eventuali provvedimenti presi.

2. Il progetto Consoc

Consoc è il sistema che permette alle **Pubbliche Amministrazioni** di adempiere all'obbligo annuale di stilare una **dichiarazione** contenente l'**elenco** delle **organizzazioni private** e dei **consorzi** dei quali la stessa Pubblica Amministrazione ha una **partecipazione economica**.

Per ogni organizzazione e consorzio la PA dichiara il **peso economico**, la natura della partecipazione e l'**elenco dei dipendenti** con i loro **ruoli** ed i loro **trattamenti economici**.

4 ARAN - Agenzia per la Rappresentanza Negoziabile delle Pubbliche Amministrazioni

Organizzazione del lavoro per il team di sviluppo

L'avvio del progetto è stato un momento critico. C'erano nel progetto sviluppatori con esperienze diverse e molti di loro non si conoscevano. I tempi inoltre erano molto stretti, il progetto vasto e non si conoscevano ancora in dettaglio tutte le funzionalità richieste.

Il lavoro non partiva da una solida e strutturata analisi a priori (metodo **Waterfall**⁵) dal momento che si era scelto, in accordo con il cliente, di lavorare con metodologia **UP**⁶.

L'esigenza di predisporre il sistema al cambiamento e la possibilità di una continua evoluzione sono stati i punti cruciali su cui si sono basate le scelte architetturali.

Il primo passo è stato creare una forte e netta separazione dei moduli applicativi: **view**, **core** e **persistence** (gedap-webapp, gedap-bo, gedap-persistence). Le scelte effettuate sono state dettate dalle esigenze legali e logistiche attuali, ma un domani si potrebbe avere la necessità di delegare la persistenza a degli **EJB**⁷ remoti per aumentare la sicurezza o per necessità di performance, oppure si potrebbe cambiare completamente l'interfaccia grafica web (interazione con l'utente). Questa infatti è stata sviluppata in modo molto semplificato per garantire l'accessibilità secondo la **Legge Stanca**⁸, ma un domani, con lo sviluppo sempre più avanzato di **browser speciali**⁹, questa legge potrebbe essere resa meno restrittiva e DFP potrebbe richiedere un intervento grafico sul sistema.

Il modulo a cui si è data più importanza è quindi il "core" o meglio il business del progetto. I processi di business del cliente potranno evolvere, ma mai verranno buttati e fatti da capo. Certo anche in questo caso il progetto deve prevedere il cambiamento seguendo l'evoluzione del mondo reale, ma l'ambito business non sarà mai così volatile come una restrizione Legale sull'accessibilità o un requisito non funzionale come ad esempio la **scalabilità**¹⁰.

5 Waterfall – processo di sviluppo software che prevede una fase di analisi a priori rispetto alla fase di design e sviluppo

6 UP – Unified Process, processo di sviluppo software iterativo, dove le fasi di analisi, design e sviluppo procedono parallelamente in una continua iterazione.

7 EJB – Enterprise JavaBean, componente architetturale server-side per la costruzione modulare di applicazioni Enterprise

8 Legge Stanca – Legge sulla accessibilità, disposizioni per favorire l'accesso dei soggetti disabili agli strumenti informatici

9 Browser speciali – Browser vocali, braille o con alcune funzioni lato client disabilitate (es. javascript o css)

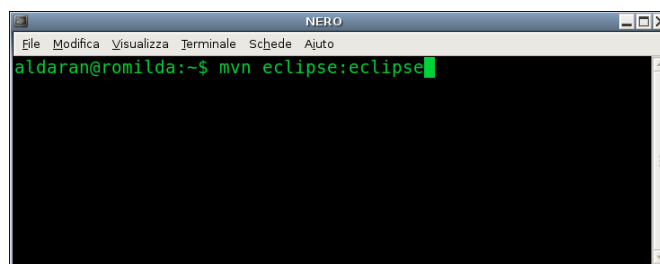
10 Scalabilità – Capacità di un sistema di "crescere" o "decrescere" (aumentare o diminuire di *scala*) in funzione delle necessità e delle disponibilità.

1. La scelta "Maven"

Inizialmente gli sviluppatori del team, hanno contestato molto Maven come strumento per gestire il build del progetto, probabilmente per la sua natura separata e non integrata ad un ambiente di sviluppo grafico. A mio parere, troppo spesso si delega a un **IDE**¹¹ la gestione di un progetto software. Gli IDE di sviluppo cambiano, ogni programmatore ha il suo preferito e quindi uno sviluppatore che utilizzi un IDE diverso da quello con cui si è creato il progetto è costretto a ricreare un ambiente idoneo, a riscrivere gli script di build, a gestire separatamente il deploy e le librerie esterne e molto altro ancora. Tutto questo, a discapito della produttività.

Maven è uno strumento di software life-cycle management, che permette di descrivere in modo standard, tramite un linguaggio formale (**XML**¹²), il ciclo di vita del software in tutte le sue fasi. Si basa su un principio fondamentale: **Convention over Configuration**. Un "default" nella configurazione rende tutto più semplice, Maven permette di descrivere il progetto con il suo ciclo di vita fin nei minimi dettagli, ma non è più obbligatorio. Vengono descritte quindi solo le fasi "personalizzate".

Maven inoltre permette l'integrazione con gli IDE di sviluppo, mediante dei plugin eseguibili a linea di comando:



I tre plugin sotto elencati, generano automaticamente i file necessari ad ogni IDE di sviluppo per gestire il progetto, impostando il classpath e alcune configurazioni specifiche descritte nel **pom.xml**¹³.

- *mvn eclipse:eclipse*
- *mvn idea:idea*
- *mvn netbeans:netbeans*

11 IDE – Integrated development environment, ambiente di sviluppo integrato per la realizzazione di programmi informatici

12 XML – eXtensible Markup Language, è un metalinguaggio utilizzato per creare nuovi linguaggi, atti a descrivere documenti strutturati, spesso usato come mezzo per l'esportazione di dati o per le configurazioni di sistemi informatici.

13 Pom.xml – Project Object Model, file di configurazione di un progetto maven.

A volte si devono prendere decisioni che vanno in una direzione non sempre chiara a tutti, ma per me era chiarissima, ed oggi non c'è nessuno del nostro team che non inizierebbe un progetto con questo strumento.

Maven permette l'organizzazione precisa e dichiarativa del progetto, tutto è centralizzato, chiunque prenda in mano il progetto, leggendo la configurazione dei pom.xml, è in grado di vedere tutti i passi del processo di sviluppo: la compilazione, il filtraggio delle risorse, i test unitari, i test di integrazione, il tipo di packaging (**jar**, **war**, **ear**¹⁴), il deploy, il rilascio di una nuova release, le dipendenze e il loro scope (compile, test, runtime, provided e system), il sistema di versioning, il sistema di gestione anomalie (issue tracking) e tanto altro ancora.

Inoltre Maven, istruito a dovere, può generare un sito documentale pubblicando una serie di informazioni e report di ogni tipo.



Gedap Project 1.1-SNAPSHOT | Ultima Pubblicazione: 24/gen/2008 10:57 Home | Wiki Gedap | Wiki Consec | X-Tech

Getting Started

- Repository
- Profilo
- SQLLog

Documentazione del Progetto

- ▼ Informazioni sul Progetto
- Benvenuto in
- Continuous Integration
- Dependencies
- Dependency Convergence
- Issue Tracking
- Mailing Lists
- Project Licence
- Project Summary
- Project Team
- Source Repository
- ▼ Rapporti del Progetto
- Change Log
- ▼ Developer Activity
- File Activity**
- JavaDocs
- JavaNCSS Report
- PMD Report
- QALab Main Report
- QALab Movers Report
- Source Xref
- Stats SCM
- Tag List
- Test Source Xref

Moduli

- Gedap Interfaces
- Gedap Persistence
- Gedap Business Logic
- Gedap Webapp
- Integration 1
- SchedulerJobs

Links

- Wiki
- X-Tech

Info

- Guides
- FAC
- Repository Stats

File Activity Report

Changes between 2007-12-25 and 2008-01-25

Total commits: 223
Total number of files changed: 608

Filename	Number of Times Changed
/branches/Gedap_Est/gedap-persistence/src/main/java/it/tech/gedap/persistence/dao/istituti/ IstitutoDichiaratoDAO.java	22
/branches/Gedap_Est/gedap-persistence/src/main/resources/it/tech/gedap/persistence/dao/istituti/ IstitutoDichiaratoDAO.properties	18
/branches/Gedap_Est/gedap-bo/src/main/java/it/tech/gedap/istituti/dichiarati/ AbstractIstitutoDichiarato.java	13
/branches/Gedap_Est/gedap-bo/src/main/java/it/tech/gedap/istituti/ IstitutoSindacaleGradoDichiarato.java	13
/branches/Gedap_Est/gedap-persistence/src/main/java/it/tech/gedap/persistence/dao/ AbstractConnectionDAO.java	12
/branches/Gedap_Est/gedap-bo/src/main/java/it/tech/gedap/bo/managers/ DBSearchManager.java	12
/branches/Gedap_Est/gedap-persistence/src/main/java/it/tech/gedap/persistence/dao/ PublicaAmministrazioneDAO.java	12
/branches/Gedap_Est/gedap-persistence/src/main/java/it/tech/gedap/persistence/dao/ ReportDAO.java	11
/branches/Gedap_Est/gedap-webapp/src/main/java/it/tech/gedap/view/struts/action/ RegistrazioneIstitutedaDFPAction.java	11
/branches/Gedap_Est/gedap-bo/src/main/java/it/tech/gedap/istituti/fissi/ IstitutoSindacaleGrado.java	11
/branches/Gedap_Est/gedap-bo/src/main/java/it/tech/gedap/istituti/ PermessoFunzionePubblicaDichiarato.java	11
/branches/Gedap_Est/gedap-interfaces/src/main/java/it/tech/gedap/interfaces/persistence/dao/istituti/ IstitutoDichiaratoDAO.java	11
/branches/Gedap_Est/gedap-webapp/src/main/resources/it/tech/gedap/view/struts	11

Ad esempio può pubblicare i possibili bugs segnalati da **findbugs**¹⁵, le parti duplicate del codice (**PMD**¹⁶), una serie di best practise sulla formattazione e la scrittura del codice (**checkstyle**¹⁷), il **javadoc**¹⁸ aggregato dei moduli con l'estensione UML (**umlgraph**¹⁹), la copertura dei test (**cobertura**²⁰), le statistiche dell'utilizzo del source version system

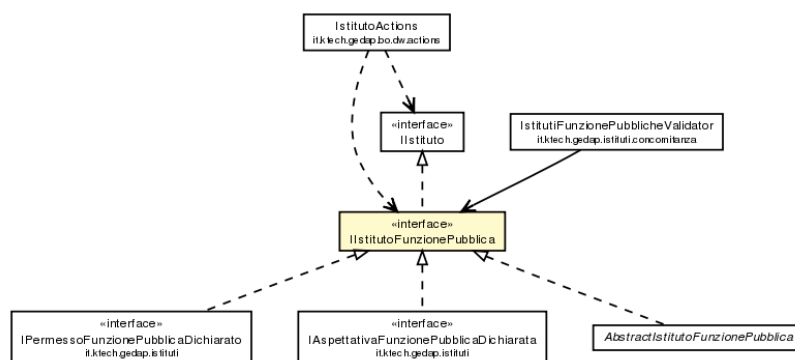
14 Jar, war, ear, ejb, ejb3, par, rar – Possibili formati dei progetti maven, rispettano gli standard per gli archivi JEE
 15 Findbugs – programma open source che, attraverso l'analisi statica del codice, identifica centinaia di potenziali tipi di errore
 16 PMD – programma che analizza il codice sorgente alla ricerca di duplicate code, dead code, classi con complessità ciclotomiche elevate, codice che usa metodi non preformanti, ecc.
 17 Checkstyle – programma per il controllo del codice sorgente, per problemi di layout, problemi di design, duplicate code, o bug patterns come il “double checked locking”
 18 Javadoc – è un applicativo incluso nel JDK, per la generazione automatica della documentazione del codice sorgente.
 19 Umlgraph – strumento per la generazione dichiarativa di class diagram e sequence diagram. Usato per la generazione dei grafici all'interno della documentazione javadoc.
 20 Cobertura – è un free Java tool che calcola la percentuale di codice acceduto dai test.

(StatSCM²¹), le metriche dei package con le dipendenze **efferenti**²² e **afferenti**²³, l'astrattezza e la stabilità di ogni package (**Jdepend**²⁴). Tutto collegato al codice sorgente pubblicato tramite **JXR**²⁵.

Gli stessi strumenti sono stati poi installati sugli IDE di sviluppo e sono state pubblicate sul sito le guide per la loro installazione e per il loro utilizzo.

