Lezione 19
Extreme Programming e JUnit

Vittorio Scarano
Corso di Programmazione Distribuita (2003-2004)
Laurea di I livello in Informatica
Università degli Studi di Salerno

Organizzazione della lezione

- Extreme Programming (XP)
  - i valori di XP
  - le *core practices* (prassi consolidate) di XP
  - alcuni commenti finali
- Test-Driven Development
  - il debugging
- JUnit
  - motivazioni
  - caratteristiche
  - conclusioni e un esempio con Eclipse

Struttura del Seminario

- **Extreme Programming (XP)**
  - **i valori di XP**
  - le *core practices* (prassi consolidate) di XP
  - alcuni commenti finali
- Test-Driven Development
  - il debugging
- JUnit
  - motivazioni
  - caratteristiche
  - conclusioni e un esempio con Eclipse

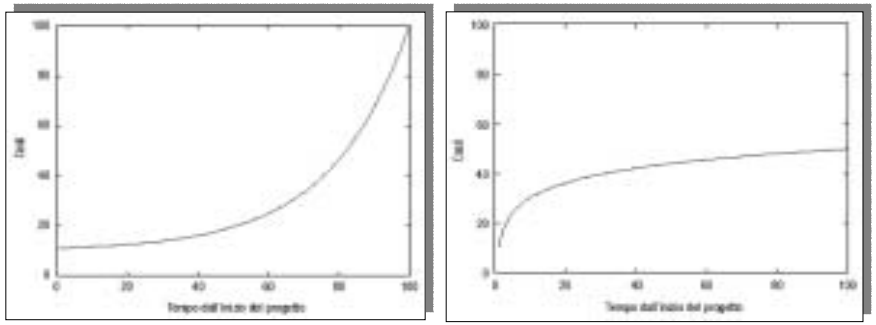
Extreme Programming

*Extreme programming è una disciplina di sviluppo software basata sui valori di **semplicità**, **comunicazione**, **testing** e **coraggio***

- Wow!
- L'obiettivo principale è quello di evitare "semilavorati" durante la produzione, a parte quelli relativi alla realizzazione dei programmi
- Insomma, niente analisi, diagrammi, dettagliate specifiche formali, puntigliosa documentazione …
- Progettare software deve essere come guidare una macchina: minima interazione per direzione, frenate se necessario

I valori di XP: semplicità

- "I programmatori XP fanno la cosa più semplice che possa risolvere il problema"
  - non costruire per il futuro, la OOP permette modularità e facilità di modifiche in corso d'opera
- "I programmatori XP lasciano il sistema nella condizione più semplice possibile"
- Modifica del tradizionale modello di costi/modifica



I valori di XP: comunicazione

- Tra cliente e sviluppatore:
  - i clienti sono parte del progetto (meeting etc.)
  - *user stories*
- Identificazione di 4 variabili:
  - *scope* (cosa è fatto e cosa rimane da fare)
  - *qualità* (correttezza ed altre misure)
  - *risorse* (personale, materiale, spazio)
  - *tempo* (durata del progetto)
- Date tre variabili la quarta (tempo) può essere trovata
- Comunicazione tra sviluppatori
  - niente team separati
  - lavoro in coppie
  - *openspace* con computer liberamente a disposizione
  - meeting in piedi (max 15 minuti)

I valori di XP: testing

- Critico mantenere la qualità del prodotto
- Principio base:
  - quando si aggiungono nuove caratteristiche al sistema, non vogliamo che parti del sistema (che prima funzionavano) non funzionino più
- Test che devono essere superati al 100%
- Testare ogni singola parte del sistema in maniera strutturata (*unit testing*)
- Test forniti dal cliente (*functional testing*)
  - corrispondente alle user stories

I valori di XP: coraggio

Semplicità + comunicazione + testing = aggressività

- Il testing aiuta a mantenere il sistema semplice
- Il sistema semplice può essere facilmente modificato
- La comunicazione continua permette di sapere chi sta facendo e cosa
- Quindi le modifiche necessarie possono e devono essere fatte subito
  - una regola non scritta del programmatore non-XP:
  - "If it ain't broken, ain't fixin"… "Se non è rotto, non si aggiusta"
  - Il testing continuo permette di scoprire se modifiche introducono bug

Struttura del Seminario

- **Extreme Programming (XP)**
  - i valori di XP
  - **le *core practices* (prassi consolidate) di XP**
  - alcuni commenti finali
- Test-Driven Development
  - il debugging
- JUnit
  - motivazioni
  - caratteristiche
  - conclusioni e un esempio con Eclipse

