Parte 3: la pratica.

In questa sezione viene mostrato il lavoro svolto nel tentativo di applicare programmazione estrema.

9. Prima di conoscere XP.

Più studiavo ed approfondivo XP, più mi accorgevo che già avevo tentato di sperimentare molte delle tecniche di questa metodologia senza neanche conoscerle. Era stato semplicemente il **buon senso** ad indicarmele.

Alcuni mesi fa, io e Francesco abbiamo sviluppato assieme un interprete lisp ed uno prolog in java. Progetto che abbiamo utilizzato come elaborato per un esame. Io precedentemente avevo avuto piccole esperienze di programmazione, mentre Francesco non era mai andato oltre programmi quali il bubble sort scritto in c o pascal.

Insieme su questo progetto di interprete abbiamo lavorato per circa 5 mesi.

Abitiamo a circa mezz'ora di macchina, dunque qualche volta lavoravamo assieme e qualche volta ognuno a casa sua, tenendoci sempre in stretto contatto tramite e-mail e telefono.

Quando eravamo assieme applicavamo **pair programming** (senza nemmeno sapere cosa fosse) esattamente nel modo descritto da Kent Beck. Forse eravamo costretti dal fatto che sia a casa mia che a casa sua avessimo a disposizione un solo computer.

Quando eravamo ognuno a casa propria applicavamo un concetto di pair programming che penso sia nuovo anche per XP: *pair programming a distanza*. Esistono strumenti, quali l'instant message, che potrebbero essere utilizzati per supportare pratiche del genere, ma per la mia esperienza non li considero essenziali.

Appena una parte di codice veniva scritta da uno di noi due, veniva subito inviata all'altro. Questo poi rileggeva il codice e lo analizzava, eventualmente modificandolo e correggendolo.

Ci è capitato parecchie volte che chi correggeva il codice trovasse facilmente delle incongruenze o piccoli errori. Era come se uno avesse

in mano la tastiera per scrivere codice e l'altro stesse osservando lo schermo ed allo stesso tempo avesse anche il tempo di ragionare ad un più alto livello strategico. Capitava che questa seconda persona semplicemente restituisse una e-mail con un elenco di dubbi che gli si presentavano.

L'overhead dovuto al controllo che la seconda persona faceva ci è stato ampiamente ripagato: abbiamo evitato diverse fasi di debugging in cerca di errori e, cosa fondamentale, aumentava enormemente la conoscenza del sistema da parte nostra: ogni parte era ben nota ad entrambi.

All'inizio, per ogni classe che facevamo, aggiungevamo il metodo main per poterla **testare**. Appena eravamo sicuri che funzionasse eliminavamo il metodo main per lasciare solo quelli che ci servivano. Poi veniva il momento di modificare il codice e la necessità di testare nuovamente. Veniva così aggiunto di nuovo il metodo main e spesso conteneva lo stesso identico codice scritto la settimana o il mese prima.

Codice che era stato cancellato e perso (all’interno dei numerosi back up) e dunque bisognava riscriverlo da zero.

Dopo un paio di volte ci siamo fatti leggermente più furbi ed abbiamo deciso di lasciare i metodi main. Nella release finale del progetto che abbiamo consegnato al professore sono ancora presenti tutti i test. Anzi siamo riusciti a sfruttare questi metodi per aggiungere nuove funzionalità al nostro programma.

Faccio un esempio. Il nostro interprete era stato pensato per avere una interfaccia windows. Alla fine, oltre all'interfaccia windows, il nostro programma poteva essere utilizzato anche da riga di comando e sfruttava la shell DOS. Questo è stato fatto senza nessuno sforzo riutilizzando i metodi main che usavamo per provare come funzionavano i due interpreti lisp e prolog che stavamo costruendo.

Inoltre, poiché java è notoriamente carente in fatto di prestazioni sull'interfaccia grafica, risulta molto più comodo utilizzare i nostri interpreti sfruttando il DOS, e solo così possono esserne messe in evidenza le potenzialità ed i limiti.

Quando trovavamo una frase che il nostro riconoscitore non interpretava bene, la aggiungevamo al metodo main e successivamente correggevamo l'errore.

Alla fine avevamo tanti piccoli test che ci assicuravano che i nostri interpreti funzionassero correttamente anche dopo modifiche sostanziali.

L'unica pecca era che i nostri test *non erano automatici*!

Per quanto riguarda il **refactoring**, ci è venuto naturale. Io sono pignolo e mi piacciono le cose fatte bene. Anche Francesco un po' lo è e si è fatto facilmente influenzare. Non esiste una parte del nostro codice che non sia stata modificata almeno due o tre volte. Molte modifiche erano semplici e servivano solo per rendere il codice più comprensibile (*autoesplicativo* direbbero i fautori di XP), altre comprendevano parti ampie di codice ed erano funzionali.

Abbiamo per esempio riscritto da zero il parser prolog quando ci siamo accorti che il nostro non rispecchiava le specifiche prolog con le precedenze degli operatori definite nel range 1-1200 e le varie regole di associatività per operatori unari e binari.

Sul primo parser avevamo lavorato parecchio, soprattutto per cercare di ovviare all'inerente complessità ed ambiguità della sintassi prolog. Ci sembrava che riscrivere il parser fosse come ripartire da capo.

Abbiamo poi deciso che non costava molto **provare** a fare un nuovo parser e finché non avesse funzionato potevamo tenerci quello vecchio.

Se il nuovo codice scritto non avesse dato subito dei risultati sarebbe stato subito abbandonato. È stata una scommessa che ha richiesto parecchio **coraggio**, ma alla fine è stata vinta.

Nella release finale entrambi i parser funzionano correttamente, anche se il secondo ha molte più funzionalità.

Altro caso analogo è stato quando ho deciso di riscrivere da capo l'unificatore prolog quando mi sono accorto che quello che avevamo era altamente inefficiente e praticamente illeggibile (arrivava a 6 livelli di if innestati).

Il fatto di lavorare a distanza, spesso sulle stesse parti di codice,

implicava una rigida politica di **integrazione** delle varie parti.

Ogni vota che ognuno di noi faceva una modifica che riteneva sostanziale e che avrebbe potuto influenzare il lavoro dell'altro si procedeva ad integrare tramite strumenti per il confronto dei file.

Più di rado si facevano controlli su tutto il software per integrare anche le piccole modifiche. Ci è capitato anche di avere fatto la stessa modifica o di aver corretto lo stesso errore contemporaneamente, ignari di cosa stesse facendo l'altro.

Sul tavolo avevo sempre un foglio di carta in cui annotavo tutti i lavori da fare (**to do**) successivamente, che mi venivano in mente mentre scrivevo il codice. Dalla lista sceglievo poi sempre il più importante e quello che avesse **aggiungesse maggior valore**. Anche ad interprete ultimato la lista non era ancora vuota, ma le cose rimaste erano di poco conto, oppure erano state rese inutili da cambiamenti di rotta.