Un sito pubblico che espone delle misure di qualità del software, rende il progetto più chiaro nella sua globalità, aumenta così la coscienza del lavoro svolto e le capacità/conoscenze dei membri del team di sviluppo.

Immagine di un class diagram generato automaticamente da UMLGRAPH e inserito automaticamente nel Javadoc della interfaccia "IstitutoFunzionePubblica":



Fondamentale, è stato anche dare uno standard per la formattazione del codice. Dal momento che non tutti gli sviluppatori usano lo stesso IDE, si è scelto di usare uno strumento di formattazione esterno come **Jalopy**²⁶, i cui punti di forza sono l'integrazione con tutti gli IDE e la possibilità di condividere una configurazione unica.

L'importanza della formattazione del codice è legata non solo alla velocità di lettura da parte di un altro sviluppatore, ma anche alla capacità di poter usare con semplicità gli strumenti di confronto tra versioni diverse. Capita spesso infatti di dover unificare due file sostanzialmente uguali ma con formattazioni diverse. Formattando invece il codice nello stesso modo, si evitano ore intere a fare "merge" tra versioni diverse dello stesso file, come a volte capita quando si lavora in gruppo sugli stessi file sorgenti.

21 StatSCM – genera le statistiche sull'attività di commit degli sviluppatori nei source versions system (SVN o CVS)

22 Dipendenze Efferenti – è il numero di package da cui le classi del package/classe attuale dipendono.

23 Dipendenze Afferenti – è il numero di package che dipendono da un determinato package/classe

24 Jdepend – applicazione java che calcola le metriche di dipendenza tra i package.

25 JXR – è un generatore di sorgente con riferimenti incrociati, produce file HTML files che mostrano il codice sorgente annotato.

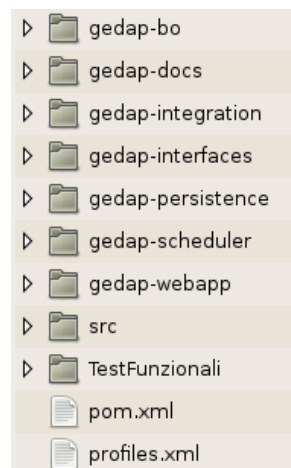
26 Jalopy – è un formattatore di codice sorgente basato su delle regole configurabili.

1.1. Organizzazione fisica del progetto

Il progetto è stato strutturato usando una gestione multi-modulo, quindi un progetto padre “Gedap Project” (nella root directory), e i suoi sotto-moduli.

Questa è la sezione del pom “Gedap Project” (il padre) che descrive le relazioni con i suoi sotto moduli:

```
<modules>
  <module>gedap-interfaces</module>
  <module>gedap-bo</module>
  <module>gedap-persistence</module>
  <module>gedap-webapp</module>
  <module>gedap-integration</module>
  <module>gedap-scheduler</module>
</modules>
```



Il pom.xml è il file dove viene descritto il progetto e da questo è possibile configurare tutto il ciclo di vita del software. Ogni sotto-modulo ha il suo pom.xml ed eredita tutti i comportamenti del modulo padre.

Inoltre nel profiles.xml vengono definiti delle proprietà e dei comportamenti diversi a seconda del profilo scelto. Ad esempio sono state definite le configurazioni degli ambienti di su profili diversi: sviluppo, test ed esercizio.

- `mvn compile -P sviluppo,debug`
- `mvn test -P test`
- `mvn package -P esercizio,nolog`

I profili di maven permettono la riscrittura di alcune “variabili d'ambiente” che, durante la fase di preparazione del sorgente, ossia prima della compilazione, filtrano i contenuti delle risorse sostituendo le variabili con i loro valori. È quindi possibile ad esempio cambiare lo “schema” del database (personalizzando le variabili nei profili *sviluppo*, *test* e *esercizio*) oppure modificare il livello dei log (profili *debug* e *nolog*).

2. *Test per la non regressione*

L'esigenza di predisporre il sistema al cambiamento e la possibilità di una continua evoluzione implicano una condizione indiscutibile: il mantenimento di una suite di test per la non regressione. Come essere sicuri che un cambiamento non provochi danni?

Spesso si pensa che scrivere dei test per l'applicazione non sia importante quanto scrivere del buon codice. Ma il software evolve, il codice cambia o aggiunge nuovi comportamenti ed è ormai provato statisticamente che ogni tre modifiche fatte ad un codice, subentra un possibile bug. È quindi indiscutibile che senza una base solida di test non è possibile scrivere una applicazione agile (disponibile al cambiamento).

I test comunque sono e rimangono componenti per auto-testare il sistema, che però non aggiungono funzionalità. Per questo motivo si tende, in momenti critici, come ad esempio nei giorni di consegna, a lasciarsi alle spalle: "li farò domani, oggi non ho tempo!". Questa frase è tanto tipica quanto pericolosa.

È facile constatare che un codice viene letto molte più volte di quante venga scritto. Lo stesso vale per i test, vengono scritti una volta e poi continuano a garantire che tutto il sistema risponda come previsto.

La scrittura dei test dovrebbe essere sempre parallela allo sviluppo di nuove funzionalità, a volte però questo non accade. In questo senso ci è stato utile pubblicare sul sito generato da Maven la copertura dei test (plugin Cobertura): linee di codice verdi se i test hanno avuto successo, rosse se sono falliti, nessun colore se i test non sono passati.

In questo modo è sempre stato possibile, non solo seguire l'andamento dei test, ma anche verificare quali linee di codice non erano ancora state testate e risolvere a posteriori le eventuali mancanze.

Non abbiamo invece usato il **TDD**²⁷. Credo sia un metodo molto interessante, ma quando lo si usa, il dialogo tra il team di sviluppo e il cliente deve essere diretto e inoltre si dovrebbe programmare in **Pair**²⁸. Questo non era il nostro caso, spero tuttavia in futuro di aver la possibilità di provarlo, possibilmente in un progetto molto più piccolo.

²⁷ TDD – Test Driven Development, metodologia agile per lo sviluppo del software. Scrivere dei test per guidare il design.

²⁸ Pair Programming – Una delle tecniche di base dell'eXtreme Programming, programmare con 4 mani e 4 occhi.

Vale la pena approfondire le tipologie di test che sono state usate in questo progetto:

1. **Test Unitari + Mock Object²⁹ (Unit Test)**

Ogni test viene eseguito su uno e un solo componente, gli oggetti da cui dipende vengono simulati per isolarlo, in questo modo se il test fallisce, si può individuare l'errore in modo immediato.

2. **Test di Integrazione (Module/Integration/System Test)**

Si testa un componente in un sistema pienamente funzionante. È da tener conto che non si sta solamente testando il singolo componente, ma tutti i componenti chiamati a partire da esso. I test di integrazione si dividono in tre sottogruppi:

- **Module Test:** controllano solamente un singolo modulo (es. persistence)
- **Integration Test:** controllano più di un modulo (es. business e persistence)
- **System Test:** controllano l'intero sistema, ossia tutti i moduli

3. **Acceptance Test (Functional Test)**

Il loro nome enfatizza il fatto che i test devono dimostrare/garantire l'aderenza del software ai requisiti e di conseguenza mettere in grado il cliente di accettare o meno il prodotto (mediante la definizione di precisi criteri di accettazione). Sono quindi test mostrati e fatti in collaborazione con il cliente.

4. **Performance testing (Not-Functional Test)**

Sono test per la verifica delle performance dell'applicazione.

- I **Load Test** controllano il comportamento dell'applicazione in condizioni di carico (utenti) normali e di picco. Questi test servono a verificare se con quella combinazione di software e hardware vengono soddisfatti i requisiti forniti e richiesti dal cliente (**Performance SLA³⁰**).
- Gli **Stress test** verificano come si comporta l'applicazione in condizioni di carico elevatissimo. Servono sostanzialmente per trovare il **punto di rottura**, oltre il quale l'applicazione inizia a produrre errori di vario genere.
- I **Capacity test** verificano il livello di scalabilità dell'applicazione o meglio rispondono alla domanda del cliente: “che cose debbo fare se il numero di utenti raddoppia?”

²⁹ Mock Object – sono degli oggetti simulati/finti che mimano il comportamento degli oggetti reali in maniera controllata

³⁰ Performance SLA – Performance Service Level Agreement, strumenti contrattuali che definiscono le metriche di servizio che devono essere rispettate (disponibilità, tempi di risposta, ecc.)

Fra tutti questi Test i più importanti sono chiaramente gli Acceptance Test. Se fallisce l'aspettativa del cliente, il problema da piccolo diventa grande, cresce la tensione nel team e aumentano le aspettative del cliente. Per avere la sicurezza del pieno successo di questi test, devono innanzitutto non fallire i test di basso livello: Unit Test e Integration Test.

I Performance Test servono invece per controllare e per far emergere le possibili problematiche relative a performance, connessioni al database non chiuse, problemi a livello di **HEAP**³¹ della **JVM**³² ecc.

Per la non regressione, i test più importanti sono i test unitari, ma sono anche i più difficili da scrivere. Sicuramente sono i più potenti perché, testando un solo componente, isolano il problema. Tuttavia non è così banale scrivere uno Unit Test, perché questo implica l'uso dei mock object per ogni dipendenza efferente, inoltre introduce la necessità di scrivere i componenti in modo che sia possibile simulare i loro oggetti dipendenti. È per questi motivi e perché implicano un grande dispendio di energie, che gli Unit Test sono una impresa difficile da portare avanti.

Un Integration Test è molto più semplice da scrivere, non ci si preoccupa di isolare il componente, ma lo si testa nell'insieme. L'unico problema è che a volte falliscono non per un bug, ma perché è cambiato un valore nel database o perché sono cambiate delle specifiche non relative a quel modulo. Nel complesso, un Integration Test ha bisogno di più manutenzione di uno Unit Test, infatti può fallire ad ogni cambiamento dei componenti da esso coinvolti.

In conclusione, per avere una solida base di test di non regressione, ritengo importante usare Unit Test, almeno negli oggetti più critici della applicazione. Per tutti gli altri oggetti è possibile anche essere coperti solamente dagli Integration Test.

31 HEAP – Spazio della memoria della JVM allocato dinamicamente, dove risiedono gli oggetti necessari alla applicazione.

32 JVM – Java Virtual Machine, macchina virtuale dove a runtime vengono eseguiti programmi java

3. *Continuous Integration*

Una volta che si ha una base di test che garantiscono la non regressione, è buona pratica lanciare tutta la suite di test ogni giorno, in modo da accorgersi immediatamente se un cambiamento al sistema ha causato il fallimento di un test.

Il sistema di continuous integration ogni notte crea un **nightly build**³³ e lancia tutte le suite di test, quindi in caso di fallimenti manda una mail di avviso.

Noi abbiamo usato uno strumento di nome **Continuum**³⁴, che ha il vantaggio di essere integrato al 100% con Maven. Questo strumento è stato configurato settimanalmente anche per far generare e pubblicare il sito di reportistica di Maven

4. *Code Review e code Refactoring*

Con la Continuous Integration e una base di test di non regressione, che in un certo senso si occupano di controllare il lavoro svolto, ogni membro del team si può concentrare sulla scrittura del codice, senza paure di rompere qualcosa. Questo timore infatti nasce molto spesso al momento di fare refactoring, ossia quando si effettuano dei cambiamenti strutturali al sistema per migliorarne la struttura interna, senza però cambiare il comportamento atteso del sistema. Sembra un punto banale, ma non lo è. Mantenere una base di test è uno dei passi per costruire un sistema robusto.

Per risolvere un determinato problema, uno sviluppatore dovrebbe sempre pensare all'**algoritmo migliore**, alla soluzione ottimale. Spesso però, in un software, di problemi critici che necessitano di soluzioni complesse, ce ne sono pochi. Il 90% del codice di un prodotto utilizza algoritmi semplici. In questi casi i valori basilari da tener presente sono **Comunicazione, Semplicità e Flessibilità**³⁵. Valori che aiutano a scrivere un software capace di evolvere.

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” [M. Fowler, "Refactoring"]

33 Nightly build – build automatico notturno, rilascio di versioni SNAPSHOT ossia versioni non ufficiali del branch in sviluppo

34 Continuum – Apache Maven Continuum, è un server per la continuous integration server per schedulare i build dei progetti java.

35 Comunicazione, semplicità e flessibilità – Valori espressi da Kent Beck in "Implementation Patterns"

Un software deve comunicare qualcosa a chi lo legge. Non bisogna scrivere per se stessi, né per la macchina. Una macchina non distingue un buon design da una programmazione procedurale e tanto meno da uno **spaghetti code**³⁶, mentre un essere umano si.