Le 13 *core practices* di XP

- Whole Team
- Planning Game
- Small Releases
- Customer Tests
- Simple Design
- Pair Programming
- Test-Driven Development
- Design Improvement
- Continuous Integration
- Collective Code Ownership
- Coding Standard
- Metaphor
- Sustainable pace

1 – Whole Team

- Un unico ambiente per tutto il team
- Esiste un rappresentante del committente (business)
  - fornisce requisiti, setta le priorità, e guida il progetto
  - meglio se veramente è il cliente stesso
- Il team deve includere dei Tester
  - aiutano il cliente a definire i test di accettazione del prodotto
- Deve esserci un Manager (gestione delle risorse)
- ***Nessun ruolo è predefinito***
  - ognuno può scambiare ruolo
- Non ci sono specialisti

2 – Planning Game

- Enfasi sulla guida del progetto rispetto alla accurata predizione di quello che accadrà
- Due fasi:
  - release planning (requisiti, stima difficoltà, piano approssimativo del progetto)
  - iteration planning
    - ogni due settimane si fornisce un programma funzionante
    - durante la iterazione, il cliente fornisce i requisiti per la prossima iterazione
    - i requisiti servono a fornire predizioni e piani del progetto più precisi
- Il progresso risulta visibile ogni 2 settimante
- Qualsiasi errore viene evidenziato al più presto

3 – Customer test

- Test automatici di accettazione
- Il team definisce, costruisce ed usa questi test
- Importante che i test siano automatici
  - in caso di pressione (scadenze da rispettare) la prima cosa che viene eliminata (di norma) è proprio il testing
  - come "*spegnere la luce quando la notte si fa scura*"
- I test del cliente sono trattati come i test del programmatore:
  - una volta che vanno a buon fine, vanno mantenuti in quello stato
- Il progetto fa solamente passi avanti, mai indietro
  - *"Se avanzo seguitemi…"* ☹

4 – Small Releases

- Release
  - sia interne (alla fine di ogni iterazione)
  - sia esterne (agli utenti finali) il più spesso possibile

5 – Simple Design

- IL progetto deve essere semplice
- La progettazione non separata dalla codifica
- Processo incrementale

6 – Pair Programming

- Programmazione fatta da due persone
  - uno solo scrive (alla volta) al computer
- Il codice viene continuamente rivisto da due persone
- Le coppie si scambiano di ruolo spesso
- La composizione delle coppie si cambia spesso
- Obiettivi:
  - migliorare la qualità
  - comunicare e disseminare la conoscenza del sistema
  - training on the job estremamente efficiente
- Pratica che sembra poco pratica ma che (una volta provata) diventa (sperimentalmente) preferita dal 90% dei programmatori

"du' is meglio che one?"

- A fronte di un degrado minimo nelle prestazioni…
- Si ottiene codice meglio scritto (correttezza)…
- … e di meno linee (compattezza)

7 – Test- Driven Development

- Ossessione con il feedback (test infected software!)
- Ciclo di produzione (brevissimo)
  - aggiunta di un test (che non funziona)
  - scrittura di codice che può risolvere
- Struttura automatica per il test continuo di tutti i test sviluppati finora per il software prodotto

8 – Design Improvement

- Refactoring: eliminare duplicazioni, aumentare la coesione del codice e decrementare la dipendenza
- Inserita nel ciclo di produzione del test-driven development:
  - test, scrittura codice, improvement…

9 – Continuous Integration

- Il sistema viene mantenuto integrato in ogni istante.
- Quindi non ci sono parti sviluppate isolatamente
- Build completa del software: 8-10 volte ***al giorno!***
- In caso di integrazione non frequente:
  - il team non è pratico dei problemi e può generare problemi
  - problemi di "code freeze": non si può aggiungere del nuovo

10 – Collective Code Ownership

- Ogni coppia di programmatori può lavorare su ogni parte del progetto
- Maggiori attenzioni al codice da parte di più persone
- I problemi possibili sono superati con le attività di test e con il pair programming

11 – Coding standard

- Utilizzo di standard per facilitare modifiche da tutti

11 – Metaphor

- Uso di metafore naturali per il programma
  - sw agents come "api che tornano portando polline per il cibo.."
- O almeno (se sprovvisti di capacità poetiche ☺)
  - uso di una terminologia chiara e definita per le parti e le funzioni del sistema