Ora non mi vengono in mente altre analogie con XP, ma sono sicuro che ce ne sono altre. Mentre leggevo il libro di Kent Beck spesso mi sorprendevo scoprendo quante volte dicevo a me stesso "questo l’ho già provato". Forse è proprio questa la ragione per cui XP mi ha subito affascinato. XP comprende ed organizza nel migliore dei modi un insieme di tecniche che derivano dalle buone abitudini prese da chi si deve scontrare quotidianamente con la stesura di programmi.

Non c'è niente di artificioso e quasi tutto deriva dal buon senso comune.

Un'ultima cosa che voglio sottolineare è che scrivere il codice per i due interpreti è stato **divertente**. Siamo stati entrambi soddisfatti del lavoro svolto ed abbiamo condiviso la gioia e la consapevolezza di aver costruito software di alta **qualità**. Idee su come proseguire e migliorare il nostro progetto ne avevamo tantissime (il giorno dopo l'esame, in un pomeriggio siamo riusciti ad introdurre la tail recursion per l'interprete lisp) ed ogni parte del nostro codice è pronto a richiedere ulteriori modifiche, ma per l'esame era più che sufficiente per stupire il professore.

10. Le esigenze che ci hanno spinto.

Presso la ditta dove ho fatto la tesi era stato sviluppato un applicativo denominato *Acquisizione Messaggi* dedicato alla catalogazione all'interno di un database di documenti reperiti sul web.

Durante i suoi sette mesi di vita, l'applicazione aveva subito parecchie innovazioni e correzioni. Era stato necessario considerare diversi casi particolari ed eccezioni: dall'eliminazione di parti di documenti duplicate (si pensi alle versioni con frame e quelle senza), dannose durante la fase di indicizzazione, ai vari problemi di codifica (windows, mac, lettere accentate, formati delle date, …).

Ognuno di questi veniva trattato separatamente ed ogni volta che si presentava un altro problema l'applicazione cresceva e diventava più complessa da gestire.

Inoltre la fase di acquisizione non era separata da quella della catalogazione, dunque per verificare che una modifica funzionasse era necessario avere un documento on-line che presentasse quel particolare inconveniente.

A causa di tutti questi problemi spesso capitava che la correzione di un errore ne introducesse altri, che oltretutto non saltavano fuori immediatamente, ma durante il normale utilizzo del programma da parte della redazione.

Acquisizione messaggi era stata scritta per passi successivi, senza partire da una chiara idea di ciò che si sarebbe voluto ottenere. Era stata concepita come un semplice ausilio alla realizzazione di newsletter, per diventare uno strumento complesso ed indispensabile.

Ora che erano ben noti i suoi obiettivi e che gli utenti avevano effettivamente compreso quali vantaggi potevano trarre dall'applicazione, sarebbe stato utile riorganizzare tutto il codice per poter raffinare il programma e renderlo stabile.

Ovviamente nessuno aveva voglia di rimaneggiare un software così

complesso che ora, nonostante tutto, *funzionava*.

Se aggiungere ulteriori modifiche sembrava arduo e rischioso, ristrutturarlo completamente era semplicemente folle: le continue modifiche avevano reso il codice fortemente interconnesso ed una piccola variazione inesorabilmente si ripercuoteva sull'intero sistema.

Si era partiti da una corretta implementazione della programmazione ad oggetti per giungere sempre più ad un insieme confuso di oggetti in cui difficilmente si potevano riscontrare le caratteristiche essenziali quali l'ereditarietà, l'isolamento, la scomposizione, l'assegnazione e la suddivisione delle responsabilità.

Una possibile soluzione era riscrivere il codice partendo da zero, ma parallelamente si doveva continuare a sviluppare la versione attuale perché le richieste da parte della redazione erano sempre più pressanti.

Altrimenti poteva essere seguita una metodologia che riducesse al minimo i rischi e che permettesse di fare controlli accurati sul corretto funzionamento anche dopo pesanti modifiche.

Fu a questo punto che si prese in considerazione XP. Le promesse erano di risolvere la maggior parte dei problemi senza rischiare di compromettere il lavoro svolto fino a quel punto.

Poteva essere fatto tutto per piccoli passi successivi controllando di volta in volta che il sistema continuasse a funzionare.

Anche l'adesione alla metodologia XP poteva essere fatto gradualmente, cercando di imparare e di introdurre una tecnica alla volta.

11. Le esperienze.

11.1. I primi approcci.

Giulio, il responsabile del settore software, aveva appena deciso di provare ad implementare XP per risolvere i problemi che ho descritto nel precedente capitolo quando io ho iniziato a collaborare con la ditta IBN (Internet Business News) per sperimentare XP per la mia tesi.

Io ero la quarta persona del team ed in meno di un mese si sarebbero aggiunte due nuove persone per supportare un sistema in continua e rapida evoluzione, dove le idee ed i progetti di nuovi sviluppi non mancavano ed il solo problema era scegliere in ogni momento quale fossero gli argomenti più utili da intraprendere per aumentare il valore dell'azienda.

Per poter modificare in maniera sicura il codice di acquisizione messaggi era necessario prima costruire una serie di test che assicurassero di non introdurre errori.
Si sarebbe poi potuto passare a ristrutturare per piccoli passi il codice, lavorando in coppia per aumentare la sicurezza.

Insomma era l'ora di sperimentare se questa nuova programmazione estrema portava tutti i vantaggi descritti da chi l'aveva già applicata.

IBN è una ditta che fa ampio uso di database per catalogare documenti web e rendere le informazioni nuovamente disponibili accedendo tramite browser.
Il primo problema che ci si poneva era come costruire test automatici per controllare gli accessi ai database.

Navigando su internet all'interno dei gruppi di discussione su extremeprogramming un argomento ampiamente trattato riguarda la realizzazione di test.

Alcuni gruppi di sviluppatori hanno costruito tool per facilitare l'esecuzione dei test in modo automatico alla fine della compilazione.

Il più famoso tra questi tool è **Junit**, implementato in linguaggio java da Eric Gamma e Kent Beck, i fautori di XP.

Junit è open-source ed è stato tradotto in diversi altri linguaggi da vari gruppi di programmatori.

Noi, utilizziamo l'ambiente di programmazione **WebObjects** (vedi appendice A) della Apple, basato sul linguaggio Objective C (vedi appendice B), quindi ci serviva la versione di Junit riscritta dalla ditta svizzera Sente e rinominata **OCUnit**.

11.2. OCUnit e SenTestingKit.

Alla fine della compilazione parte un eseguibile chiamato **otest** che ha il compito di lanciare i test e di mostrare il risultato finale.

Per individuare i test, *otest* sfrutta SenTestingKit, un framework (librerie di classi) che racchiude tutto il funzionamento del tool.

I test sono dei metodi che vengono eseguiti uno alla volta. Ogni metodo è un test a sé stante e questi sono raccolti all'interno di diverse classi denominate testSuite per raggrupparli ed isolarli.

Ogni classe che contenga test che devono essere eseguiti in automatico deve derivare dalla classe SenTestCase appartenente al framework SenTestingKit.

Ocunit si preoccupa di cercare tutte le classi discendenti di SenTestCase e di avviare tutti i loro metodi il cui nome inizia per *test*.