Riassumendo:

- *Comunicazione*, il software deve dirci cosa fa (non lo devono fare i commenti)
- *Semplicità*, deve fare le cose nel modo più semplice possibile
- *Flessibilità*, deve essere capace di adattarsi alle modifiche o alle estensioni

When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous. [M. Fowler, "Refactoring"]

Questi valori sono importanti nella programmazione, soprattutto durante il Code Review e il Refactoring, due strumenti ai quali abbiamo dato forte importanza. Li chiamo strumenti, perché ritengo l'intelletto umano il più importante strumento che esista. Un programmatore è tanto più bravo quanto è immaginativa la sua mente nel creare le giuste metafore per disegnare la realtà.

Kent Beck in "**Implementation Pattern**³⁷" da un'immagine della programmazione molto più vicina alla filosofia che alla matematica. Elenco qui brevemente i suoi principi di programmazione:

- *Local Consequences*
- *Minimize Repetition*
- *Logic and data Together*
- *Symmetry*
- *Declarative Expression*
- *Rate of Change*

³⁶ Spaghetti code – un classico anti-pattern, un codice con una struttura di controllo del flusso complessa e/o incomprensibile, con uso esagerato ed errato dei costrutti di programmazione che lo si può paragonare ad un piatto di spaghetti.

³⁷ K. Beck, "Implementation Patterns"

Sembra impossibile, arrivato a questo punto del discorso, non citare anche i design pattern, gli antipattern e i code smell la cui importanza è tale da essere diventati l'ABC di ogni sviluppatore.

- **Design Pattern**

Individuano una soluzione progettuale generale ad un problema ricorrente. Consistono in una descrizione o in un modello da applicare per risolvere un problema, che può presentarsi in diverse situazioni durante la progettazione e lo sviluppo del software. I primi ad aver raggruppato e formalizzato alcuni design pattern sono la **Gang of Four**³⁸.

- **Antipattern**

Anche gli antipattern possono essere definiti "pattern", in quanto sono una soluzione progettuale generale ad un problema ricorrente e sono costituiti da un nome, un problema e una soluzione. La differenza tra un normale pattern (o "Pattern Migliorativo") ed un antipattern è che il primo è caratterizzato da una soluzione fondamentalmente buona/positiva, mentre il secondo è caratterizzato da una falsa soluzione o da una soluzione fondamentalmente negativa.

- **Code Smell**

Sentite puzza di bruciato? I code smell sono dei sintomi che indicano che qualcosa nel codice potrebbe essere sbagliato. Sugeriscono che una parte di codice dovrebbe essere rifattorizzata o che il design dovrebbe essere riesaminato. Il termine sembrerebbe essere stato coniato da Kent Beck sul **WardsWiki**³⁹ per poi diventare popolare grazie alla celebre pubblicazione di Martin Fowler "**Refactoring: Improving the Design of Existing Code**"

38 GoF - Gang of Four ("banda dei quattro"), è usato per riferirsi agli autori del libro Design Patterns: Elements of Reusable Object-Oriented Software: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides

39 WardsWiki – un wiki storico, uno dei primi wiki mai pubblicati

Design dell'Architettura Software

Per organizzare il lavoro in modo da costruire un software Flessibile, Robusto e Riutilizzabile, ho cercato di tenere il sistema disaccoppiato, separando quindi i comportamenti della applicazione a partire dai moduli strutturali fino ad arrivare ai package e alle classi. Il sistema quindi è stato diviso in più moduli. Ogni modulo è stato pensato ed organizzato per offrire un servizio specifico nel contesto applicativo, per il quale è l'unico responsabile. Ogni modulo è per questo autonomo e disaccoppiato dal funzionamento degli altri moduli applicativi.

Questa modularità ha garantito una costruzione del software organizzata, ordinata e quindi maggiormente leggibile. Tenendo conto di questo principio e applicandolo a partire dall'architettura fino ad arrivare alle singole classi, si è riusciti ad abbassare i tempi di sviluppo e conseguentemente i costi del progetto. In questo modo è stato possibile aumentare la velocità di erogazione del prodotto, misurata dal fattore **Time To Market**⁴⁰ e aumentare dunque il fattore **Return Of Investment**⁴¹.

Per rafforzare la separazione dei moduli, è stato creato un modulo composto solamente di interfacce, verso il quale tutti i moduli hanno una unica dipendenza (a livello di compilazione). Il modulo in questione è "Gedap Interfaces". In poche parole ogni modulo è "accoppiato" solamente a "Gedap Interfaces" e non ci sono (non esistono) relazioni tra gli altri moduli del sistema.

40 TTM – tempo che intercorre dall'ideazione di un prodotto alla sua effettiva commercializzazione

41 ROI – indice di redditività del capitale investito

1. *La ricerca di un buon design*

La progettazione di un software, a partire dalla sua architettura, deve sempre tenere conto dei rischi di un cattivo design, prima di tutto quindi è necessario analizzare quali sono i sintomi di un cattivo design:

- **Rigidità:** Ogni cambiamento (piccolo o grande) influenza troppe parti del sistema, rendendo così difficile e meticoloso il cambiamento. Chiaramente questo è dovuto ad un alto accoppiamento delle parti del sistema. Il principio **Low Coupling**⁴² è stato violato.
- **Fragilità:** I cambiamenti al sistema sono pericolosi. Modificare una parte di codice può creare dei mal funzionamenti altrove. Un cambiamento diventa un problema. L'unica soluzione è avere una suite di test per la non regressione che possano continuare a garantire il funzionamento corretto.
- **Immobilità:** Difficoltà nell'estrarre e isolare delle parti del software. È difficile il riutilizzo dei moduli, poiché parti di interesse dello stesso design sono strettamente correlate con parti che non si vogliono riutilizzare. Spesso, il costo della separazione dei moduli da estrapolare, è maggiore del costo del loro rifacimento.
- **Viscosità del disegno:** Leggibilità difficile, disegno poco chiaro e soprattutto molto articolato. Gli sviluppatori, per fare qualcosa, cercano sempre la strada più semplice, alcune di queste vie preservano il disegno, altre no. Se la viscosità del disegno è alta, una modifica non banale porterà quasi sicuramente a violare il disegno.
- **Viscosità dell'ambiente:** Ambiente complesso, sistemi di build non chiari, packaging e deploy separati, non centralizzati. Gli sviluppatori prenderanno decisioni di sviluppo che riducono l'impatto dell'ambiente, scordandosi il disegno, reinventando la "ruota"⁴³. Soprattutto nei progetti che evolvono, crescono e si ampliano, l'ambiente deve essere comodo, integrato e soprattutto comprensibile.

Elencare i sintomi è facile, più difficile è saperli prevedere ed evitare. Il compito di un progettista è dare gli strumenti agli sviluppatori per far sì che il design non venga corrotto. Innanzitutto quindi bisogna organizzare la struttura del progetto: moduli, packages,

⁴² Low Coupling – Uno dei Principi GRASP di cui Craig Larman scrive in "Applying UML and Patterns"

⁴³ Antipattern Reinventing the wheels – (Reinventare la ruota) quando in un sistema si riscrive del codice per compiere qualcosa che è già stato fatto in un'altra parte del programma.

interfacce, ruoli e comportamenti, rispettando dei principi di buon design.

Elenco qui di seguito i principi più importanti per il disegno di una architettura:

- **Open Closed Principle (OCP)**

Un modulo deve essere aperto all'estensione e chiuso alle modifiche. La riusabilità dei componenti non deve derivare dalla loro modifica, si dovrebbe progettare in modo che un modulo possa sempre essere esteso. Ma è anche vero che lo si dovrebbe rendere stabile, quindi bisognerebbe riuscire a dargli le capacità evolutive senza bisogno di aggiungere nuovi comportamenti.

- **Dependency Inversion Principle (DIP)**

È importante dipendere dalle astrazioni e non da concetti concreti. Creare una sorta di framework stabile composto di classi astratte e interfacce. Infatti le implementazioni possono cambiare di frequente e un loro eccessivo accoppiamento potrebbe rendere il sistema troppo rigido. Le astrazioni invece cambiano più raramente e quindi dovrebbero essere la base stabile dell'applicazione.

- **Interface Segregation Principle (ISP)**

Scrivere una interfaccia per ogni client, anche se numerosi, è sicuramente meglio di avere un'unica interfaccia general purpose per tutti. Questo non è un concetto banale, spesso si tende a pensare che un numero di classi/interfacce troppo elevato rende il software illeggibile o ingestibile. Le interfacce dovrebbero identificare il "cosa" deve fare un oggetto (non il "come", che invece riguarda l'implementazione). È chiaro che, come è difficile dare un nome ad interfacce con comportamenti multipli lo è altrettanto riusare quelle stesse interfacce. Quello che si cerca di evitare quindi sono le Fat interfaces, dove i client sono costretti a implementare responsabilità che non gli competono.

- **Release Reuse Equivalency Principle (REP)**

La granularità del riuso è la granularità della release. È importante tenere un sistema che preveda il cambio di release in modo da poter far evolvere il software senza infrangere l'esistente. Questo concetto è legato al rilascio e all'utilizzo di librerie esterne al progetto. Se non c'è un rilascio del software con una precisa

versione, nessuno potrà affidarsi alla libreria. Un esempio è la K-Tech Library, modulo esterno al progetto gedap in comune a tutti i progetti K-Tech. Ogni volta infatti che viene rilasciata una versione ufficiale di gedap, è necessario che anche la K-Tech library abbia un avanzamento di versione.

- **Common Closure Principle (CCP)**

Le classi che cambiano insieme, vivono insieme. Bisogna cercare sempre di mettere nello stesso modulo o anche nello stesso package, le classi che sono fortemente collegate, nel senso che se cambia una, cambia l'altra. Identificare quindi i componenti che vivono insieme e organizzarli insieme, è un difficile compito, ma rispettare questo principio dà ad ogni sviluppatore la visibilità e la conoscenza globale del sistema e serve inoltre anche a mantenere ordinato e leggibile il codice.

- **Common Reuse Principle (CRP)**

Classi non riusate insieme non dovrebbero essere raggruppate insieme. Bisogna cercare sempre di separare le classi che vengono riusate da quelle che non vengono riusate o anche separare quelle che vengono riusate in due contesti diversi. Questo principio è molto simile al CCP, ma identifica il concetto di riuso.

- **Acyclic Dependencies Principle (ADP)**

Le dipendenze tra packages non devono essere cicliche. Una dipendenza ciclica è un sintomo di immobilità. $A \rightarrow B \rightarrow C \rightarrow A$, se volessi prendere A, B o C dovrei comunque prenderli tutti e tre ossia avrei difficoltà ad estrarre le singole parti. Se tra i package non dovrebbe mai verificarsi una ciclicità, tra le classi questo accoppiamento bidirezionale è permesso, purché limitato a due massimo tre classi. Se però sono i packages ad essere circolari, vuol dire che la distribuzione delle classi nei package non ha probabilmente rispettato CCP e CRP.

- **Stable Dependencies Principle (SDP)**

Le dipendenze dovrebbero essere in direzione della stabilità, ossia ogni modulo

dovrebbe dipendere solo da moduli più stabili. Questo concetto si avvicina molto al concetto di DIP identificandone la direzione. Per cui se ordiniamo i packages in ordine di stabilità, quelli più stabili dovrebbero essere anche i più astratti.

- **Stable Abstraction Principle (SAP)**

Infine per chiudere il discorso sull'astrazione si arriva alla conclusione che i pacchetti stabili devono essere anche astratti.

Un buon progettista deve tenere conto di questi principi, ma innanzitutto deve sapere a cosa si riferiscono. Esistono molti tool open source o commerciali che mostrano gli indici di stabilità e astrazione, ma è essenziale sapere come vengono calcolati.