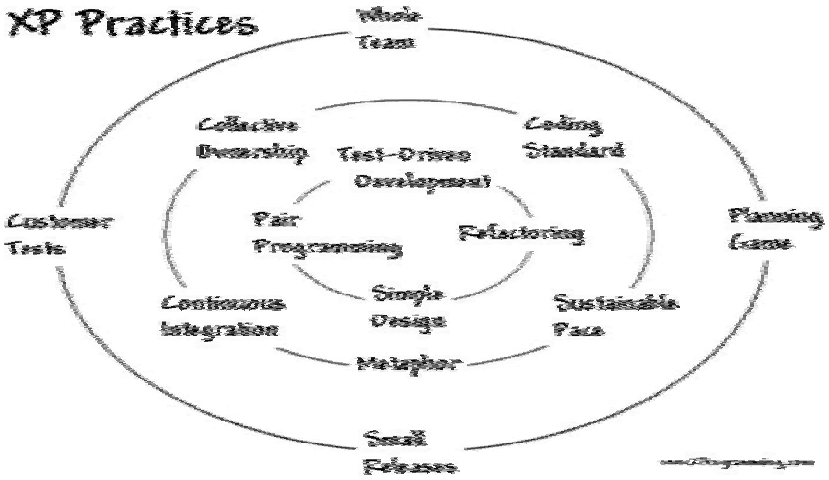
12 – Sustainable pace

- Niente lavoro straordinario *(40 hours week):*
  - si lavora a passo veloce ma in modo da poter sviluppare codice senza la stanchezza di lavoro extr
    - Hanno parlato con i sindacati? ☺

Struttura del Seminario

- **Extreme Programming (XP)**
  - i valori di XP
  - le *core practices* (prassi consolidate) di XP
  - **alcuni commenti finali**
- Test-Driven Development
  - il debugging
- JUnit
  - motivazioni
  - caratteristiche
  - conclusioni e un esempio con Eclipse

Una visione delle core practice di XP



Una visione critica…

- XP utile in sistemi di dimensione piccolo-media
  - fortemente caratterizzati da requisiti poco definiti e instabili
- Necessario stemperare l'estremismo di XP per sistemi di grandi dimensioni
  - per i quali ci vuole una specifica formale dei requisiti ed una analisi formale del prodotto
- Riconoscendo, comunque, il rischio che, a volte, ci si concentra sulla documentazione e non sulla produzione di codice di qualità
- Alcune pratiche di indubbio successo:
  - pair programming, ***test-driven development***

Struttura del Seminario

- Extreme Programming (XP)
  - i valori di XP
  - le *core practices* (prassi consolidate) di XP
  - alcuni commenti finali
- **Test-Driven Development**
  - **il debugging**
  - semplici tecnologie Java per il debugging: assert
- JUnit
  - motivazioni
  - caratteristiche
  - conclusioni e un esempio con Eclipse

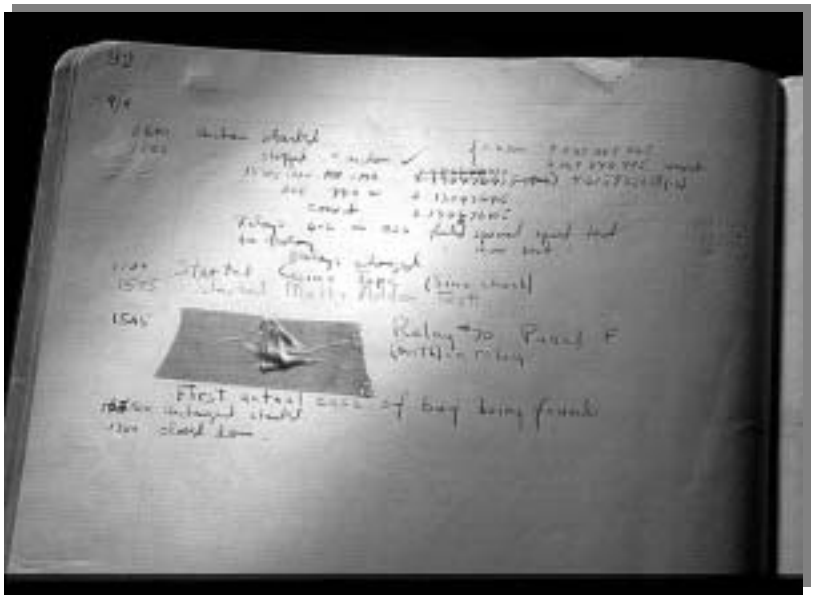
Test-driven Development

- Una maniera per testare il software con facilità
- Permette allo sviluppatore di testare il proprio programma mentre lo progetta e lo modifica
  - in maniera continua, facile e naturale
- Spendere metà del proprio tempo per fare testing
  - serve a risparmiare molto più tempo in debugging
- Secondo Karl Popper:
  - una teoria scientifica è tale se è confutabile
  - cioè se esiste almeno un esperimento il cui fallimento può provare la falsità della teoria