SenTestingKit è stata studiata per separare la fase di inizializzazione del test dal test vero e proprio. A questo proposito la classe SenTestCase contiene due metodi che possono essere ridefiniti dalle sottoclassi e che vengono avviati rispettivamente prima e dopo ogni test.

Il primo metodo, *setUp*, serve per preparare l'ambiente in cui eseguire

il test. Ad esempio se vogliamo testare il funzionamento di un oggetto, nel setUp potremmo inserire il codice per creare l'oggetto stesso e gli oggetti a lui correlati.

Il secondo metodo, tearDown, serve per esempio per distruggere gli oggetti creati nel metodo *setUp*.

In genere si raggruppano tutti i test relativi ad un oggetto all'interno della stessa testSuite e si inseriscono nei metodi setUp e tearDown tutte le operazioni che andrebbero ripetute per ogni singolo test.

Lo scopo dei test automatici è permettere all'utente di sapere se l'applicazione continua a funzionare correttamente e tutti i test passano .

L'unica risposta che si riceve è il numero di test che falliscono ed in tal caso viene facilitata l'individuazione dei test che hanno fallito. Il tutto deve essere il più trasparente possibile. Addirittura si utilizzano interfacce grafiche che nascondono il funzionamento del tool per i test e presentano solo una spia verde in caso di successo.

Molti sviluppatori fanno girare i test anche quando sono sicuri di non aver introdotto errori, solo per poter vedere la spia verde accesa ed aumentare così il loro livello di sicurezza.

11.3. IBNDBTestingKit.

Ho già accennato al fatto che noi avevamo la necessità di testare gli accessi al database.

Un primo approccio potrebbe essere quello di utilizzare i dati già esistenti sul database e controllare se i metodi che abbiamo usato per manipolare i dati funzionino correttamente.

Un grave difetto di questo approccio è che non considera il fatto che il database è dinamico ed i dati sono differenti se eseguiamo il test in momenti diversi e i risultati potrebbero variare.

Per questo motivo è più consigliabile costruire un database separato per ogni gruppo di test, isolando i vari problemi. Si evita così anche di andare a danneggiare i dati all'interno del database qualora un oggetto non fosse stato costruito correttamente e malauguratamente andasse a

scrive dati non validi o coerenti all'interno del database.

Le operazioni che devono essere fatte per creare un database separato (o semplicemente un utente diverso all'interno dello stesso database se lavoriamo con Oracle) sono numerose e non possono essere gestite manualmente, soprattutto se il test modifica il database ed ogni volta bisognerebbe ripristinare le condizioni di partenza.

Abbiamo dunque deciso di costruire una estensione al framework SenTestingKit che permettesse di gestire tutto ciò automaticamente. È dunque nato IBNDBTestingKit.

Questo framework si occupa di creare il database, creare le tabelle necessarie, popolarle dopo aver convertito i dati a seconda del tipo di database, aggiungere i vincoli. Tutte queste operazioni vengono fatte all'interno del metodo *setUp* della classe IBNDBTestCase che deriva da SenTestCase.

Nella *tearDown* ci si limita a distruggere il database.

Per diminuire il numero di accessi al database e per velocizzare i test abbiamo pensato che non fosse necessario distruggere il database ogni volta. Alcuni test non modificavano i dati e si limitavano caricare dei dati in memoria per accederci.

Sarebbe stato utile anche avere due differenti metodi *setUp*, il primo che veniva eseguito solo una volta per un gruppo di test ed il secondo che invece veniva ripetuto per ogni test.

Questa considerazione ci ha portato ad introdurre un metodo *setUp* di classe, oltre a quello già esistente di istanza. Anzi abbiamo comunicato la nostra esigenza direttamente a Sente che ci ha prontamente risposto dicendoci che avrebbe inserito due nuovi metodi nella classe SenTestCase: +*setUp* e +*tearDown*, metodi di classe riconoscibili grazie al segno + che li precede, in contrapposizione del – che viene usato per i metodi d'istanza (simbologia presa dalla sintassi del linguaggio objective C: vedi appendice B).

All'inizio utilizzavamo FrontBase come database, poi siamo passati ad Oracle per ragioni di sicurezza e praticità. Questo ci ha costretto a

modificare e ad aggiungere nuove funzionalità.

Trattandosi di un tool per i test ci sembrava esagerato costruire ulteriori test per controllare il suo funzionamento. Ancora non eravamo entrati nella mentalità proposta da XP: prima si scrivono i test e poi il codice. Sono servite numerose fasi di debugging e parecchio tempo perso per comprendere che anche IBNDBTestingKit doveva avere la sua suite di test automatici.

Questo fatto evidenzia un problema fondamentale: non si rischia di procedere all'infinito? La domanda ovvia è: chi testa i test?

La risposta che posso dare dopo le esperienze che ho avuto e usando il **buon senso** è che ogni parte di codice che è abbastanza complesso da poter nascondere subdoli errori dovrebbe essere testato. La caratteristica dei test è essere estremamente semplici in modo da evitare assolutamente la probabilità di commettere errori.

Un test deve essere racchiuso in poche linee di codice: l'invocazione di un metodo e la verifica di uno stato sarebbe l'ideale.

In questo modo, anche quando il codice cambia, il lavoro da dedicare per adeguare i test è minimo.

Anche se IBNDBTestingKit è uno strumento per eseguire i test automatici è comunque un tool abbastanza complesso. Per aumentare la sua efficienza accede a chiamate a basso livello che spesso sono mal documentate e dunque la probabilità di malfunzionamenti è alta.

Durante il periodo che ho passato presso IBN il tool è sempre stato in continua evoluzione, è stato raffinato ogni volta che trovavamo una situazione non gestibile, oppure non avevamo bene compreso il funzionamento dell'ambiente sottostante o del database. Dunque i test per questo framework sono essenziali.

Il framework ci è stato anche richiesto da altre ditte che avevano i nostri stessi problemi e che abbiamo contattato tramite i gruppi di discussione su extremeprogramming. Ci hanno anche chiesto il permesso di tradurre in nostro framework in java e di renderlo pubblico.

Tutto questo lavoro potrebbe sembrare eccessivi e gratuito. Sicuramente è dispendioso e potrebbe far sorgere il dubbio se

effettivamente valga la pena fare tanti sforzi per riuscire a mettere in piedi una serie di test.

Se confrontiamo i sacrifici con i risultati ottenuti posso tranquillamente dire che ne è valsa la pena.

Innanzitutto bisogna premettere che i test automatici sono stati inventati da poco e sono ancora in pieno sviluppo gli strumenti per riuscire ad implementarli. Per IBNDBTestingKit noi abbiamo speso costose risorse, ma chi utilizzerà il nostro framework si troverà la strada in discesa, così come anche noi, nello stesso modo, abbiamo potuto sfruttare il lavoro di altri quando abbiamo utilizzato SenTestingKit, oppure HttpUnit, di cui parlerò di seguito.

Il tempo è stato speso, ma non sprecato. Per esempio io sono riuscito a conoscere abbastanza bene le classi di EOF, il framework dell'ambiente di programmazione che serve per rendere trasparenti gli accessi al database ed evitare il codice SQL inline e, pur essendo stato solo alcuni mesi presso IBN, alla fine ero io quello che conosceva meglio alcune parti del sistema a basso livello.

11.4. Le pratiche di XP al lavoro.

Durante la fase di sviluppo di IBNDBTestingKit scrivevamo i test per i framework generali che erano stati sviluppati all'interno di IBN e che venivano utilizzati da tutte le applicazioni.

Per scrivere i test bisogna conoscere ciò che si vuole testare e aggiungere successivamente i test come stavamo facendo noi era una faccenda abbastanza complessa.

Scrivere test prima di scrivere il codice può essere una attività stimolante che permette di analizzare il problema per poterlo affrontare meglio e ci assicura sulla corretta implementazione quando i test passano tutti.

Ma scrivere test per codice prodotto tempo indietro e soprattutto da altre persone può a volte essere frustrante.

Innanzitutto c'è una prima fase di comprensione del codice, poi i

tentativi di scrittura di un abbozzo dei test per comprendere meglio il funzionamento. Infine si scrivono i veri test.

Durante questo lavoro è facile scoprire che alcune parti sono poco comprensibili e verrebbe la voglia di riscriverle, ma ancora non possiamo farlo in sicurezza perché non esistono test e non possiamo sapere tutte le possibili implicazioni che una modifica potrebbe produrre in qualsiasi altra parte del programma.

Molte di queste difficoltà sono state superate in parte grazie alla programmazione in coppia. Avere di fianco nei momenti critici una persona che conosceva meglio il codice o che semplicemente riusciva a rilevare in tempo alcune sviste rendeva tutto più facile e manteneva alta la fiducia.

Altro fatto positivo era avere una copia locale del framework in cui fare gli esperimenti, per poi inserirli nel repository condiviso da tutti i programmatori solo quando si era sicuri che le modifiche apportate non introducessero bug.

Mentre si aggiungevano questi test, si continuava a perfezionare IBNDBTestingKit. Questo framework veniva continuamente ristrutturato per mantenerlo semplice e leggibile in modo da rendere facile l’introduzione delle modifiche.

Quando siamo dovuti passare da FrontBase ad Oracle il lavoro ci è stato reso facile grazie alla modularità che aveva il framework in quel momento. Inoltre una fase di ristrutturazione ha reso le cose ancora più agevoli.

Per esempio c’era una classe in cui tutti i metodi si passavano un parametro a vicenda richiamandosi l'un l'altra, mentre questo parametro veniva effettivamente utilizzato solo da uno dei metodi; creando una variabile istanza ho potuto evitare tutti questi passaggi inutili.

Un altro caso era quando mi sono accorto che i metodi delle classi che racchiudevano la strategia del database (FrontBase o Oracle) venivano invocate più volte dal testManager e bisognava passare ogni volta diversi parametri. Aggiungendo anche la strategia per Oracle il numero dei parametri sarebbe cresciuto perché i due database sono

sostanzialmente diversi. Un metodo con troppi parametri è poco comprensibile. Visto che le strategie ed il testManager erano strettamente collegati abbiamo sostituito tutti i parametri con uno unico: passavamo direttamente un puntatore a testManager.

Ogni volta che facevo una ristrutturazione ad IBNDBTestingKit controllavo se i test continuavano a passare tutti. Quando integravo eseguivo anche i test che avevamo aggiunto o stavamo aggiungendo ai framework per avere maggiore sicurezza che le cose continuassero a funzionare e fare in modo che la probabilità di trovare bug in futuro fosse molto bassa.

Lo sviluppo del framework IBNDBTestingKit era sotto la mia responsabilità, ma il peso veniva attenuato grazie alla programmazione in coppia.

11.5. Property list (plist).

IBNDBTestingKit basa il suo funzionamento su una serie di opzioni che possono essere settate tramite un file di configurazione.

Per esempio, per velocizzare i test, alcune tabelle read-only vengono rese permanenti per tagliare i tempi di creazione e distruzione delle stesse. Oppure poteva essere eseguito un unico test per cercare l'errore quando questo test non passava.

Per far ciò era sufficiente modificare le opzioni di configurazione.

Il framework legge tutti questi parametri da un file formattato secondo le regole delle plist o property list. All'interno di questo file c'è anche un elenco di altri file, formattati nello stesso modo, che contengono la rappresentazione esterna dei dati con cui popolare il database per eseguire i test.

Le plist sono organizzate come un NSDictionary (dizionario di oggetti e rispettive chiavi) e possono contenere array, stringhe, dati puri e dictionary annidati.

Queste plist vengono create automaticamente esportando i dati da un

database, ma per i test devono esser fatte ad hoc per riuscire ad emulare particolari situazioni. Si cerca in genere di riprodurre in formato ridotto il database di lavoro e la creazione di queste plist è fatta a mano.

Quando vengono lette tramite un metodo apposito di NSDictionary, se la sintassi non è corretta viene restituito un puntatore a nil (null pointer) senza segnalare il punto dove la lettura si è bloccata.

Questo ha rappresentato un punto debole di IBNDBTestingKit sin dall'inizio. Per questo motivi e per evitare lunghe e noiose ricerche di banali errori, ho costruito un analizzatore sintattico per plist che non fa altro che segnalare il punto dove viene individuato un errore.

Come prevede XP ho prima costruito alcuni test per capire quali funzionalità volevo ottenere. Naturalmente fallivano tutti dato che il codice ancora non c'era.

Ho poi iniziato a scrivere il codice ed ho terminato non appena tutti i test completavano con successo. Per scrivere il codice usavo la tecnica che dice di utilizzare sempre "la cosa più semplice che funzioni". Non è stato così difficile creare velocemente del codice semplice per una classe che proprio banale non è.

Ho così ottenuto una classe funzionante, anche se con codice ridondante e replicato. Inoltre non ero ancora sicuro che fossero coperti tutti i possibili casi di errore nella scrittura della plist.

Ho dunque iniziato a pensare ad ogni possibile e plausibile modo per far fallire il codice che avevo scritto e per ogni idea sensata aggiungevo un test. Sono state così corrette alcune imperfezioni e coperti alcuni errori nelle plist che non venivano ben segnalati.

Con la sicurezza che mi davano i test sono poi passato ad eseguire un refactoring per eliminare del codice replicato e per rendere più leggibile il codice scritto.

Grazie ai test, non mi sono preoccupato molto mentre facevo le modifiche per la ristrutturazione. In meno di venti minuti tutto era come volevo. Eseguo i test ed uno non passa. Mi viene restituita una errata posizione dell'errore: la linea ha un valore spostato di 1.

Il difetto è subito individuato ed i test eseguono tutti correttamente.

Il refactoring non era fondamentale, ma è risultato utile quando, dopo qualche settimana, mi fu chiesto di aggiungere una piccola funzionalità: in caso di errore stampare anche uno stack che rappresentasse il livello di innestamento tra dictionary ed array.

È bastata una breve letta al codice per rinfrescare le idee e poter subito fare le modifiche necessarie. Modifiche limitate a parti localizzate di codice, proprio perché ben ristrutturato.

Se le ridondanze fossero state ancora presenti avrei dovuto replicare le modifiche per ogni parte di codice duplicato.

Questa piccola utility per controllare le plist è stata ampiamente utilizzata ed apprezzata durante la costruzione di ogni test che accedesse al database.

Alla fine abbiamo deciso di integrare il controllo delle plist all'interno del framework per i test sul database: ogni volta che la lettura di una plist ritornava un valore nil veniva attivato in automatico il controllo per conoscere il punto in cui intervenire.

Questo ha permesso do ridurre drasticamente il tempo per mettere in piedi nuovi test.

11.6. I vantaggi di avere IBNDBTestingKit.

Il mio lavoro successivo è stato quello di riordinare e fare alcune modifiche ad IBNEOExtensions, uno dei framework base sviluppato precedentemente dalla ditta.

Tale framework era già fornito di test che avevamo aggiunto durante lo sviluppo di IBNDBTestingKit. Ogni volta che spostavo una classe da file a file, oppure quando spostavo un metodo da classe a classe o semplicemente quando rinominavo una variabile per renderne più chiaro il significato, grazie ai test potevo facilmente trovare potenziali errori che avrebbero rallentato il lavoro di chi successivamente avrebbe utilizzato il framework.

Ho anche aggiunto alcune classi ed alcuni metodi e per questi ho

aggiunto anche i test automatici. Purtroppo, non conoscendo a fondo il sistema nel suo complesso e non potendo immaginarmi tutti gli usi potenziali delle classi da me aggiunte, i test erano più orientati al codice che alle funzionalità.

Se le applicazioni che usavano il framework fossero già state dotate di test avrei subito avuto la sicurezza di aver fatto un buon lavoro. Invece, senza test, i difetti nascosti sono saltate fuori uno alla volta come errori che hanno bloccato l'applicazione principale per vari debug.

Quello che mi ha permesso di fare modifiche senza preoccuparmi troppo degli eventuali effetti finali è stata la possibilità di lavorare in locale. Successivamente l'integrazione è stata fatta per gradi, un pezzetto alla volta, utilizzando anche la tecnica del programmazione in coppia: io conoscevo bene il framework, le modifiche fatte ed IBNDBTestingKit; il mio collega conosceva bene l'applicazione.

Non era ancora un pair programming tale e quale viene predicato da XP. Infatti nel nostro caso veniva fatto per integrare e per ovviare al problema che ancora non tutto il software era fornito di test automatici, comunque era già un notevole passo verso XP.

11.7. Alcuni risultati confortanti.

La maggior parte delle modifiche ad *Acquisizione Messaggi* (vedi capitolo 10) sono alla fine state fate da Gabriele, un neoassunto che fino a quel momento non conosceva nulla dell’ambiente di programmazione WebObjects, né del linguaggio Objective C, né tanto meno dell’applicazione.

All’inizio ha lavorato in coppia con me mentre imparava a scrivere test utilizzando IBNDBTestingKit. Entrambi, appena avevamo un dubbio su acquisizione messaggi chiamavamo Giulio, ovvero colui che la conosceva meglio e che ci aveva passato sopra più tempo.

Successivamente Gabriele ha lavorato in coppia con Giulio per completare la scrittura di tutti i test e per impostare la riorganizzazione del codice.

Infine ha collaborato con Stefano per la scrittura del nuovo codice e l'aggiunta delle nuove funzionalità che venivano continuamente richieste.

Attualmente Acquisizione Messaggi è una applicazione che funziona correttamente e che continua ad essere utilizzata quotidianamente dalla redazione. Ogni volta che un documento non viene correttamente catalogato è possibile costruire un test per trattare il nuovo caso.

L'aggiunta di questi nuovi test è eseguita in maniera molto semplice e veloce grazie alla modularità dell'applicazione.

Una volta scritto il test si può modificare il codice con la ragionevole certezza di non reintrodurre bachi che erano stati corretti precedentemente.

Infatti questi test si aggiungono a quelli per il controllo delle funzionalità dell'applicazione e possono essere eseguiti tutti assieme in pochi minuti e senza bisogno di accedere a documenti on-line: è sufficiente una copia locale degli articoli che rappresentano eccezioni particolari.

11.7. Httpunit.

Leggendo gli interventi sulla mailing list di extremeprogramming si può notare come la difficoltà di testare le interfacce grafiche sia un problema molto sentito. Fare test su GUI significa dover simulare l'intervento umano e ciò non è banale. Le difficoltà derivano anche dalla inerente complessità e dal numero di dettagli presenti in una interfaccia grafica.

Possono esserci sia errori che pregiudicano il funzionamento (come per esempio la mancanza di un bottone, oppure l'assenza di una procedura che parta alla sua pressione), sia errori che modificano la grafica e l'aspetto.

Probabilmente per questi ultimi creare dei test automatici sarebbe inutile ed estremamente dispendioso. Se un individui riesce ad accorgersi che qualcosa non va semplicemente da un'occhiata, un test automatico dovrebbe controllare tutte le parti che vengono aggiunte alla

grafica. In pratica sarebbe come replicare la costruzione dell'interfaccia.
Improponibile!

Invece testare la funzionalità è una cosa possibile ed anche abbastanza semplice se si hanno gli strumenti adatti. Si può facilmente simulare la pressione di un bottone via software e controllare i risultati finali: ad esempio l'esistenza di un nuovo file sull'hard disk se il bottone corrisponde ad un salvataggio.

Nel caso di IBN le interfacce grafiche sono costruite tutte tramite pagine html.
Esiste un tool costruito in java denominato httpunit che si sostituisce al browser e permette di accedere alle pagine html programmaticamente.

Httpunit sfrutta alcune librerie prodotte da W3c definite secondo gli standard per internet. Tramite esse è possibile accedere a tutti i componenti di una pagina html in modo molto semplice. Ad esempio è possibile cercare un link con un certo nome, oppure una form o un frame. Per le form è possibile riempire i campi e fare la submit. Per i link è possibile simulare il click del mouse e richiedere la pagina presente all'indirizzo URL indicato nel link.

Anche httpunit, così come IBNDBTestingKit è in continua evoluzione e molte delle funzionalità non sono ancora complete. La prima volta che l'abbiamo provato non erano pienamente supportate le form ed abbiamo dovuto scaricare direttamente dal repository la versione più aggiornata per farla funzionare (aggiornata appena due giorni prima).
Un'altra funzionalità non presente era la possibilità di leggere i file locali (presenti sul proprio hard disk). Visto che sono disponibili sia i sorgenti dell'httpunit, che di *jtidy* (una delle librerie) abbiamo provveduto ad aggiungere le modifiche necessarie. Abbiamo poi inviato le nostre modifiche a Gold Russel, colui che si occupa della gestione dell'httpunit, e dopo due giorni le nuove funzionalità erano integrate nel repository e disponibili a chiunque avesse un acceso ad internet.

Io e Gabriele abbiamo provato ad utilizzare httpunit per scrivere alcuni test su interfacce web esistenti e ci siamo accorti che è abbastanza

semplice.

Poiché le nostre applicazioni web accedono ai database sarebbe stato opportuno riuscire ad integrare l'httpunit all'interno dell'IBNDBTestingKit.

Purtroppo i linguaggi di programmazione erano diversi e, nonostante esistano alcuni strumenti WebObjects che permettano di eseguire programmi scritti in parte in linguaggio Objective C ed in parte in Java, non era possibile fare ciò che avevamo in mente.

Si potevano tradurre httpunit e le librerie in Objective C, ma sarebbe stato un lavoro troppo dispendioso e rischioso: a priori non era nota la difficoltà.

Visto che httpunit è abbastanza indipendente si è alla fine deciso di continuare a scrivere i test per l'interfaccia web in java. Per eseguire i test dall'interno di Objective C abbiamo creato una categoria (vedi appendice B) su SenTestingKit che invocasse il compilatore java e quindi l'interprete java. In questo modo i test potevano essere eseguiti ed il risultato ritornato.

Una categoria Objective C è ereditata, dunque anche IBNDBTestingKit poteva sfruttare l'httpunit.

In pratica SenTestingKit ed IBNDBTestingKit si occupavano rispettivamente di lanciare l'applicazione e di creare il database, poi successivamente entrava in gioco httpunit che testava l'applicazione invocando alcune pagine web costruite dinamicamente a partire dai dati sul database.

12. WebFetcher.

Per comprendere meglio l'effettiva utilità di alcune pratiche ho svolto alcuni prove mentre io e Giancarlo costruivamo assieme un nuovo tool per la catalogazione automatica di file html: WebFetcher.

Il progetto partiva praticamente da zero (a parte il fatto di dover utilizzare una tabella già esistente nel database) e questo ci permetteva di applicare le tecniche di XP in modo diretto, senza essere costretti ad aggiungere test a software già esistenti. Anche le librerie realizzate all'interno di IBN che utilizzavamo erano a questo punto corredate di propri test.

La responsabilità del lavoro era di Giancarlo ed io ero il suo compagno.

Come primo passo ci siamo costruiti un test funzionale, assistiti da Giulio che in questo caso rappresentava il nostro utente e che ci aveva fornito le specifiche.

Previsione sul completamento di un prototipo funzionante del lavoro: circa 2 giorni.

La tentazione di scrivere immediatamente il codice del tool, senza passare per i test, era stata forte, ma grazie alla pair programming siamo riusciti a sostenerci a vicenda ed a vincere le nostre debolezze.

Sicuramente il test funzionale scritto non era completo a causa della nostra ancor scarsa esperienza, comunque ci ha un po' chiarito le idee sui vari passi che avremmo dovuto intraprendere e ci ha permesso di scomporre il problema in diversi task.

Partendo dal primo di questi abbiamo costruito un test e successivamente il codice. Giancarlo scriveva sulla tastiera ed io osservavo, anche se spesso mi capitava di prendergli il mouse per indicargli alcuni errori o punti che non erano ben chiari.

Il test non funzionò la prima volta che lo facemmo girare, comunque con alcune piccole correzioni il primo task risultò essere completo.

Per il secondo task abbiamo poi deciso di separarci: lo avrebbe fatto Giancarlo, mentre io mi occupavo di leggere alcuni articoli per la tesi. Ad un certo punto Giancarlo mi ha interrotto per chiedermi di aiutarlo: era convinto di avere finito il secondo task, ma il nuovo test che aveva scritto non passava.

Dopo una lunga fase di debugging e la correzione di diversi errori siamo riusciti a sistemare il problema. Per fare ciò sono stato costretto a riguardare tutto il codice assieme a Giancarlo ed a meditare sulle scelte già fatte da lui.

Decidemmo che anche la terza fase l'avrebbe svolta Giancarlo da solo. Mi chiamò quando era convinto di averla finita, ma ancora non l'aveva provata. Aveva scritto direttamente il codice senza fare prima i test.

Abbiamo scritto il test assieme ed anche per questo è stato necessario un noioso debugging. Addirittura ci siamo accorti che il passo 2 (quello precedente) non era completo ed il test relativo (che aveva scritto Giancarlo da solo) non si accorgeva di ciò, mentre l'inconsistenza saltava chiaramente fuori nel test per il passo 3 che doveva rielaborare il risultato del passo precedente.

I successivi due passi li abbiamo scritti assieme, implementando prima i test, poi il codice. Non è filato tutto liscio come nel primo passo, soprattutto a causa della difficoltà nello scrivere il test per la quinta parte che era abbastanza complicata e difficile da testare.

La difficoltà nasceva poiché noi spostavamo i file html in una directory il cui nome dipendeva dalla data del giorno in cui eseguivamo il tool ed era molto difficile nei test ricrearci tali nomi che dipendevano anche da vari altri parametri non facilmente accessibili dall'esterno del codice (e dunque dal test).

Siamo infine riusciti a sfruttare in un modo furbo i dati nel database che rappresentavano tali file.

Arrivati praticamente alla fine, Giulio ci ha modificato le specifiche aggiungendo altre funzionalità e modificando le precedenti.

Abbiamo prima completato il nostro lavoro facendo passare anche il

test funzionale e abbiamo salvato tutto sul repository.

Avevamo intanto indicato a Giulio che ci sarebbero serviti altri 4 giorni di lavoro per soddisfare le nuove richieste.

Introdotte tutte le modifiche, siamo passati a controllare quali test non passavano. Alcuni test li abbiamo corretti per rispecchiare le nuove richieste. Ci siamo stupiti come in solo mezza giornata siamo riusciti a correggere il codice scritto fino a quel momento. Eravamo convinti che ci fossero ancora tanti piccoli particolari che ancora non avevamo considerato e corretto, ma i test ci indicavano chiaramente di no e che il codice era pronto per ricevere l'ultima funzionalità.

Scriviamo il test, poi il codice ed in un altro giorno di lavoro il nostro tool era pronto.

Come ultima cosa Giancarlo ha proposto di realizzare un test sul funzionamento generale ora che avevamo le idee più chiare su tutto e mi disse che a spingerlo era stato un dubbio che aveva.

Scriviamo il test e non passa! La colpa era mia: precedentemente avevo convinto Giancarlo dell'inutilità di eseguire un passo che a prima vista sembrava ridondante. Lì per lì Giancarlo si era trovato d'accordo con me. Ora però il test che aveva scritto dimostrava inequivocabilmente che il nostro tool si sarebbe trovato in difficoltà se per qualche causa fosse stato interrotto durante la quarta fase e successivamente riavviato dall'inizio.

Un'altra mezza giornata per fissare il problema e la seconda parte era completata con due giorni di anticipo sulle nostre previsioni.

Abbiamo provato ad immaginarci altre cause di malfunzionamento, ma non ci sono venute in mente e dunque abbiamo consegnato il tool a Giulio.

Dovevamo ancora imparare a stimare meglio i tempi di realizzazione, ma avevamo compreso chiaramente l'utilità di lavorare assieme e di costruire i test.

Nella seguente tabella riassumo i tempi impiegati, relativi a ciascuna

fase della realizzazione del tool.

	Previsioni	**Tempo impiegato e tipo di lavoro**
Analisi	4 giorni	Mezza giornata in pair programming
I fase		Mezza giornata abbondante in pair p.
II fase		1 giorni intero (di cui mezzo in pair p.)
III fase		2 giorni (di cui più di uno e mezzo in pair p.)
IV fase		Mezza giornata in pair p.
V fase		1 giorno e mezzo in pair p. (quasi un giorno per sistemare il test)
Modifiche alle specifiche	4 giorni	2 giorni in pair programming

Con una settimana circa di lavoro WebFetcher era pronto per andare in produzione. L'abbiamo subito passato alla redazione che l'ha provato e ci ha successivamente suggerito altre modifiche e richiesto nuove funzionalità.

Una così rapida retroazione ci ha permesso di raffinare alcuni particolari, ma soprattutto ci ha assicurato che il lavoro svolto fino a quel momento era un buon lavoro. La fiducia acquisita ci ha permesso di continuare con maggior coraggio nelle successive fasi.

Ancora oggi WebFetcher è in continua evoluzione per rispecchiare le nuove esigenze della redazione e per diventare uno strumento sempre più potente sostituendosi all'uomo nei compiti più noiosi.

13. Organizzazione del lavoro.

Negli esperimenti descritti nei paragrafi precedenti non ho preso in considerazione alcuni aspetti della programmazione estrema che sono comunque importanti.

Qui di seguito farò riferimento alle pratiche, non strettamente legate alla stesura del codice, che servono per avere una completa implementazione di XP.

Il nostro gruppo di sviluppatori è formato da 6 persone. Lavoriamo in un ambiente aperto in modo che sia facile comunicare e chiedere aiuto nei momenti di bisogno.

Abbiamo 6 scrivanie abbastanza ampie da permettere a due persone di lavorare in due sullo stesso computer. All'occorrenza è possibile trovarsi in tre davanti ad uno stesso schermo senza sacrificarsi troppo.

Tra noi 6 Giulio esercita un ruolo particolare. È il responsabile del settore sviluppo software di ibn ed in un certo senso rappresenta il manager che svolge il ruolo di assistente per i programmatori. Spesso lavora in coppia con un altro programmatore, ma difficilmente segue un processo dall'inizio alla fine. Piuttosto cerca di essere a conoscenza di quali siano i processi attualmente in corso.

Coordina le parti di uno stesso progetto che devono interfacciarsi. Non gli serve scendere nel dettaglio: a volte è sufficiente un colloquio con i diversi sviluppatori.

Circa sei mesi fa, quando a sviluppare software erano solo lui e Stefano, non era necessario avere un'assistente, ma ora che siamo in 6 la sua presenza è essenziale per non smarrirci.

Tiene sotto controllo anche il progredire delle applicazioni, considerando anche aspetti quale la comunicazione, il rispetto dei principi della programmazione, la scelta della tecnologia.

Oltre a noi 6 nella sede in cui lavoriamo c'è anche Enrica. Il suo lavoro è fare da tramite tra noi sviluppatori e gli utenti. È essa stessa un utente

e non ha alcune basi di programmazione.

Il suo scopo è quello di utilizzare l'applicazione per svolgere i suoi lavori e segnalarci tempestivamente i problemi. È sempre presente come on-site customer ed è a lei che ci rivolgiamo quando non siamo sicuri su ciò che stiamo implementando. Oltre a questo raccoglie i suggerimenti e le lamentele di tutti gli altri utenti presentandoceli.

Rimane spesso a contatto con gli sviluppatori ed è sempre presente quando si devono pianificare nuove applicazioni: in gran parte è lei che decide come devono essere fatte le varie interfacce grafiche spiegandoci quali sono i bisogni degli utenti.

Naturalmente il tempo che spreca per aiutarci è una piccola parte del tempo che passa in ditta. Normalmente riesce ad eseguire il suo lavoro come qualsiasi altro membro della redazione.

All'interno della stanza in cui lavoriamo c'è anche una zona appartata costantemente rifornita con cibo o piccoli stuzzichini in cui ci si può spostare quando si vuole fare una pausa durante le intense ore passate a programmare.

Esiste anche un tavolo attorno a cui possiamo riunirci per decidere le strategie e per discutere dei problemi comuni.

Nella nostra azienda non applichiamo la pianificazione proprio come prevede XP. Poiché siamo solo 6 non sentiamo la necessità della pianificazione iterativa. Per comunicare le idee usiamo delle semplici metafore ed a folte dei rapidi schemi UML.

14. Conclusioni.

Programmazione estrema (XP) è stata ideata e sviluppata per affrontare le esigenze specifiche dello sviluppo del software condotto da piccoli gruppi di lavoro, a fronte di requisiti vaghi e mutevoli. Questa nuova metodologia *leggera* mette in discussione molte assunzioni convenzionali.

Si tratta di un approccio nuovo, che attacca in particolare alcuni paradigmi dello sviluppo, fra i quali il presupposto secondo cui il prezzo delle modifiche salga vertiginosamente nel corso del tempo.

Sotto certi aspetti si presenta come una proposta rivoluzionariamente innovativa, che scardina i principi sui quali, ormai da più trent'anni, si basa l'insegnamento e la pratica dello sviluppo del software.

Per questo motivo è anche fortemente criticata e vengono sottolineati con forza i suoi limiti.

A prima vista potrebbe sembrare un ritorno a una codifica disordinata e non documentata, che può essere efficace per qualche tempo, ma poi porta a sistemi da incubo, la cui manutenzione è virtualmente impossibile. In realtà, XP è un processo molto strutturato che mantiene un forte controllo sulla qualità e modificabilità del software prodotto.

XP è una disciplina che cerca di deferire le decisioni importanti finché non siano effettivamente necessarie.

Ci si chiede se sia pensabile applicare XP all'interno di strutture già organizzate ed adeguate alle richieste dettate dalle metodologie standard, con i relativi formalismi e rigide procedure operative.

Sicuramente ci sono condizioni che impediscono ad XP di avere successo. Se i livelli manageriali pretendono di avere specifiche chiare e complete prima di iniziare a sviluppare software, oppure una lunga fase di test finali prima di entrare in produzione, allora vengono direttamente minati i presupposti di XP: cicli rapidi e veloce retroazione.

Quello che questa nuova metodologia propone è di abbattere i cronici ritardi, gli eccessivi costi e la dubbia qualità che si presentano nella

maggior parte dei progetti software, cercando di mantenere bassa l'entropia dei sistemi software durante tutto il ciclo di sviluppo e manutenzione.

XP non è per tutti. Comunque la maggior parte dei progetti sviluppati sotto la crescente pressione di ambienti legati ad Internet possono ampiamente beneficiare di questo tipo di approccio. Anche aziende giovani e con poca esperienza possono proficuamente usufruire del supporto di una metodologia come quella proposta da XP.

Molte persone si sono accorte di avere già applicato alcuni suggerimenti presenti in articoli su XP ed hanno apprezzato la formalizzazione data da Kent Beck e dai suoi seguaci nella stesura di una coerente metodologia.

Ognuna delle tecniche dell'XP non è nuova, ed è stata proposta e applicata in passato. La novità sta nell'averle messe tutte insieme, mirando alla massima sinergia e imponendo che tutte siano applicate senza compromessi (di qui il nome "estrema").

XP dimostra che, introducendo nel processo di sviluppo dei principi e delle regole coerenti, la produzione di ottimo software può essere economica, efficiente e addirittura divertente e appagante.

L'applicazione di tutte le tecniche dell'XP porta alla costruzione di sistemi intrinsecamente molto semplici e puliti, la cui complessità è minima, e viene costantemente ridotta tramite la ristrutturazione.

Idealmente, si sviluppano sistemi il più semplici possibile, e il costo delle modifiche resta costante, o almeno cresce molto meno che nel caso degli approcci tradizionali. Gli enormi risparmi promessi stanno ovviamente attirando l'attenzione di tutto il mondo sull'XP: esiste una mailing list dedicata ad estreme programming che conta più di 2000 iscritti e durante questo anno ha ricevuto più di 16000 messaggi.

In conclusione, XP è una nuova tecnica di produzione del Software molto promettente. Certamente, come per tutte le cose nuove ed efficaci, non è facile adottarla e applicarla correttamente, ma se mantiene anche solo una parte delle promesse di produttività e qualità del software prodotto, vale la pena provarla.

Appendice A: WebObjects.

WebObjects della Apple, Next, è l'ambiente di programmazione che utilizziamo presso IBN.

È una piattaforma per sviluppare applicazioni web client-server e per supportare pubblicazioni dinamiche.

Per ogni utente che si collega è possibile creare una sessione personale che contenga i dati e le richieste e creare pagine html in modo dinamico.

Le pagine vengono create a partire da template contenenti riferimenti ad oggetti che hanno il compito di produrre frammenti di html che costituiranno la pagina letta dal cliente.

Apple è stata la prima a proporre strumenti come WebObjects che integrino le applicazioni web con possibilità di connettersi ad un database e generare html dinamicamente. Negli ultimi anni sono apparsi strumenti analoghi come l'**ASP** della Microsoft o **PHP** opensource.

WebObjects sfrutta pienamente anche la programmazione ad oggetti per avere tutti i vantaggi come la modularità, l'information hiding, il polimorfismo, ...

WebObjects continene una importante libreria che si chiama EOF (Enterprise Objects Framework). Questa è fondamentalmente una astrazione che rende trasparente l'accesso ai database.

Sono utilizzati dei modelli per poter accedere alle tabelle tramite oggetti. Vengono eliminate le fastidiose stringhe SQL inline. Infatti le query sono generate automaticamente dal framework partendo da qualifier (filtri), cioè oggetti che vengono generati in una precisa parte dell'applicazione e da un singolo oggetto e non sono sparsi all'interno di tutta l'applicazione.

È così possibile modificare il modello del database aggiungendo o eliminando attributi alle tabelle, modificando solo l'oggetto che rappresenta tale tabella e che ha il compito di costruire e restituire i

qualifier per accedere ai dati. Chi utilizza i qualifier non deve preoccuparsi se vengono modificati dettagli interni. I qualifier possono essere facilmente concatenati tramite and ed or costruendo nuovi qualifier e dunque nuovi oggetti, rendendo tutto estremamente modulare.

Come ho già detto, ogni tabella ha il suo corrispondente oggetto che la rappresenta. Ogni suo attributo può essere reso accessibile tramite due metodi dell'oggetto (un getter per leggerlo ed un setter per scriverlo).

EOF può gestire diversi database di diverse marche lasciando il programma inalterato. L'accesso diretto al database è delegato ad adaptor personalizzati per Oracle, Sybase, Informix, database che aderiscono a ODBC server.

WebObjects è stato sviluppato utilizzando il linguaggio Objective C, ma ultimamente Apple sta facendo grossi sforzi per trasportare tutto in java. Attualmente sono supportati entrambi i linguaggi, ma la tendenza è privilegiare java, data la sua diffusione e la sua costante crescita.

Appendice B: Objective C.

È un linguaggio object-oriented, costruito da Next a partire dal linguaggio C ANSI, per l'ambiente di sviluppo OpenStep.

Simile a SmallTalk, utilizza il concetto di messaggio: noi inviamo dei messaggi agli oggetti. Non importa se l'oggetto possiede il metodo; il messaggio gli arriva ugualmente, poi magari è l'oggetto che solleva l'eccezione se non sa a chi delegare il messaggio.

È stato scelto il C per mantenere la compatibilità con i programmi già esistenti. È un linguaggio semplice da imparare. La sua sintassi non è complessa ed è sufficiente leggere un libretto di un centinaio di pagine per conoscere tutto quello che è necessario sapere per utilizzarlo.

Si possono usare le normali notazioni del C, ma ne sono state aggiunte di nuove per rendere il codice più leggibile.

Per esempio i metodi possono essere invocati usando la seguente notazione:

[puntatoreOggetto metodo:primoParametro nomeSecondoParametro:secondoParametro];

Risulta in tal modo più semplice capire quale tipo di parametri sono passati al metodo, soprattutto quando il numero di parametri non è piccolo.

Objective C supporta concetti come dynamic typing, dynamic binding e dynamic loading. Non esiste una stretta tipizzazione ed i controlli vengono fatti runtime per rendere agevole il polimorfismo. Addirittura è possibile modificare le classi aggiungendo categorie dall'esterno anche dopo che il programma è stato avviato.

Naturalmente tutte queste sono importanti scelte che caratterizzano fortemente il linguaggio, ma non è nello scopo di questo trattato giudicarle.

In Objective C si usa il costrutto protocol per creare qualcosa di simile alle interface java.

Come ho già accennato esiste il costrutto category per poter aggiungere i metodi ad una classe senza essere costretti a derivare una

sottoclasse. Possono essere usate per aggiungere funzionalità a classi di cui non possediamo il codice, per gestire e raggruppare i metodi di classi complesse e per compilarle in modo incrementale, per configurare una classe in modo differente per applicazioni diverse senza la necessità di mantenere due differenti versioni dello stesso codice.

Per ogni oggetto creato viene prima invocata la alloc per allocare lo spazio in memoria e di seguito la init (o una sua variazione). Alla fine l'oggetto viene distrutto tramite una release.

I metodi possono essere di classe ed in tal caso sono preceduti dal segno +, oppure di istanza e dunque preceduti dal segno -.

Esiste una libreria, denominata Fondation, che contiene vari oggetti di uso comune come stringhe (NSString), array (NSArray), dictionary (NSDictionary)...

Spero che questi pochi elementi di Objective C possano permettere una agevole lettura del codice che viene presentato nelle seguenti appendici.