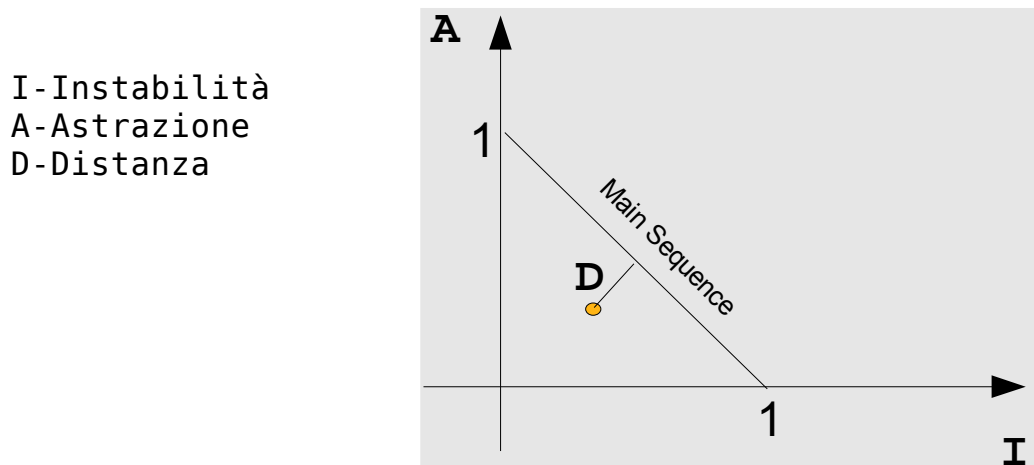
Elenco di seguito alcune metriche relative ai principi sopra citati:

- **Ca** : Afferent Couplings (Accoppiamento Afferente ÷ Classi Afferenti):
Numero di classi esterne al package che dipendono da una classe interna.
- **Ce** : Efferent Couplings (Accoppiamenti Efferente ÷ Classi Efferenti):
Numero delle classi interne al package che dipendono da classi esterne.
- **I** : Instability (Instabilità) [0,1]: $C_e \div (C_a + C_e)$
Rapporto tra la necessità dei componenti esterni e la totalità dei collegamenti. Mostra quindi quanto il package dipenda dall'esterno invece che dall'interno.
0 = STABILE, non dipende da alcuna classe esterna al package
1 = INSTABILE, dipende totalmente dall'esterno
- **A** : (Astrazione): $\text{abstractClasses} \div \text{totalClasses}$
Rapporto tra le classi astratte e quelle concrete, mostra quanto di quel package è astratto. È una metrica imprecisa perché conteggia tra le classi astratte anche quelle che posseggono un solo metodo astratto dei loro N metodi (non fa una proporzione). È comunque una metrica rilevante in quanto dà una idea del coinvolgimento del package.

Infine possiamo definire la distanza di un package dalla sua posizione ottimale, mediante un grafico cartesiano: Instabilità * Astrazione. Misuriamo su questo grafico la perpendicolare tra il valore del package, espresso sugli assi cartesiani, e la retta “Main Sequence”, tracciata tra i punti [0,1] e [1,0]. Il principio SAP ci dice che i pacchetti stabili dovrebbero essere astratti [0,1] e di conseguenza che i pacchetti instabili dovrebbero essere concreti [1,0]. La retta “Main Sequence” mostra tutti i possibili valori che i package dovrebbero assumere in condizioni ottimali.

La distanza da questa retta è quindi un indice di coesione del package. Più la distanza è piccola, più la coesione è alta.

Ecco di seguito il grafico cartesiano:



Nell'elenco ho ommesso volutamente di parlare dei principi del **GRASP**⁴⁴, del **Rasoio di Occam**⁴⁵ (KISS), del principio di **Liskov**⁴⁶ (LSP), del **Design by Contract**⁴⁷ (DBC), dell'**Incapsulamento**⁴⁸ e dell' **Information Hiding**⁴⁹ poiché essi sono essenziali ad ogni sviluppatore junior che si avvicina alla programmazione ad oggetti, mentre i principi precedentemente elencati, sono di carattere diverso. Essi infatti sono legati ad un concetto di modulo e di package e lavorano quindi ad un livello differente, ossia verso uno sviluppo del software in funzione della crescita e dell'evoluzione.

44 GRASP – General Responsibility Assignment Software Principles, principi di programmazione formulati da C. Larman in “Apply UML and Patterns”

45 Rasoio di Occam – Tra due soluzioni bisogna preferire quella che introduce il minor numero di ipotesi e che fa uso del minor numero di concetti. “Keep it Simple, Stupid”

46 Principio di Liskov – Ogni sottoclasse deve essere sostituibile con la classe base.

47 Design by Contract – Data una classe T, ogni classe derivata S deve garantire che le precondizioni di ogni metodo siano uguali o più deboli, e le postcondizioni siano uguali o più forti

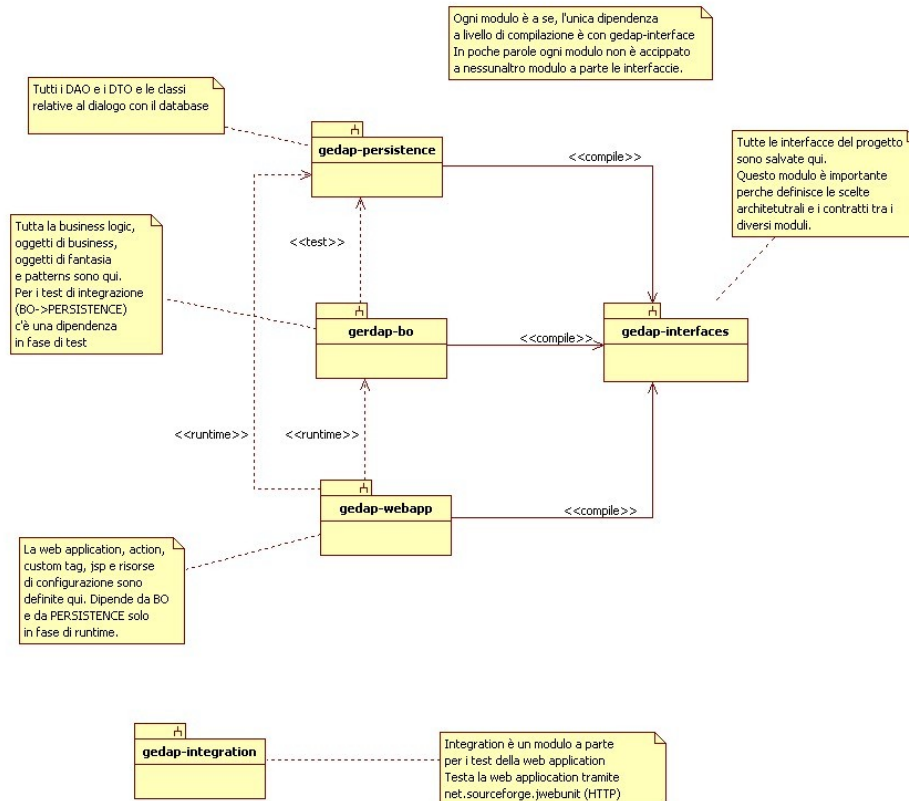
48 Incapsulamento – Raggruppamento in un'unica entità di metodi e attributi realizzato in modo da subordinare la modifica dello stato all'impiego dei metodi di interfaccia fornita dall'incapsulamento

49 Information Hiding – di un oggetto è necessario conoscerne le responsabilità ma bisogna ignorarne la struttura. Le informazioni sullo stato possono essere richieste e/o impostate attraverso metodi specifici.

2. Identificazione dei Moduli del progetto

Quello che segue è l'elenco dei moduli del progetto, per ognuno dei quali indico il nome e la funzionalità.

- | | |
|-------------------------------|---|
| • <i>Gedap Project</i> | <i>Configurazione globale progetto</i> |
| • <i>Gedap Interfaces</i> | <i>Le interface con cui comunicano i moduli</i> |
| • <i>Gedap Persistence</i> | <i>Persistenza su database</i> |
| • <i>Gedap Business Logic</i> | <i>Logica di business</i> |
| • <i>Gedap WebApp</i> | <i>Web application</i> |
| • <i>Gedap Integration</i> | <i>Test di integrazione</i> |

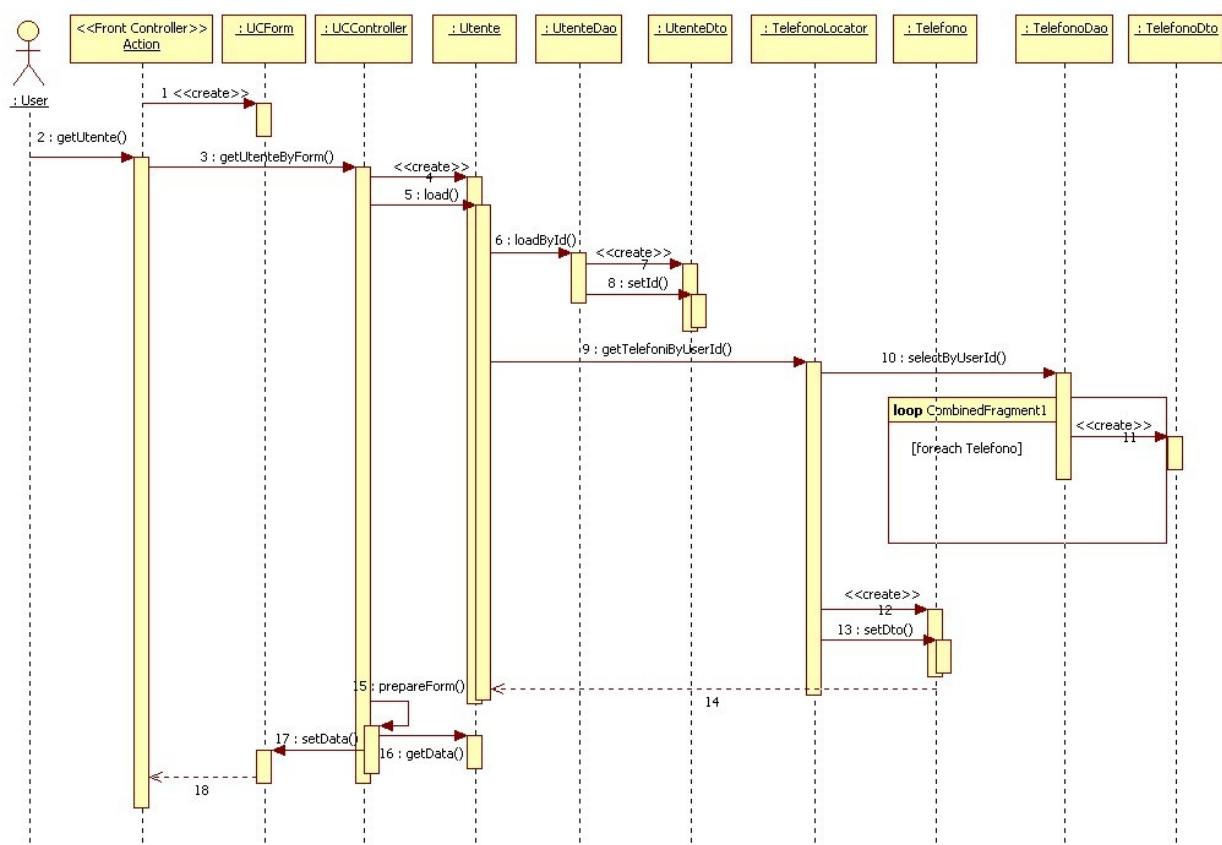


3. Sequences diagrams

Per comprendere l'architettura software che abbiamo adottato, comincerò con mostrare alcuni sequence diagram architetturali. Prima di essi però mostrerò un sequence diagram, scritto quando ancora non avevamo ricevuto il primo caso d'uso. È stato con questo che durante la prima riunione con tutti gli sviluppatori, ho presentato questa architettura.

3.1. Un primo sequence diagram

Il primo sequence diagram scritto, riguarda degli oggetti Utente e Telefono che non fanno parte del dominio poiché all'epoca non era ancora chiaro cosa avremmo dovuto realizzare. Nel sequence si vede come la "Action" (il front-controller) tramite un "UCForm" (Use Case Form) comunica con lo Use Case Controller. Lo UCController è il punto di ingresso con il quale l'utente può interagire per far evolvere lo stato del caso d'uso.



In questo modo, separando in modo netto le azioni della View dal Caso d'uso vero e proprio, si rende il sistema modulare ossia in grado di sostituire o modificare uno dei

componenti, senza cambiare gli altri moduli. Come si può vedere dal diagramma, le richieste dell'**Utente** sono interpretate dal **Front-Controller** (il modulo view), che manda dei messaggi allo Use Case Controller (Il punto di ingresso del caso d'uso) tramite lo Use Case Form (UCForm, un Form di comunicazione, una interfaccia, un contratto tra view e bo).

lo Use Case Controller è il responsabile del business e quindi è l'unico a conoscere come e quali **oggetti di business** (Utente e Telefono) sono coinvolti. Questi ultimi oggetti, Utente e Telefono, sono gli oggetti del modello, e solitamente sono gli unici oggetti di cui si può parlare al cliente.

Ogni oggetto è responsabile di tutto ciò che lo riguarda, fatta eccezione per il salvataggio o il caricamento dal database. Per questo compito è stato applicato il design pattern Data Access Object, quindi le operazioni di salvataggio e reperimento dati dal database sono delegate agli oggetti **UtenteDAO** e **TelefonoDAO**. Come richiesto dal pattern, esistono rispettivamente gli oggetti **UtenteDTO** e **TelefonoDTO** che trasportano i dati tra il modulo della persistenza e quello del business.

L'oggetto **TelefonoLocator** gestisce le collezioni del suo **BO (Telefono)**, in questo caso si occupa del reperimento di tutti i numeri di Telefono per un determinato Utente.

Lo **UCController** poi si occupa di passare tutte le informazioni alla view, memorizzando i dati nello **UFForm**.