L'incubo del programmatore: il debugging

- L'origine del termine…
- Per controllare il malfunzionamento del calcolatore elettro-meccanico Mark II Aiken Relay Calculator ad Harvard
  - una falena (tarma) fu trovata in un relay (9/9/1945)
  - e gli operatori dell'epoca scrissero sul "giornale di bordo" *"First actual case of a bug being found!"*
- Il termine era comunque già in uso:
  - Edison a proposito di circuiti elettrici intorno al 1870
  - a proposito di qualsiasi difetto in macchinari

Le prove… ☺



Struttura del Seminario

- Extreme Programming (XP)
  - i valori di XP
  - le *core practices* (prassi consolidate) di XP
  - alcuni commenti finali
- Test-Driven Development
  - il debugging
- **JUnit**
  - **motivazioni**
  - caratteristiche
  - conclusioni e un esempio con Eclipse

JUnit

- JUnit è un framework per scrivere test
  - scritto da Erich Gamma
    - famoso per i Design Pattern (della Gang of Four)
  - e da Kent Beck
    - metodologia Extreme Programming
- Usa la reflection di Java
- Supporta il programmatore per :
  - la definizione di test e di suite (insiemi) di test
  - formalizzazione di requisiti
  - scrivere e debugging di codice
  - integrare codice e essere sempre pronto alle release
- Non ancora parte di Sun
  - compreso però in BlueJ, JBuilder, Eclipse, etc.
- Disponibile in altri linguaggi (xUnit, per ogni x)

Terminologia

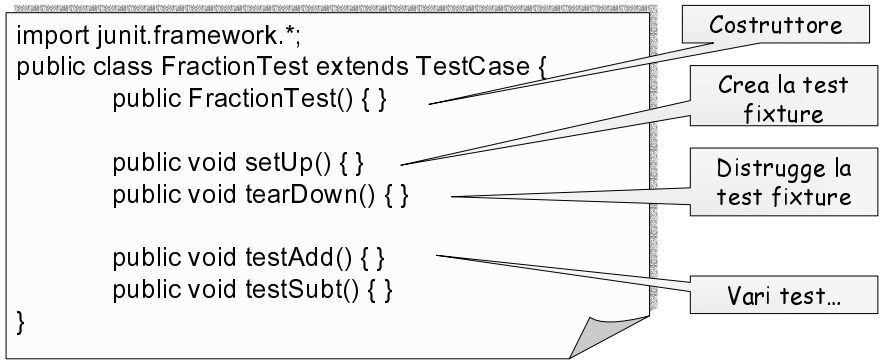
- Una *test unit* controlla una singola classe
  - mentre i singoli test controllano le risposte di un singolo metodo su uno specifico input
- Una *test suite* è una collezione di test unit
- Un *test runner* è un programma che esegue e collezione (e riporta) i risultati
- Un *test fixture* prepara dati (=oggetti) e primitive (= metodi) che servono per le test unit
  - se sto testando una classe Employee, ho bisogno di creare un oggetto Employee per poterlo testare in tutte i test

Struttura del Seminario

- Extreme Programming (XP)
  - i valori di XP
  - le *core practices* (prassi consolidate) di XP
  - alcuni commenti finali
- Test-Driven Development
  - il debugging
  - semplici tecnologie Java per il debugging: assert
- **JUnit**
  - motivazioni
  - **caratteristiche**
  - conclusioni ed un esempio con Eclipse

Struttura di una test unit di JUnit

- Sia data una classe da testare: Fraction
- Allora si crea una test unit chiamata: FractionTest



- Mediante introspezione, per ogni metodo testXXX()
  - setUp()
  - testXXX()
  - tearDown()

I metodi utilizzabili nei test (1)

- In un test (singolo metodo di una test unit)
  - si chiama il metodo da testare e si ottiene il risultato
  - si controlla con una assert se il risultato è corretto
- Questi passi possono essere ripetuti
- I metodo di assert che fornisce JUnit
  - lanciano una eccezione AssertionFailedError se la assert fallisce
- Alcuni metodi:
  - static void assertTrue(boolean test)
  - static void assertTrue(String message, boolean test)
  - static void assertFalse (boolean test)
  - static void assertFalse(String message, boolean test)