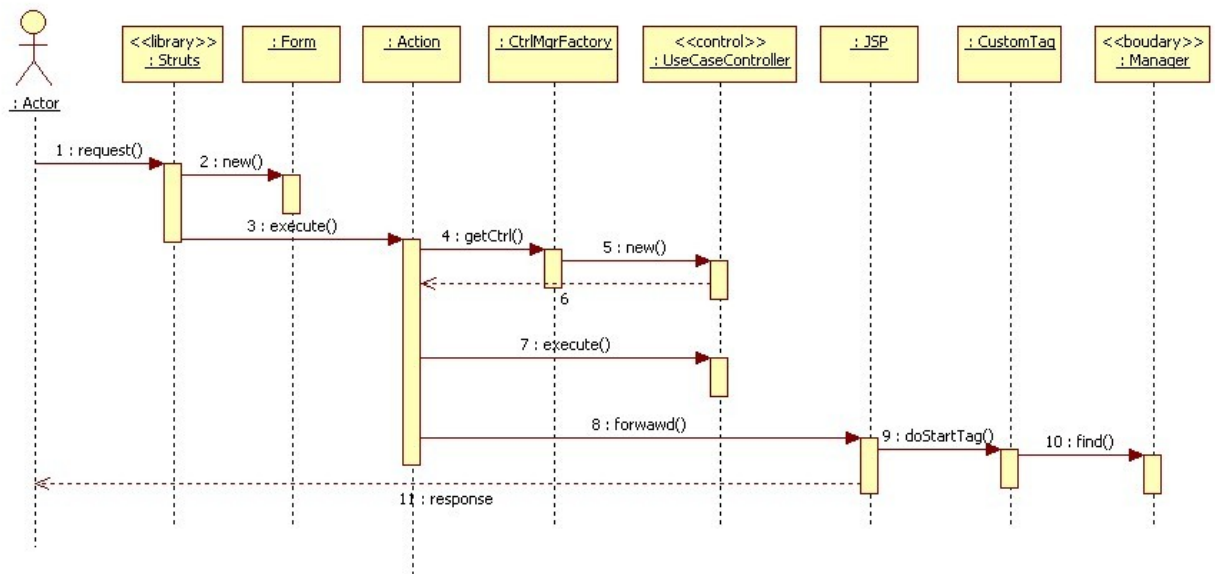
In particolare, possiamo notare che la view è completamente ignara di come funzionano le cose dopo il controller, essa semplicemente legge e scrive informazioni su un form dedicato al caso d'uso.

3.2. **Sequence diagram Webapp**

Questo diagramma mostra il modulo webapp, lato utente e mette in evidenza il sistema di visualizzazione delle pagine web, basato su **Struts**⁵⁰. Inoltre, cosa più importante, fa vedere quali sono i punti di interazione con i moduli di business del progetto. Si nota infatti che l'unico modo per accedere ai moduli che governano gli oggetti di business è quello di passare attraverso due oggetti:

1. **UCController** : per gestire il caso d'uso
2. **Manager**: per ottenere informazioni in sola lettura sulle collezioni di dati

50 Apache Struts - framework open source per lo sviluppo di applicazioni web



L'utente fa una richiesta. **Struts** risponde e in base alla configurazione crea un form (che implementa l'interfaccia **UCForm** relativa al caso d'uso), dopodiché invoca la **Action** relativa (precedentemente nominata front-controller). Il sequence mostra un elemento nuovo. Per ricevere il controller specifico del caso d'uso, la **Action** chiede l'istanza di uno **UController** ad una **Factory**⁵¹ (**CtrlMgrFactory**, o meglio **Factory** dei **Controller** e dei **Manager**). Possiamo anche vedere il modo con cui le **JSP**⁵² ottengono alcuni dati, ad esempio di una ricerca. La **JSP** invoca un **CustomTag**⁵³, che si occupa di ricevere i dati della ricerca usando un **Manager**. Il **Manager** non viene richiesto alla **Factory** perché è un componente inizializzato allo startup della applicazione.

3.3. Sequence Diagram Use Case Controller

Lo **UController** è l'oggetto che disaccoppia il modulo di business dalla webapp fornendo i metodi necessari per la gestione del caso d'uso. Per gestire un caso d'uso, si avrà la necessità di scambiarsi dei dati e questi viaggiano tramite uno o più form di comunicazione.

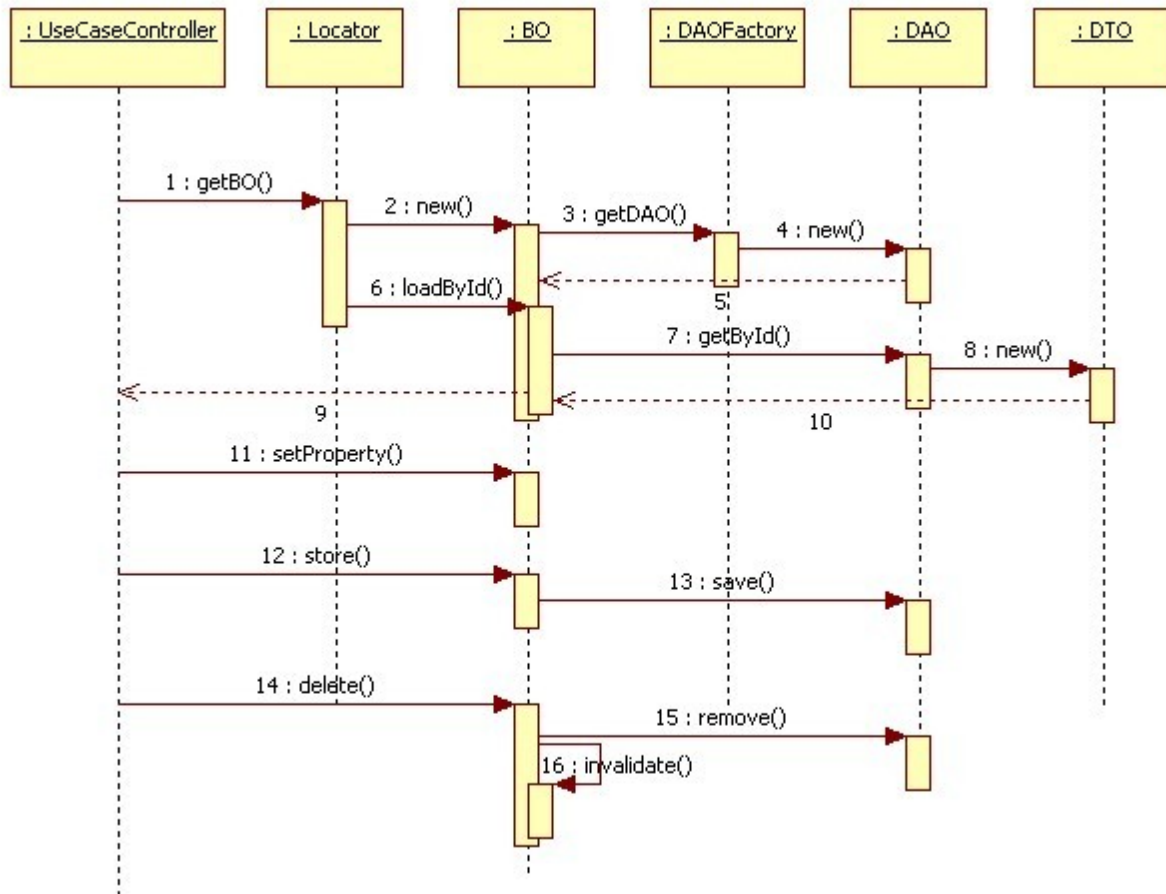
Esiste un aspetto transazionale del controller, di cui parlerò più avanti, il quale garantisce che tutti i metodi di scrittura (insert, delete e update) invocati nel modulo persistenza siano fatti in un contesto transazionale.

51 Factory Method Pattern – definisce un'interfaccia **Creator** per ottenere una nuova istanza di un oggetto delegando ad una classe la scelta di quale classe concreta istanziare

52 **JSP** – Java Server Page, servlet dinamica contenente un mix tra tag HTML, **CustomTag** e scriptlet (codice java)

53 **Custom Tag** – estensione del **JSP** che permette di ridurre gli scriptlet nelle **JSP** occupandosi loro di alcuni specifici task.

Il controller svolge il ruolo di **Façade**⁵⁴ per l'accesso al modulo di Business logic e quindi fa una vera e propria separazione tra i due moduli. A partire da questo componente si scatenano, una dopo l'altra, le azioni tra gli oggetti del modello. In questo sequence vediamo quali sono i componenti coinvolti fino allo strato di persistenza, anche qui come prima tra i moduli c'è una Factory che fornisce le implementazioni dei DAO.



I passi dall'uno al dieci mostrano come il controller ottiene l'oggetto di business, il quale carica il suo stato invocando un metodo nel rispettivo DAO (ottenuto dalla DAOFactory).

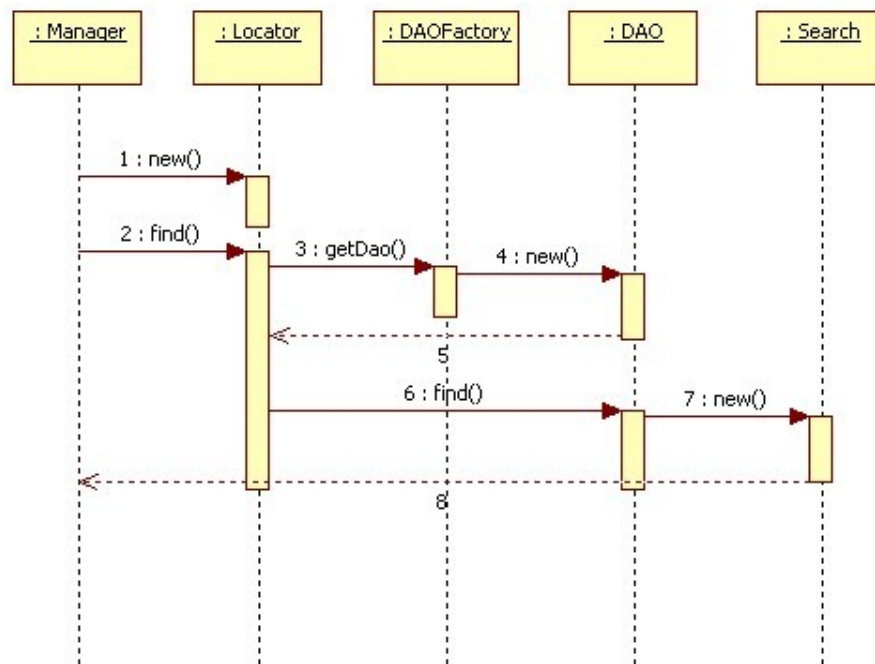
I passi dall'undici al tredici mostrano come è possibile interagire con l'oggetto BO, salvandolo o cancellandolo. Esso chiaramente, a seconda se viene cancellato o salvato, invocherà il metodo relativo del proprio DAO.

⁵⁴ Façade Pattern - Letteralmente significa "facciata", indica un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse.

3.4. Sequence Diagram Manager

Il Manager è un oggetto **Singleton**⁵⁵, specializzato per la ricerca di collezioni di oggetti di business. Le collection restituite contengono DTO light chiamati **Search**.

Il Manager è il <<**Boundary**⁵⁶>> a disposizione del front-end e va usato per la ricerca di collection di dati; ovviamente trattandosi di dati in sola lettura ogni Manager dispone di un aspetto, simile a quello dei controller, che gli fornisce un contesto transazionale in sola lettura.



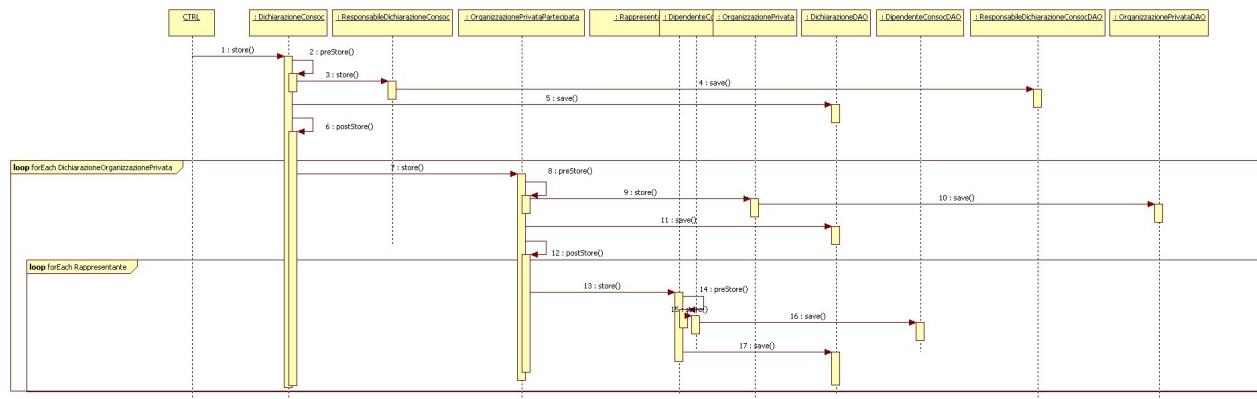
I passi da 1 a 8 mostrano come il Manager si occupa di reperire una collezione di dati. Notiamo che gli elementi restituiti non sono dei DTO, ma sono degli oggetti più leggeri, chiamati Search object. Questi sono gli unici oggetti che attraversano tutti e tre i moduli dell'applicazione.

55 Singleton Pattern – permette di assicurare che una classe abbia una e una sola istanza e che questa sia globalmente accessibile.

56 Boundary – Oggetto di confine, servizio usato per diversi scopi e da utenti differenti

3.5. Sequence Store (Template Method⁵⁷)

In questo sequence viene alla luce la relazione tra gli oggetti di business. Ad esempio nel salvataggio di un oggetto “Dichiarazione Consoc”, gli oggetti di business contenuti nella dichiarazione vengono a loro volta salvati. Ogni oggetto quindi è responsabile di chiedere agli oggetti aggregati di provvedere al proprio salvataggio.



Il Controller invoca il metodo “store” sulla “Dichiarazione Consoc”, questa si occupa a sua volta di invocare i metodi “store” sui suoi oggetti aggregati, scrivendo le chiamate nei metodi astratti “preStore” e “postStore”. Se gli oggetti sono nuovi, non posseggono un database ID, perché questo verrà generato in modo univoco al momento del suo salvataggio. Dipende quindi dal verso della relazione il fatto che un oggetto venga salvato nella “preStore” o nella “postStore”. In questo caso il Responsabile non conosce direttamente la Dichiarazione, infatti potrebbe essere il Responsabile di più Dichiarazioni. Mentre la Dichiarazione di una partecipazione su una “Organizzazioni Privata” (Consorzio o Società) è strettamente legata a questa sola Dichiarazione, quindi viene salvata successivamente all’inserimento nel database della Dichiarazione, in quanto ha bisogno dell’ID della Dichiarazione padre.

Ricordo a questo punto, che tutto il sistema è sotto una transazione e in caso di un qualsiasi errore, sarà l’aspetto transazionale del controller a occuparsi di catturare l’eccezione e invocare la **rollback**⁵⁸ sulla connessione.

57 Template Method Pattern – definisce lo scheletro di un algoritmo e delega ad una classe derivata la possibilità di ridefinire alcuni passi (es. preStore, postStore) dell’algoritmo senza bisogno di modificare l’intero algoritmo o la sua struttura.

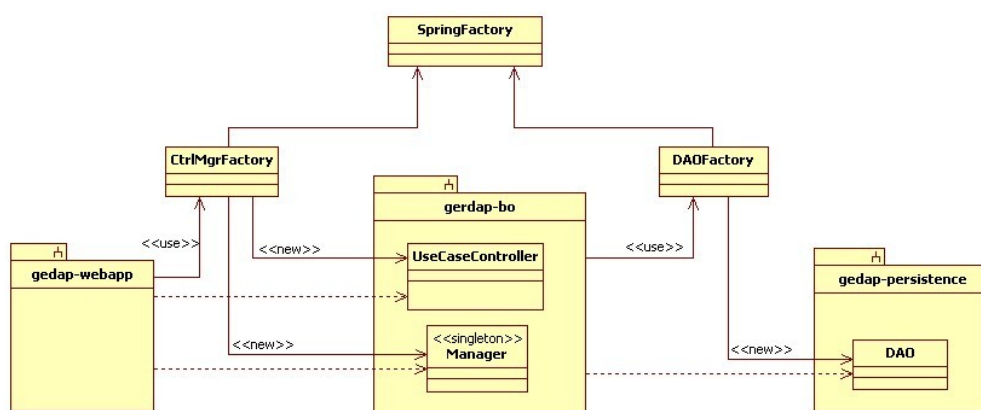
58 Rollback – è uno statement SQL che cancella tutte le modifiche apportate al database durante la stessa transazione.

4. Component Diagram

Attraverso i **component diagram** vengono descritti i componenti architetturali del software.

4.1. Factory Component Diagram

Nel seguente **component diagram**, si evidenzia come i moduli parlano tra di loro tramite delle **Factory**. In alcuni file di configurazione, vengono mappati i nomi delle interfacce dei Controller, dei Manager e dei DAO con le loro implementazioni. Le Factory, leggendo dai file di configurazione, sono in grado di collegare ad una richiesta da parte di un modulo l'implementazione dell'oggetto richiesto.



Il modulo “gedap-webapp”, per accedere ad un caso d'uso, chiederà alla **CtrlMgrFactory** l'implementazione del proprio controller (conosce solo l'interfaccia) e allo stesso modo, il modulo “gedap-bo” chiederà l'implementazione dei propri DAO alla **DAOFactory**.

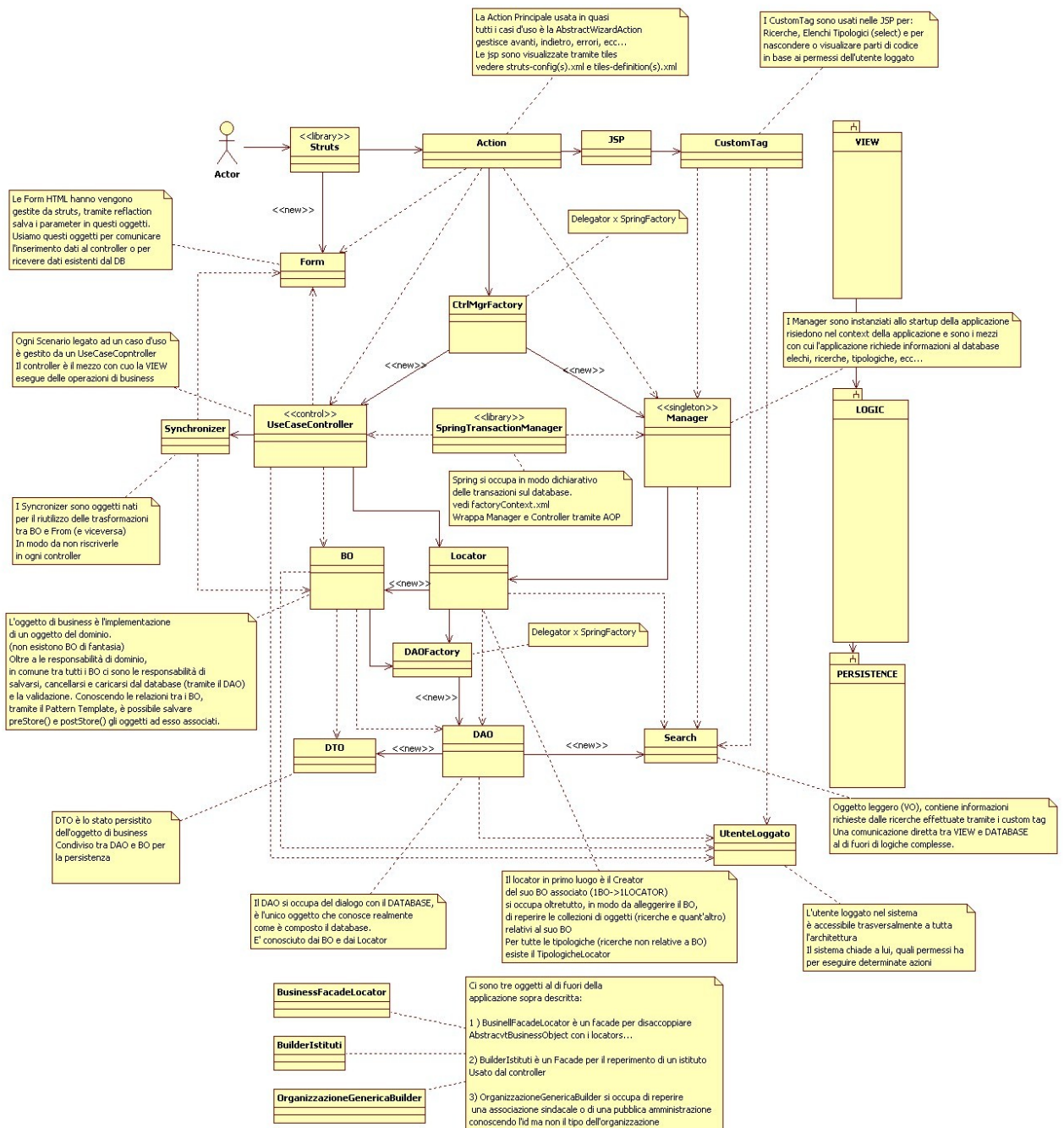
È grazie a queste Factory che è possibile disaccoppiare i moduli e far sì che possano compilare autonomamente.

Come si può notare entrambe le Factory, **DAOFactory** e **CtrlMgrFactory**, usano la **SpringFactory**. Questo oggetto è stato aggiunto in una fase successiva, contemporaneamente all'introduzione del framework **Spring**⁵⁹, di cui parlerò più avanti.

⁵⁹ Spring – IoC Container opensource sviluppato da SpringSource.

4.2. Architectural Component Diagram

Mostro qui di seguito il diagramma architeturale di tutti i componenti del progetto.



4.3. Componenti Coinvolti

- **View**

La parte di view è composta dalle JSP e dalle classi di Struts. Governano la navigazione e la visualizzazione lato web.

- **Form**

E' il Value Object che contiene i dati inseriti dall'utente e attraverso i quali il modulo View comunica con il modulo di Business Logic.

- **UCController**

Il *Controller* è la classe dalla quale s'invocano le azioni descritte nel singolo caso d'uso. Per questo motivo esiste una classe *Controller* per ogni caso d'uso implementato.

- **BO**

I *Business Object* sono le classi che rappresentano l'entità reale del sistema. Nei *Business Object* sono rappresentati i comportamenti di business delle singole entità. Sono loro a comunicare con gli oggetti della persistenza (i DAO). La comunicazione avviene attraverso due tipi di Value Object: *DTO*, quando il modulo di persistenza restituisce l'intero stato dell'oggetto, *Search*, quando la persistenza deve restituire ai *Business Object* oggetti leggeri che servono alle funzioni di ricerca.

- **BOLocator**

Esiste un *Locator* per ogni *Business Object*. Essi sono i **Creator**⁶⁰ dei *Business Object* e si occupano di gestire e reperire le loro collezioni.

60 Pattern Creator – Il Creator è il principio per il quale si assegna la responsabilità di creazione di un oggetto

5. *Scelte di progettazione*

5.1. **Riguardo i Frameworks**

I framework devono essere un sostegno allo sviluppo e non un peso. Ritengo che una azienda, che ne abbia i mezzi e le possibilità, dovrebbe scrivere i propri framework da sola. Vincolare infatti una parte del proprio software a terze parti (coloro che scrivono i framework) è pericoloso, soprattutto se il progetto è a lungo termine, ossia se il software deve essere scritto e soprattutto mantenuto da una azienda per diversi anni. Può capitare infatti che progetti/framework attuali vengano negli anni abbandonati o che subiscano forti variazioni. L'azienda che li sta adoperando potrebbe così essere costretta a rifare il lavoro per adattarsi o a rimanere con un framework non più mantenuto. D'altro canto, troppo spesso, le **PMI**⁶¹ per accorciare i tempi di sviluppo e restare competitive sul mercato, hanno la necessità di appoggiarsi a framework esterni.

È necessario quindi porre molta attenzione nella loro scelta, ci sono molte decisioni durante lo sviluppo del software che possono essere modificate in corso d'opera, ma introdurre il framework sbagliato all'inizio di un progetto, potrebbe essere disastroso. Rimuovere qualcosa di troppo è più difficile che introdurre qualcosa successivamente.

Prima di scegliere un framework è importante porsi alcune domande:

- *Perché usare un framework?*
- *Cosa risolve?*
- *Quali pattern implementa?*
- *Dove sono i vantaggi?*
- *Quanto tempo impiego per imparare ad usarlo?*

Inoltre è da considerare che una architettura troppo complessa, necessita di un team altamente specializzato. La ricerca della semplicità e della chiarezza devono sempre essere alla base di un disegno architettonico. Introdurre un framework implica che ogni membro del team deve conoscerlo. Introdurre contemporaneamente più di un framework nuovo, non facilita quindi le cose.

61 PMI – Piccola Media Impresa

Esistono due classici Antipattern che potrebbero riguardare i framework:

- **Golden Hammer**
Martello d'oro: consiste nel ritenere che un tool/framework sia la soluzione a tutti i problemi e nell'inserirlo di conseguenza in ogni progetto, senza però verificarne con precisione il contesto.
- **Gas Factory**
L'abuso di frameworks può trasformarsi in una gas factory, ossia in una applicazione che ha la possibilità di fare anche il caffè, senza però che ce ne sia una vera esigenza. Il che allunga inutilmente e a dismisura i tempi di sviluppo.

5.2. I Framework di Gedap

Nel team di sviluppo ci furono pareri discordanti in merito all'adozione di alcuni frameworks. In particolare, si discusse molto sulla possibilità di utilizzare **Spring** per le sue capacità di **IoC container**⁶², ed anche sul possibile uso degli **EJB Entity**⁶³ o di un **ORM**⁶⁴ per il mapping degli oggetti al database (**Hibernate**⁶⁵ o **Ibatis**⁶⁶).

Spring inizialmente non fu utilizzato (fu introdotto dopo sei mesi), la persistenza fu delegata alla semplice implementazione del design pattern **DAO**⁶⁷ (usando **JDBC**⁶⁸) e per la view fu utilizzato **Struts**.

Successivamente, durante i due anni di sviluppo, furono inseriti **Quartz**⁶⁹ per lo scheduling di alcuni jobs asincroni, **Itex**⁷⁰ per la creazione di PDF, **Velocity**⁷¹ per i template delle email e **Terracotta DSO**⁷² per il clustering delle Sessioni HTTP.

5.3. Spring: Container IoC

Il progetto è iniziato con i soli strumenti conosciuti bene dagli sviluppatori, per questo motivo **Spring** è stato introdotto dopo sei mesi di sviluppo. La strategia era di far sentire tutti i membri del team a proprio agio, inserendo i framework solo quando se ne identificava la necessità. **Spring** è stato inserito per gestire in modo dichiarativo le

62 IoC Container – Container che si occupa, mediante il principio dell'inversion of control, delle relazioni tra gli oggetti

63 EJB Entity – Enterprise Java Bean destinato alla persistenza di un certo dato su un DB

64 ORM - Object-relational mapping

65 Hibernate – Framework ORM open source, per il mapping tra oggetti e tabelle

66 Ibatis - Framework ORM open source, per il mapping tra oggetti e tabelle

67 Pattern DAO – Data Access Object,

68 JDBC – Java Database Connectivity, specifiche java per le connessioni al database.

69 Quartz – Scheduler open source

70 Itex – Libreria open source per la generazione dei PDF

71 Apache Velocity – Template engine open source

72 Terracotta DSO – Software open source, per il clustering

transazioni, infatti in questa architettura così modulare, la persistenza era l'unico modulo a conoscere il database, e quindi l'unico punto nell'applicazione dove gestire le transazioni a livello applicativo. Era però necessario che la transazione fosse aperta a partire dalle operazioni effettuate dallo UController, ossia nel modulo di business. **Spring** ha la capacità di dichiarare le transazioni in un qualsiasi oggetto, anche se non direttamente accoppiato al database, infatti permette di configurare degli aspetti **AOP**⁷³ intorno agli oggetti. Un aspetto lo si può vedere come un comportamento aggiuntivo iniettato ad un oggetto in fase di caricamento (runtime).

L'inserimento di **Spring** è stato rapido. Inizialmente erano state implementate la CtrlMgrFactory e la DAOFactory come unici punti di creazione di Controller, Manager e DAO. Il refactoring ha semplicemente trasformato le Factory in semplici **Façade** verso la SpringFactory contenente tutti questi oggetti.

La configurazione delle transazioni è stata dichiarativa, da un file XML abbiamo configurato i metodi "execute" dei Controller rendendoli transazionali e chiedendo a Spring di invocare la rollback in caso di eccezione.

Le modifiche architetturali, hanno sempre un grande impatto sul sistema, soprattutto se fatte in componenti critici dell'applicazione. I test di non regressione però hanno reso immediate e sicure tutte le modifiche apportate.

Introducendo Spring abbiamo avuto la **possibilità** di utilizzare, oltre alla gestione dichiarativa delle transazioni, tutte le potenzialità di questo framework. Cerchiamo quindi innanzitutto di comprendere cosa è un **IoC Container**.

L'**Inversion of Control** è il principio per il quale è possibile descrivere come gli oggetti devono essere valorizzati e quali sono le loro dipendenze. Fondamentale in questo principio è l'esistenza di un container, che è il responsabile del collegamento fra questi oggetti e ne determina le dipendenze.

Abbiamo così potuto delegare al Container Spring, alcune relazioni tra gli oggetti come ad esempio il **datasource**⁷⁴ o l'utente autenticato nel sistema, senza il bisogno di scrivere della logica di reperimento o di lookup, ma semplicemente scrivendo queste relazioni nei file di configurazione.

73 AOP – Aspect Oriented Programming

74 Datasource – Oggetto della specifica JDBC responsabile del reperimento delle connessioni al database.

5.4. **Hibernate: ORM**

Hibernate è uno dei framework che inizialmente si era pensato di adoperare. La peculiarità degli **ORM** (Object-relational mapping) è mappare gli oggetti con le tabelle, esistono infatti delle procedure chiamate **java2ddl** che creano e/o aggiornano lo schema del database a partire da un modello ad oggetti. Purtroppo non è altrettanto semplice fare il contrario, poiché la velocità con cui cambia il database non è la stessa con cui cambia il modello.

La scelta in questo caso è stata di non usare framework, si è invece deciso di creare per ogni oggetto del modello un DAO che ne gestisse la persistenza. Ad ogni DAO non è legata necessariamente una sola tabella, esistono infatti, per via della normalizzazione, informazioni dello stesso oggetto divise fra più tabelle.

La progettazione dei database negli ultimi anni è radicalmente cambiata. Oggi si ha molto più spazio disponibile e una grande velocità di accesso ai dati. Le esigenze che portavano ad ottimizzare e quindi minimizzare lo spazio usato, hanno perso di importanza. Ora nella progettazione dei database, la duplicazione è ammessa in forma "lieve" e non è più necessario applicare la terza forma normale. L'esigenza principale è come sempre la comunicazione, l'accesso semplice alle informazioni, riducendo al minimo le join. Sempre di più, chi scrive software si concentra sull'architettura, sul codice e delega a un ORM la creazione automatica del database. Quando il progetto è arrivato ad una certa stabilità, si può allora cominciare a esaminare le performance e a ottimizzare le tabelle in funzione dell'applicazione.

Nel nostro caso, partendo da una migrazione, non è stato possibile questo approccio, il che ha escluso Hibernate da un possibile utilizzo.

5.5. **EJB Entity**

La best practise JEE, fino a pochi anni fa, consigliava l'uso degli EJB per lo strato di persistenza. Un requisito per usare gli EJB è avere un EJB container, o meglio un Application Server JEE Compliant. Nel nostro caso abbiamo scelto di tenere l'architettura più leggera, senza EJB, quindi senza il bisogno di un EJB Container. L'applicazione funziona tranquillamente su un qualsiasi Servlet Engine, nel nostro caso Tomcat in esercizio e Jetty in sviluppo.

La possibilità di utilizzare un domani gli EJB, nel caso l'applicazione cresca nel tempo, non

è stata comunque preclusa. Ogni DAO dell'architettura viene usato attraverso una interfaccia e reperito tramite una factory. Questo disaccoppiamento tra il “cosa” e il “come” garantisce la sostituibilità delle implementazioni. Un domani, se se ne avrà l'esigenza, la DAOFactory potrebbe diventare un Service Locator che reperisce la Home Interface dell'ipotetico EjbDao.

5.6. Controller Transazionale e lo stato dello Use Case

L'idea iniziale era che lo Use Case Controller, oltre ad essere un punto di ingresso per le azioni dell'utente verso il sistema, mantenesse al suo interno lo stato del caso d'uso e che venisse creato quindi da un utente al momento del bisogno e restasse in sessione fino al termine del caso d'uso.

Aggiungendo l'aspetto (AOP) transazionale al controller (tramite Spring), sono entrati in gioco dei componenti non Serializzabili, rendendo così impossibile replicare la Sessione per i componenti del cluster.

Questa esigenza ha portato alla luce le differenze tra lo Use Case Controller e lo stato del caso d'uso. È stata usata quindi un'altra feature di Spring: lo scope “Session”.

Lo Use Case Controller è diventato Singleton, quindi semplicemente un servizio e questo ha implicato due cambiamenti importanti:

- un nuovo oggetto “Memory” in sessione (gli ex-attributi del controller)
- una differente gestione della sicurezza

Inizialmente poteva esistere un solo controller in sessione quindi, cambiando caso d'uso, veniva sostituito il controller e così alleggerita la memoria di tutti gli oggetti salvati nel caso d'uso precedente.

Con i controller “singleton” la gestione dello stato del caso d'uso è stata delegata ad un oggetto “Memory” (scope session).

Come funziona lo scope session? Tramite un Filtro JEE, Spring aggancia gli oggetti della sessione utente, attraverso dei proxy AOP. Quindi, in base al thread che attraversa l'applicazione, Spring è in grado di agganciare il corretto oggetto della sessione.

La sicurezza di un caso d'uso precedentemente era affidata alla Factory dei controller, che controllava i permessi dell'utente prima di istanziarlo. Nella attuale versione rifattorizzata, non c'è l'istanziamento di un nuovo controller, infatti esso è stato trasformato in un servizio,

un singleton. Il controllo della sicurezza quindi è stato aggiunto tramite un Aspetto AOP. Ogni qualvolta viene invocato un metodo su un Controller, viene a sua volta invocato il metodo “checkPermission” sull'Aspetto. Quest'ultimo lancia una eventuale eccezione “IllegalGedapAccessException” nel caso l'utente non abbia i permessi per accedere al caso d'uso.

5.7. Clustering with Terracotta.

L'applicazione è stata rifattorizzata in modo che potesse funzionare su un cluster per la replicazione delle sessioni. Fondamentale è stato il controllo degli oggetti che finivano in sessione, in modo da renderli più semplici e atomici possibile.

Per la replicazione delle sessioni HTTP, la scelta è caduta su Terracotta DSO, in primo luogo perché è un software che non ha alcun impatto sul codice (niente API), inoltre è possibile usare questo strumento su un qualsiasi Application Server o su una qualsiasi JVM. In futuro potrebbe essere anche possibile utilizzarlo per avere una cache condivisa tra la web application e il job scheduler quartz (due processi java distinti).

Come valore aggiunto Terracotta, essendo un prodotto con un alto contenuto innovativo e tecnologico, poteva anche portare una conoscenza nuova all'interno del team.

Per rifattorizzare una applicazione verso Terracotta DSO serve la consapevolezza che questa applicazione può essere eseguita in un ambiente multi-threads. Si è trattato quindi di individuare le parti dell'applicazione critiche e condivise per renderle sincronizzate.

Ritengo che questo framework sia decisamente innovativo, per questo vorrei dedicare un piccolo capitolo alla sua esposizione.

5.7.1. Cosa è Terracotta DSO

Terracotta DSO è una soluzione di un “cluster trasparente” che, senza componenti aggiuntivi o nuove API java, fornisce una infrastruttura in grado di far girare una applicazione java (JSE) su un cluster di JVM e quindi fornisce già di suo un servizio Enterprise **Scalabile**⁷⁵ e **Affidabile**⁷⁶.

Le librerie DSO (*Distributed Shared Objects*), lette a boot-time dai clients, servono a dare in modo trasparente un comportamento di cluster a livello di JVM. Prima di tutto

⁷⁵ Scalabilità – ossia le capacità di un sistema di “crescere” o “decretere” in funzione delle necessità e delle disponibilità

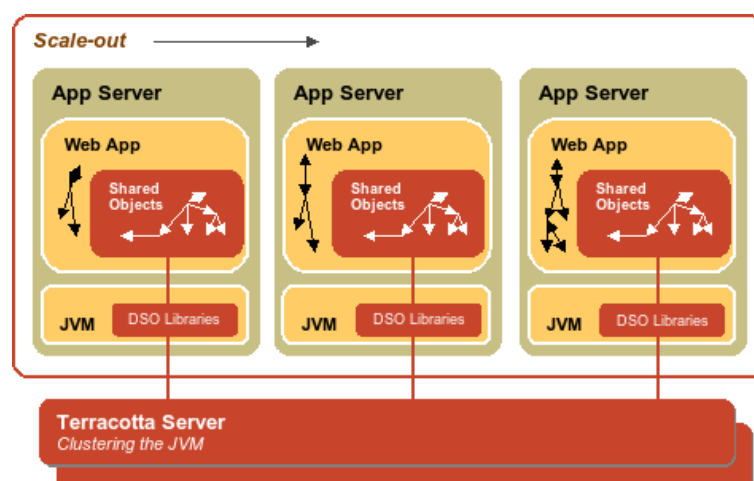
⁷⁶ Affidabilità – ossia la capacità di rispettare le specifiche di funzionamento nel tempo e inoltre la capacità di un sistema di non subire fallimenti anche in presenza di guasti (fault-tolerance)

aggiungono dei comportamenti distribuiti alle operazioni di salvataggio e cancellazione degli oggetti dell'HEAP, in secondo luogo forniscono uno scambio di informazioni (SIGNALS) centralizzato per avere una sincronizzazione multi JVM.

In una JVM ogni oggetto è salvato in memoria tramite le istruzioni bytecode PUTFIELD e GETFIELD. Terracotta modifica queste istruzioni a runtime aggiungendo alle classi "configurate" dei comportamenti condivisi. Grazie a queste istruzioni, terracotta è in grado di tenere replicati sui server del cluster, interi o parziali grafi di oggetti, comunicando le modifiche degli oggetti e le richieste di lock e unlock al server centrale terracotta.

Questo è l'elenco delle istruzioni bytecode modificate dalle librerie DSO: **GETFIELD**, **PUTFIELD**, **AASTORE**, **AALOAD**, **MONITORENTRY**, **MONITOREXIT**, **INVOKE0** e **INVOKESPECIAL**

Ogni JVM ha un **HEAP**⁷⁷ dove risiedono gli oggetti. Ogni cambiamento ad un oggetto condiviso lo possiamo vedere come una transazione terracotta, questa contiene i cambiamenti degli oggetti, o meglio solo i dati dei campi cambiati. Queste transazioni sono inviate al server e questo, immediatamente, invia i cambiamenti a tutte le altre JVM coinvolte, lasciando quindi il cluster consistente. Il server filtra inoltre i dati delle transazioni in modo da mandare i cambiamenti dei soli oggetti posseduti dalle singole JVM. Ossia se una JVM conosce uno solo degli oggetti coinvolti nella transazione, il server gli manderà i soli dati relativi a questo oggetto.



77 HEAP – Spazio di memoria del processo java nella quale la JVM scrive definizioni e oggetti java

Terracotta non usa la Java Serialization per replicare i cambiamenti, gli oggetti sono tracciati e replicati a livello di campo e le transazioni contengono solo dei frammenti di un oggetto anziché tutto il grafo degli oggetti.

La Java Serialization ha come principio il salvataggio di un oggetto su uno stream, quindi anche un cambiamento ad un singolo valore, provoca la serializzazione di tutto il grafo degli oggetti coinvolti a partire dalla radice.

L'approccio di terracotta è chiaramente molto più efficiente della Java Serialization, poiché, per mantenere sincronizzati due oggetti, sposta attraverso il cluster solo i dati che sono cambiati, anziché ogni volta replicare tutto il grafo. Le differenze di performance aumentano a dismisura se si pensa ad una struttura condivisa molto profonda.

Comunque questo approccio, a parte l'efficienza, risolve anche uno dei grandi problemi del clustering: **preservare l'identità di un oggetto.**

Se la serializzazione è usata per spostare un oggetto tra i componenti di un cluster, un oggetto cambiato dovrà quindi essere deserializzato nella JVM remota e "qualcosa" dovrà sostituire l'oggetto esistente con quello passato. Questo è il motivo per cui molte altre tecniche di clustering/caching necessitano di esplicite API (GET/PUT) per la replicazione di un oggetto.

Terracotta non ha queste restrizioni. Un oggetto condiviso nel cluster, vive nell'HEAP come qualsiasi altro oggetto. Quando un cambiamento ad un oggetto è fatto nella JVM locale, i cambiamenti sono effettuati direttamente sull'oggetto dell'HEAP. Quando i cambiamenti sono fatti da una JVM remota, la transazione è ricevuta dalla JVM locale che applica questi cambiamenti sull'oggetto locale (che risiede nell'HEAP). Questo significa che c'è una e solo una istanza di un oggetto condiviso nell'HEAP di ogni JVM.

Con TC non bisogna mai richiedere la copia "fresca" dell'oggetto, e nemmeno l'inverso, ossia non è necessario informare nessuno che l'oggetto è stato modificato. Questo perché non ci sono copie. Un oggetto condiviso nel cluster è solo un semplice oggetto nell'HEAP e un oggetto condiviso si comporta come ogni altro oggetto: ogni cambiamento effettuato è disponibile immediatamente su tutti gli oggetti che posseggono la reference ad esso, che siano locali alla JVM o remoti.

Quindi è importante capire, che un oggetto conserva la propria identità in una applicazione eseguita su un cluster TC (multi-JVM), esattamente come se l'applicazione fosse eseguita in una singola JVM.

La semplicità di Terracotta e la conservazione dell'identità di un oggetto nel cluster, rendono i problemi del cluster completamente separati dai problemi di design di una applicazione. Le capacità del cluster sono inserite a livello di JVM, fondendosi quindi con l'infrastruttura.

Così come il garbage collection permette la gestione della memoria in modo trasparente alla programmazione, Terracotta permette la clusterizzazione in modo ugualmente trasparente.

Altre attività eseguite

L'attività, le scelte e le considerazioni fatte in questa tesi sono solo una minima parte del grande lavoro, svolto giorno per giorno, nel progetto Gedap. Ho voluto approfondire alcuni aspetti del lavoro, mentre ho appena accennato ad altri. Prima di concludere però, vorrei parlare di due punti critici dell'applicazione che finora ho lasciato nell'ombra.

1. *View, Casi d'uso e la navigazione accessibile*

Un sito per una pubblica amministrazione deve seguire la *legge sull'accessibilità*, Legge 9 gennaio 2004, n. 4 conosciuta sotto il nome di “**Legge Stanca**”⁷⁸. Questa legge è stata creata per garantire pari diritti nell'accesso alle informazioni. Un sito pubblico deve essere accessibile ad ogni persona, compresi i disabili: è quindi importante che si possa usare con **browser speciali**⁷⁹ o con un classico browser che abbia alcune funzioni disabilitate (css e/o javascript).

Gedap rientrava chiaramente tra le applicazioni web che devono essere accessibili.

Uno dei punti fondamentali per garantire l'accessibilità delle informazioni è l'utilizzo attento dell'HTML, in modo che le informazioni vengano presentate in maniera sequenziale (si pensi ad esempio ad un utente che usa un browser braille o un lettore vocale). L'HTML quindi non deve avere nulla a che fare con l'impaginazione grafica. Per questo motivo la grafica di Gedap è stata completamente impostata su css2.

La scrittura dell'HTML è stata inoltre influenzata anche in altre maniere. Ad esempio è stato necessario esplicitare su tutte le immagini l'attributo ALT in modo significativo e anche definire precisamente ogni LABEL, con la giusta ACCESS KEY, su tutti i campi delle form per l'inserimento dei dati. Per questo abbiamo creato dei custom tag ad-hoc che gestissero alcuni componenti chiave dell'HTML. L'impossibilità di usare javascript ha reso le cose più elementari, ma non sempre le cose elementari sono le più semplici.

78 Legge Stanca - Dal nome del Ministro per l'innovazione e le tecnologie che ne ha promosso la stesura

79 Cfr nota n°9

2. *Lavori Asincroni*

Un modulo nuovo è nato prima nella messa in produzione del progetto: gedap-scheduler. Eseguendo dei test di performance, ci siamo resi conto delle grandi risorse impiegate nella generazione dei report PDF e nell'invio delle mail. Per questo il motore di job scheduling quartz, nato all'interno della web application, è stato spostato in una applicazione esterna, una JVM esterna.

Tutto ciò che riguarda calcoli, elaborazioni ed invio di documenti o mail, ossia tutti i lavori che non rispondono in modo sincrono ad una richiesta dell'utente, sono stati spostati in questo modulo. Allo stesso modo del modulo "gedap-webapp", la comunicazione con il business avviene tramite il modulo "gedap-bo". Ad ogni lavoro corrisponde un oggetto Task che si occupa di interagire con il business. Le transazioni in questo caso partono da questi oggetti, utilizzando sempre la stessa strategia impiegata nella webapp, ossia in modo dichiarativo con aspetti AOP. Avendo un motore che si occupa di eseguire lavori asincroni, la web application risponde sempre in tempi brevissimi e non si deve occupare dei lavori impegnativi.

Conclusioni

Disegnare una architettura in grado di evolvere e crescere nel tempo non è poi così semplice. Nel corso di questi due anni di lavoro, scelte che sembravano inizialmente affidabili si sono rivelate meno solide del previsto, mentre altre si sono delineate solo con il passare del tempo.

Credo che i requisiti fondamentali per diventare un buon architetto siano sia le competenze tecniche e la capacità di prevedere i possibili cambiamenti ma anche una buona dose di autocritica. È necessario essere sempre con i piedi per terra, aderenti alla realtà e quello che ci succede intorno. Essere aperti e elastici ad ogni opinione, seguire le scelte del team, non spingere le proprie idee contro mulini a vento e cercare di valorizzare il lavoro di tutti, responsabilizzando e delegando quando possibile.

La condivisione delle conoscenze è essenziale in ogni lavoro, ma nel nostro ambito ha un significato molto profondo. Le conoscenze attuali sono frutto del lavoro di colleghi più o meno vicini fisicamente, persone di questi ultimi 50 anni. Fisica, Chimica, Elettronica, Matematica, Filosofia e Linguistica sono e restano parte integrante del nostro lavoro, ed esse trovano espressione nelle ricerche di milioni di persone che stanno collaborando in modo più o meno cosciente verso una direzione comune.

Più crescono le mie conoscenze tecnologiche, e più sento un distacco dal reale. Per questo sento continuamente una spinta contraria, che rema verso la condivisione, verso la collaborazione, verso l'umanità.

Nel nostro lavoro è importante non perdere di vista questo: i colleghi, i clienti, gli insegnanti e gli studenti sono "reali" ed è con tali persone che bisogna relazionarsi.

Ringraziamenti

Questo progetto è stato frutto del lavoro di un team con il quale ho discusso e condiviso posizioni e scelte. Mi sembra quindi importante riconoscere capacità e meriti ad ogni persona e per questo non posso fare a meno di ringraziare tutti i miei colleghi, dai senior più esperti, che mi hanno suggerito e guidato in questo percorso, fino agli junior che pieni di idee, entusiasmo e volontà hanno portato avanti questo progetto.

Un grazie quindi (ordine preso dal wiki di progetto)

crapa pelada, manu, boromir, tano, gigio, fusco, mt, repentino, mirkino, flore, o direttore, l'avvocato, sheshi, turano, rambellaccio, alfio e mestichelli

Bibliografia

- Beck, K. (1999), **Extreme programming explained: Embrace Change**, Boston , Addison-Wesley
- Beck, K. (2008), **Implementation patterns**, Upper Saddle River N.J. , Addison-Wesley
- Brown, W. J. (1998), **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**, Wiley
- Erich G. [et al.] (1995), **Design patterns : elements of reusable object-oriented software**, Boston, Addison-Wesley
- Federici, S. e Vinci G. (2006), **AntiPattern: se lo conosci lo eviti**,
<http://www.aldaran.org/press/java/2006/01/25/antipattern-se-lo-conosci-lo-eviti/>
- Federici, S. e Vinci G. (2006), **Spring 2.0 e Inversion of Control (IoC)**,
<http://www.aldaran.org/press/java/2006/12/02/spring-20-e-inversion-of-control-ioc/>
- Federici, S. e Vinci G. (2007), **Maven2 - Project Managment**,
<http://www.aldaran.org/press/java/2007/10/22/maven2-project-managment/>
- Federici, S. (2007), **Terracotta DSO**, <http://www.aldaran.org/press/java/terracotta-dso/>
- Fowler, M. (1997), **ULM distilled : applying the standard object modeling language**, Boston, Addison-Wesley
- Fowler, M. (1999), **Refactoring: Improving the Design of Existing Code**, Boston, Addison-Wesley
- Fowler, M. (2003), **Patterns of Enterprise Application Architecture**, Boston, Addison-Wesley
- Fowler, M. (2004), **Inversion of Control Containers and the Dependency Injection pattern**,
<http://www.martinfowler.com/articles/injection.html>
- Kerievsky, J. (2004), **Refactoring to Patterns**, Boston, Addison-Wesley
- Knoernschild, K. (2001), **Java Design: Objects, UML, and Process**, Boston, Addison-Wesley
- Larman, C. (1998), **Applying UML and patterns : an introduction to object-oriented analysis and design**, Upper Saddle River N.J. , Prentice Hall
- Larman, C. (2003), **Agile and Iterative Development: A Manager's Guide**, Boston, Addison-Wesley
- Turco, R. (2003), **Refactoring: la teoria in pratica**,
<http://www.geocities.com/SiliconValley/Port/3264/corsi/refactoring/refactoring.htm>