I metodi utilizzabili nei test (2)

- Altri metodi:
  - static void assertEquals(*expected, actual*)
  - static void assertEquals(String msg, *expected, actual*)
    - dove *expected* e *actual* possono essere:
      - dello stesso tipo primitivo (int, boolean, etc.)
      - Object
  - static void assertSame(Object expected, Object actual)
  - static void assertSame(String msg, Object expected, Object actual)
  - static void assertNotSame(Object expected, Object actual)
  - static void assertNotSame(String msg, Object expected, Object actual)

I metodi utilizzabili nei test (3)

- Altri metodi:
  - static void assertNull(Object obj)
  - static void assertNull(String msg, Object obj)
  - static void assertNotNull(Object obj)
  - static void assertNotNull(String msg, Object obj)
  - fail ()
  - fail (String msg)
    - causa una generazione di AssertionFailedError
    - utile per test complessi e per generare una eccezione quando il metodo non genera una eccezione come richiesto

L'esempio della classe CounterTest

CounterTest.java

```
import junit.framework.*;
public class CounterTest extends TestCase{
    Counter c1;

    public CounterTest() {}

    protected void setUp() {
        c1=new Counter();
    }

    protected void tearDown() { }

    public void testIncrement() {
        assertTrue(c1.increment()==1);
        assertTrue(c1.increment()==2);
    }

    public void testDecrement() {
        assertTrue(c1.decrement==-1);
    }
}
```

- Per ogni metodo di test, setUp() cassegna il campo c1 con un nuovo Counter
- tearDown() vuoto
- Due incrementi successivi
- Un decremento

L'esempio della classe Counter

Counter.java

```
import junit.framework.*;
public class Counter{
    int count = 0;

    public int increment() {
        return ++ count;
    }

    public int decrement() {
        return - - count;
    }

    public int getCount() {
        return count;
    }

}
```

- Campo intero
- metodi di incremento e decremento
- Metodo accessore al campo count

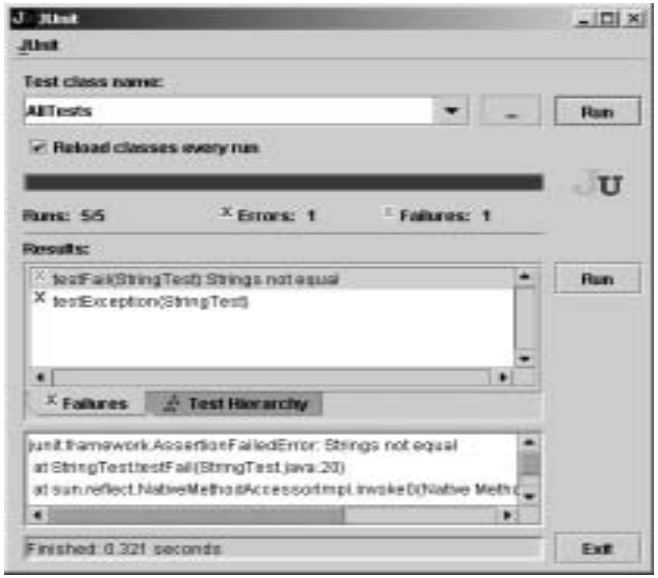
La esecuzione di CounterTest

CounterTest.java

```
import junit.framework.*;
public class CounterTest extends TestCase{
    Counter c1;

    public CounterTest() {}

    public static void main (String[] ) {
        junit.swingui.TestRunner.run(
            CounterTest.class);
    }

    // …. il resto dei metodi
    protected void setUp() {
        c1=new Counter();
    }

    // etc. etc.
```

- Chiamata della GUI per eseguire il test
- Possibile usare anche TUI
  - Text Usual Interface ☺

La esecuzione del test



Struttura del Seminario

- Extreme Programming (XP)
  - i valori di XP
  - le *core practices* (prassi consolidate) di XP
  - alcuni commenti finali
- Test-Driven Development
  - il debugging
- **JUnit**
  - motivazioni
  - caratteristiche
  - **conclusioni ed un esempio con Eclipse**

Conclusioni…

- Un ambiente automatico e di facile utilizzo
- Non è la panacea di tutti i mali:
  - non rappresenta una metodologia ma una tecnologia
    - uno strumento usato male non raggiunge il suo scopo
- Ad esempio, non si può usare (o si può usare limitatamente) in certi contesti come
  - programmazione multi-thread
  - programmazione su rete
  - funzionalità di interfacce utente
- Demo con Eclipse: