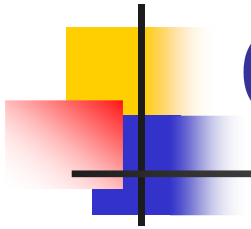
Tecniche di Programmazione avanzata

Corso di Laurea Specialistica in Ingegneria Telematica
Università Kore – Enna – A.A. 2009-2010

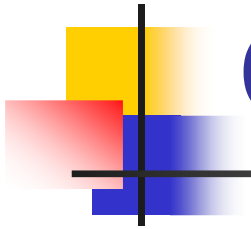
Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

Testing, debugging, logging del codice



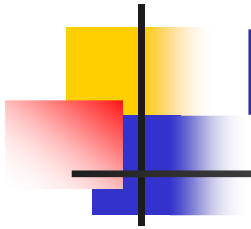
Correttezza

- Un obiettivo di **qualità a breve termine**: **la correttezza**
  - il software deve implementare interamente e correttamente le specifiche
  - ovvero realizzare correttamente i casi d'uso
- Le **tecniche fondamentali** per le verifiche di correttezza sono due:
  - i **test**
  - il **debugging** (correzione)
- Notare che:
  - esistono **altre proprietà da verificare**, oltre la correttezza, per assicurare la qualità del software (ad esempio l'affidabilità)
  - Esistono anche **altre tecniche** per assicurare in generale le proprietà del software (ad esempio, il benchmarking)



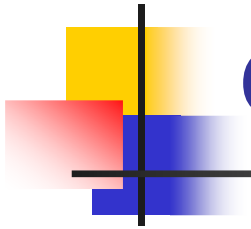
Correttezza

- **Verificare il codice con i test**
  - il gruppo di sviluppo deve mettere a punto un **"piano dei test"** che deve essere eseguito con regolarità
  - spesso c'è un gruppo di lavoro appositamente destinato a effettuare test
  - Il piano dei test deve verificare la robustezza del codice
  - Correggere il codice quando viene rilevato un errore logico ("bug") attraverso i test o attraverso prove interattive
- **Due tecniche fondamentali per il debug:**
  - utilizzo di un debugger
  - utilizzo di stampe di debug



Manutenibilità

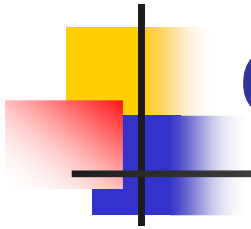
- Ma esiste un'altra prospettiva. Un obiettivo di **qualità a lungo termine: la manutenibilità**
- La regola n. 1 del programmatore
  - "le specifiche cambiano con il tempo" e i sistemi software si devono adattare
  - la possibilità di adeguare facilmente e rapidamente il codice alle nuove specifiche è una caratteristica essenziale
- Di conseguenza
  - la qualità deve essere perseguita nel tempo mantenendo alto il livello del codice anche in caso di modifica delle specifiche
- "Adattarsi ai cambiamenti"
  - una massima dell'eXtreme Programming
  - essere consapevoli che il software cambia e adottare strumenti adeguati a gestire i cambiamenti mantenendo alta la qualità



eXtreme Programming

L'**Extreme Programming** (**XP**) è una metodologia di ingegneria del software formulata nel 1999, basata su dodici regole:

1. Progettare con il cliente;
2. Test funzionali e unitari;
3. Refactoring (riscrivere il codice senza alterarne le funzionalità esterne);
4. Progettare al minimo;
5. Descrivere il sistema con una metafora, anche per la descrizione formale;
6. Proprietà del codice collettiva (contribuisce alla stesura chiunque sia coinvolto nel progetto);
7. Scegliere ed utilizzare un preciso standard di scrittura del codice;
8. Integrare continuamente i cambiamenti al codice;
9. Il cliente deve essere presente e disponibile a verificare (sono consigliate riunioni settimanali);
10. Open Workspace;
11. 40 ore di lavoro settimanali;
12. Pair Programming (due programmatori lavorano insieme su un solo computer).



eXtreme Programming

James Donovan Wells individua **quattro linee guida dell'XP**:

1. Comunicazione (tutti possono parlare con tutti, persino l'ultimo dei programmatori con il cliente);
2. Semplicità (gli analisti mantengano la descrizione formale il più semplice e chiara possibile);
3. Feedback (sin dal primo giorno si testa il codice);
4. Coraggio (si dà in uso il sistema il prima possibile e si implementano i cambiamenti richiesti man mano).

- Extreme Programming è una metodologia molto famosa, anche se comunque non troppo differente dalle altre. Deve sicuramente la sua fortuna al lavoro dell'autore che ha saputo coglierne gli aspetti positivi e trasmetterli, anche quando i progetti gestiti sono falliti, fra questi il primo in assoluto. D'altronde è Kent Beck stesso ad ammettere questi fallimenti, anzi a considerarli parte integrante della filosofia di fondo della metodologia ed a confermare che di tutte le pratiche di Extreme Programming, la più importante è il carisma del project manager.
- XP is successful because it emphasizes customer involvement and promotes team work.

Manutenibilità

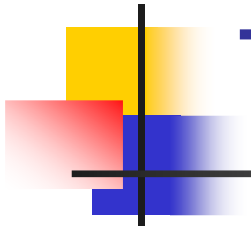
Per garantire la manutenibilità, il test ed il debugging devono evolversi, quindi occorre:

- **Scegliere la tecnica di test** da adottare
- Effettuare in ogni caso **test di regressione**, ovvero testare le funzionalità preesistenti e già collaudate di un software, per assicurare che modifiche al prodotto durante la manutenzione non ne abbiano compromesso la qualità (regressione = verificare se la qualità è regredita)
- Scegliere le metriche per misurare l'efficacia del test. Le più usate:
  - **l'analisi della copertura del codice** ottenuta tramite un profiler, che indica quante volte è stata eseguita (coperta) ogni istruzione durante il test. L'obiettivo è coprire il maggior numero di istruzioni.
  - **il rapporto tra il numero di difetti trovati in un modulo e il numero di istruzioni modificate**. Se si trova che in un modulo sono state modificate molte istruzioni ma sono stati trovati pochi difetti, si può dubitare sull'efficacia del collaudo di tale modulo.
- **Anche il debug** deve essere più strutturato, quindi sono presenti framework per la gestione di breakpoint, watch, stack analysis, step-by-step execution e logging
- **Uniformarsi agli standard** di certificazione di qualità. L'International Software Testing Qualifications Board (www.istqb.org) e' l'ente piu' importante a livello mondiale che si occupa di queste certificazioni.

Tecniche di test: tassonomia

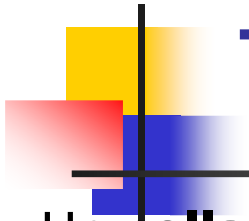
Per scegliere la tecnica di test, ecco una tassonomia:

- **Per modello di sviluppo:**
  - **a cascata**, ossia appena è completata la prima versione del prodotto (molto criticato perché spesso tardivo)
  - **guidato dal collaudo**, che considera il collaudo una parte integrante del prodotto, quindi scriverne i requisiti, l'architettura ecc.
- **Per appartenenza dei collaudatori all'azienda software**: **Alfa** (collaudo interno all'azienda), **Beta** (esterno, agli utenti).
- **Per grado di automazione**: manuale, semi-automatizzato, automatizzato.
- **Per granularità:**
  - **test di modulo**, ossia testando i singoli componenti; i componenti collaudabili di un software sono "moduli" o "unità", con granularità variabile da una singola a migliaia di routine. In OOP, la tipica componente è la classe.
  - **test di sistema**, in cui si opera con una vista di insieme, (anche se i singoli moduli sono corretti, il sistema complessivo potrebbe non esserlo).
- **Per livello di conoscenza delle funzionalità interne:**
  - A **scatola bianca**, in cui il test è effettuato richiamando direttamente le singole routine del software. Richiedendo l'accesso alle routine, tipicamente è un collaudo di modulo. Spesso è automatizzato.
  - A **scatola nera**, in cui si opera solamente tramite l'interfaccia utente, oppure tramite interfacce di comunicazione tra processi
- **Per grado di formalità: da informale a formale.**



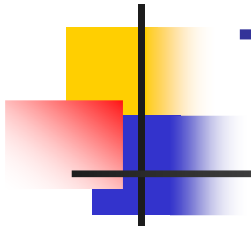
Tecniche di test: tassonomia

- **…per grado di formalità: da informale a formale.**
- Nella **verifica informale**, il programmatore, dopo aver apportato una modifica al software lo manda in esecuzione e verifica interattivamente se il funzionamento è quello atteso. Quando il programmatore è soddisfatto, la nuova versione viene inviata agli utenti e al personale di assistenza ("help desk") come versione Beta. In tale procedimento informale di collaudo, la segnalazione di malfunzionamenti e di nuove versioni non segue un iter ben definito (comunicazioni telefoniche, e-mail, ecc).



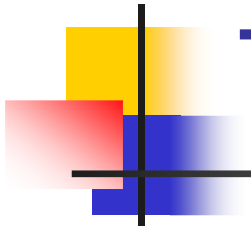
Tecniche di test: tassonomia

- Un **collaudo formale** viene descritto da un "test plan".
- Ci sono **due strategie fondamentali** per organizzare il collaudo: la "batteria di prove" ("test suite"), e lo "scenario di collaudo" ("test scenario"), spesso usati in combinazione (si eseguono test suite in un certo scenario)
  - Una **batteria di prove** è un insieme di collaudi elementari, ognuno dei quali è un "test case". Ogni test case viene descritto da: lo scopo della prova, la sequenza di operazioni necessarie per portare il sistema nelle condizioni iniziali per la prova e per ripristinarlo dopo la prova, le dipendenze con altri test (ad esempio, se una prova consiste nell'aprire un file e chiuderlo, e un'altra prova legge il file, la seconda dipende dalla prima), il risultato atteso ed il tempo massimo per ricevere il risultato.
  - Uno **scenario di collaudo** è un utilizzo realistico del software da collaudare. Mentre le prove di collaudo considerano le funzionalità elementari, ogni scenario prende in considerazione una tipologia di utente e una situazione verosimile e complessa in cui tale utente può trovarsi. Il collaudo di scenario percorre tutti i passi che l'utente percorrerebbe in tale situazione. Il collaudo Beta è di fatto un collaudo di scenario, sebbene informale. Il collaudo di scenario è necessariamente un collaudo di sistema, e tipicamente è manuale o semiautomatico.



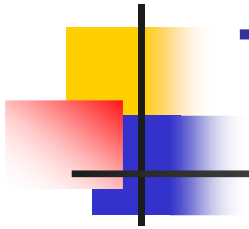
Tecniche di Test: scelte

- In pratica, lo scenario spesso presente è che si trovano errori logici, ossia test in cui il codice si comporta in modo diverso da quello atteso. In questi casi è necessario capire la causa dell'errore correggere il codice e ripetere integralmente l'esecuzione del piano dei test. In questo scenario:
  - eseguire solo **test funzionali (black box)** non fornisce indicazioni immediate sulle cause reali di un comportamento scorretto, bisogna studiare il codice per risalire dal comportamento scorretto al componente responsabile. Per questo motivo, si sceglie spesso di operare prima **test di unità (white box)**.
  - I test sono eseguiti interattivamente e il programmatore è costretto a ripeterli tutte le volte che effettua modifiche. Tale processo e noioso e aperto agli errori, quindi si sceglie ove possibile di **automatizzare** (cosi' è possibile avere **test di regressione** per favorire la manutenibilità).
  - La scelta della **verifica formale** è quasi sempre la migliore, quindi si sceglie anche di seguire una **tecnica guidata dal collaudo**.
  - **Test alfa e beta** sono solitamente entrambi presenti.



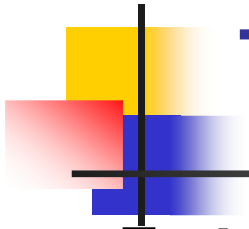
Test di Unità

- Nella programmazione a oggetti l'organizzazione del codice in componenti suggerisce una modalità più raffinata di test oltre ai test funzionali (indispensabili), ossia verificare i singoli componenti in condizioni isolate gli uni rispetto agli altri, quindi **Test di Unità**:
  - test effettuato su una singola classe di componenti (una classe e tutti i suoi oggetti)
  - per quanto possibile in modo isolato rispetto agli altri componenti
  - tipicamente: una serie di test per ciascun metodo dell'interfaccia del componente
- Rispetto ai test funzionali non vengono verificati interi casi d'uso



Test di Unità

- **Scrivere test di unità**
  - un metodo tipico: le classi di test
- Classe di test (Classe di Collaudo)
  - idea: per ciascuna classe da verificare, ne scrivo un'altra di collaudo con un metodo main che chiama i metodi della classe ed effettua stampe dei risultati.
  - la classe di test quindi non implementa un caso d'uso dell'applicazione; il suo obiettivo è quello di verificare la correttezza dei metodi di una classe specifica
  - Non occorre implementare schemi o casi d'uso complessi, resta tuttavia poco realistico (va bene per le prime fasi)
- Per eseguire il test, il programmatore avvia la classe di collaudo e verifica che i risultati delle stampe siano corretti
- si tratta di un passo avanti, ma anche in questo caso il processo è ripetitivo, quindi occorre automatizzare completamente i test

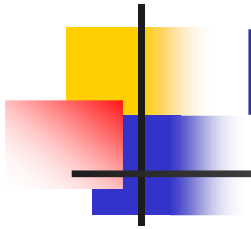


Test di Regressione

- **Test di regressione**
  - la cui esecuzione è automatizzata, quindi test eseguibile rapidamente, ripetutamente, senza intervento interattivo
  - "eseguire i test deve costare quanto schiacciare un tasto"
- Piano di test automatizzato → "batteria" di test di regressione
- Vantaggio di questo approccio
  - è possibile effettuare più facilmente modifiche al codice
  - scoprendo presto eventuali "regressioni" (passi indietro), ovvero errori introdotti dalle modifiche, in modo da poterli eliminare concentrandosi solo sulle porzioni di codice scorrette
  - i test diventano un "paracadute"

Come sono organizzati i test:

- per ogni classe dell'applicazione, una classe di test contenente vari metodi di test
- Un metodo di test è un metodo che utilizza uno o più metodi del componente da verificare sulla base di dati stabiliti dal programmatore e facendo asserzioni sul risultato atteso
- l test è superato se l'asserzione è vera, altrimenti viene tipicamente lanciata un'eccezione

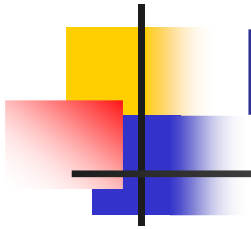


Esempio

- Come esempio di test di regressione e di unità, consideriamo una classe Java (che nel seguito è il linguaggio adottato) che modella il punto geometrico con le sue coordinate. Supponiamo di avere:

  Punto p = new Punto();
  p.setX(10);
  p.setY(20);
- Dopo queste istruzioni posso asserire che

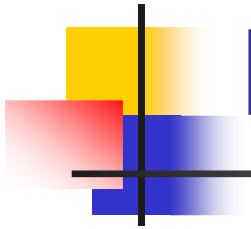
  p.getQuadrante() == 1



Esempio

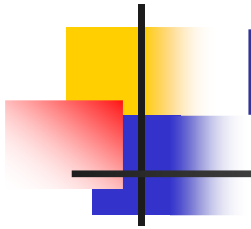
- Un possibile metodo per l'automazione dei test (quindi per la realizzazione dei test di regressione) riceve una stringa (descrizione dell'asserzione) ed un'espressione e verifica che l'espressione sia vera:

```
public void assertTrue(String s, boolean asserzione) {
    If (asserzione) {
        System.out.print(".")        ;
    }
    else {
        throw new AssertionFailedException(s);
    }
}
```



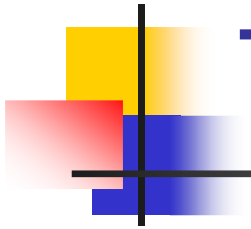
Esempio

```
package varie;
import segmenti.Punto;
public class TestPunto {
   public void assertTrue(String s, boolean asserzione) {
    if (asserzione) {
          System.out.print(".");
    } else {
          throw new AssertionFailedException(s);
    }
 }
 public void testPunto1() {
    Punto p = new Punto();
    p.setAscissa(10); p.setOrdinata(2);
    assertTrue("quadrante 1", p.getQuadrante() == 1);}
```



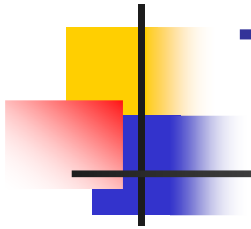
Esempio

```
public void testPunto2() {
    Punto p = new Punto();
    p.setAscissa(-1); p.setOrdinata(2);
    assertTrue("quadrante 2", p.getQuadrante() == 2);
}
public static void main(String[] args) {
    TestPunto test = new TestPunto();
    test.testPunto1();
    test.testPunto2();
    }
}
```



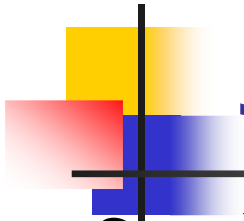
Test di Regressione

- **Uso delle asserzioni:**
  - l'idea generale è quella di aggiungere istruzioni con cui il codice si autoverifica
  - nel caso delle guardie, la verifica è una verifica "a priori"; l'asserzione è "utile" quando fallisce -> segnala un errore allo sviluppatore
  - nel caso dei test di regressione, la verifica è "a posteriori"; l'asserzione è utile quando è verificata -> segnala che il metodo è corretto
- ATTENZIONE ai vari usi delle asserzioni: le asserzioni utilizzate per le guardie servono a verificare "precondizioni" per l'esecuzione corretta dei metodi mentre le assezioni utilizzate nei test servono a verificare "postcondizioni" dopo l'esecuzione dei metodi



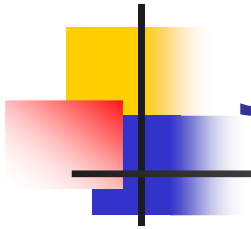
Test di Regressione

- I test di regressione non vengono eseguiti in modo interattivo ma "esercitando" i metodi del componente o dei componenti da verificare ed effettuando asserzioni sui risultati
- al termine viene prodotto un rapporto dei test "passati" e degli eventuali test "falliti"
- In questo modo è molto più facile individuare dove nascono i problemi
- In Java esistono framework molto utilizzati per lo sviluppo di test di regressione. In particolare JUnit (http://www.junit.org).
- Esercitazione: confrontare i meccanismi offerti dal C# per la gestione dei test



JUnit

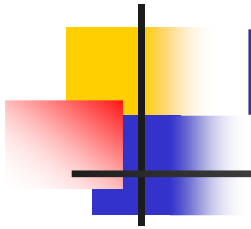
- Cosa è JUnit
  - uno strumento per lo sviluppo di test in Java principalmente test di unità (ma anche test funzionali) e di regressione
  - scritto da Kent Beck ed Erich Gamma, e poi riscritto per molti altri linguaggi
  - si tratta di un framework
  - tipicamente fornisce alcuni strumenti eseguibili (es: applicazione grafica per eseguire test)
  - stabilisce delle regole di scrittura del codice per potere usare questi strumenti
- Informazioni di base:
  - un test viene scritto sulla base di opportune classi del package junit.framework
  - è necessario utilizzare metodi particolari per le asserzioni
  - due "testrunner", junit.textui.TestRunner (testuale) e junit.swingui.TestRunner (grafico)



JUnit

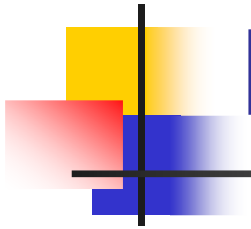
- la classe di Test estende junit.framework.TestCase
- il nome dei metodi di test deve cominciare con "test"
- le asserzioni avvengono utilizzando il metodo asserTrue() ereditato da junit.framework.TestCase

```
Package test;
import junit.framework.*;
import mediapesata.*;
public class TestStudente extends TestCase {
    public void testMediaPesata() {
    Studente studente = new Studente();
    Esame esame1 = new Esame();
    esame1.setVoto(24);
    esame1.setCrediti(3);
    studente.addEsame(esame1);
    Assert.assertTrue("media p. 2",studente.getMediaPesata() == 24);
    }
    public void testMediaPesata2() {
...
}
```



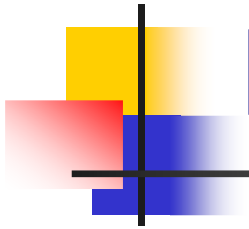
Le Regole di JUnit

- una classe di test deve estendere la classe junit.framework.TestCase
- ogni metodo di test deve essere una procedura pubblica (public void) il cui nome comincia con "test"
- le asserzioni vengono fatte usando il metodo statico Assert.assertTrue() con due argomenti: una stringa che rappresenta ilnome del test e l'asserzione da verificare
- Il metodo Assert.AssertTrue()
  - se l'asserzione è vera registra sull'interfaccia l'esecuzione corretta del test (es: stampa ".")
  - se l'asserzione è falsa, lancia junit.framework.AssertionFailedError una eccezione di tipo errore che provoca il fallimento del test
- Altri metodi per le asserzioni:
  - assertEquals(int valorePrevisto, int valoreReale)
  - assertEquals(String prevista, String reale) ecc.
  - Assert.assertNotNull(Object rif)
  - Assert.assertNull(Object rif)
  - Assert.assertFalse(boolean asserzione)



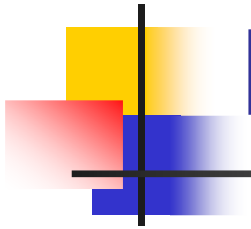
Le Regole di JUnit

- Possibili cause di fallimento di un test
  - junit.framework.AssertionFailedError: eccezione lanciata dal framework nel caso in cui una asserzione fallisca
  - qualsiasi altra eccezione (java.lang.Throwable): durante l'esecuzione dei test si è verificato un problema inaspettato che ha causato l'eccezione
- Quindi, al termine dell'esecuzione il test runner riporta quattro numeri
  - numero di test eseguiti
  - numero di test completati con successo
  - numero di test completati ma falliti per via del fallimento di una asserzione
  - numero di test non completati a causa del verificarsi di una eccezione



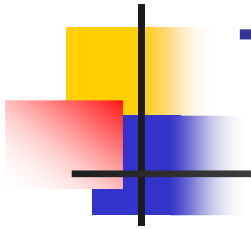
Fixture e SetUp

- Il concetto di "fixture" e di setUp:
  - Tipicamente, tutti i metodi lavorano con uno o più oggetti della classe da verificare. Trattandosi di test di unità, ogni metodo di test deve essere effettuato in modo isolato rispetto agli altri e cioè su oggetti completamente nuovi e non intaccati dai test precedenti
- Una possibilità
  - ciascun metodo di test crea ex-novo gli oggetti su cui eseguire i test
  - svantaggio: molto codice duplicato inutilmente per la creazione; es: Studente
- Una possibilità migliore
  - definire un gruppo di oggetti condivisi tra i test (una "fixture" – impianto, infrastruttura)
  - e ricrearli ogni volta in modo automatizzato



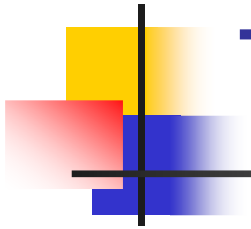
Fixture e SetUp

- In Junit, la fixture corrisponde alle proprietà della classe di test
- viene definito un metodo public void setUp() che inizializza le proprietà
- il metodo setUp() viene rieseguito dal framework prima di eseguire ciascun test, per avere sempre una fixture "fresca"
- Attenzione alla differenza tra setUp() ed un costruttore:
- il costruttore può inizializzare le proprietà una volta sola (alla creazione)
- setUp() viene richiamato dal framework prima di ogni test
- è previsto anche un metodo tearDown() nel caso in cui la fixture richieda operazioni di "pulitura" dopo il test



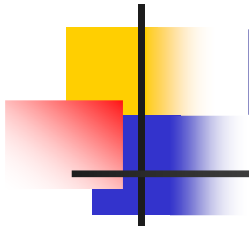
TestRunner e TestSuite

- Il concetto di testRunner
  - un testRunner è una classe java capace di eseguire i test di una classe di test
  - junit.textui.TestRunner è un esecutore testuale
  - uso: java junit.textui.TestRunner <classeDiTest>
- Esempio
  - java junit.textui.TestRunner segmenti.test.TestPunto
- junit.swingui.TestRunner
  - esecutore grafico con barra di progresso verde/rossa
  - possibilità di analizzare l'albero dei test
  - possibilità di ricaricare la classe di test dopo le modifiche
  - uso: java junit.swingui.TestRunner <classeDiTest>
- Esempio
  - java junit.swingui.TestRunner segmenti.test.TestPunto



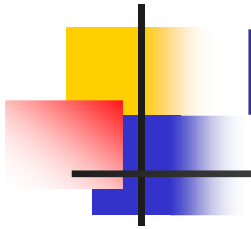
TestRunner e TestSuite

- Il concetto di testSuite()
  - i testRunner accettano come argomento il nome di una singola classe, ma spesso è necessario eseguire tutti i test dell'applicazione: molte classi
  - in questo caso è necessario creare una "suite di test", ovvero una classe che raccoglie i test di varie classi
  - eseguendo la suite si eseguono tutti i test



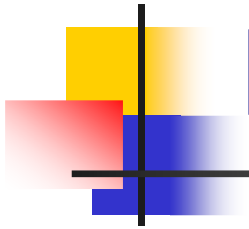
Linee Guida per il testing

- Linee guida n. 1
  - quali classi verificare ?
  - test di unità sono particolarmente adatti per i componenti di modello e persistenza
  - meno adatti ad interfaccia e controllo, per i quali è necessario scrivere test funzionali
- Linee guida n. 2
  - quali metodi verificare ?
  - verificare solo l'interfaccia dei componenti
  - verificare solo i metodi che possono contenere errori, e cioè quelli che hanno una qualche logica applicativa, ad esempio non è il caso di verificare i metodi set e i get
- Linea guida n. 3
  - quanti test per ogni metodo ?
  - per ogni metodo da verificare è necessario preparare un piano di test che copre le varie condizioni di utilizzo (condizioni particolari)
  - varie condizioni ordinarie (es: media con vari esami, media con un esame) e scorrette (es: media senza esami)



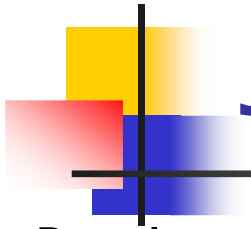
Linee Guida per il testing

- Linee guida n. 4
  - Come verificare le eccezioni ?
  - bisogna eseguire il metodo in condizioni scorrette, e verificare che sollevi le eccezioni attese
  - in questo caso il test non deve effettuare asserzioni, ma limitarsi a catturare l'eccezione con una regione catch
  - Se l'eccezione non si verifica, il metodo si comporta in modo scorretto e il test deve fallire; il metodo Assert.fail() provoca forzatamente il fallimento di un test
- Schema di test per una eccezione
  - viene chiamato il metodo in una regione try
  - se viene sollevata l'eccezione, viene catturata con il catch e il metodo termina con successo
  - se NON viene generata l'eccezione, bisogna forzare il fallimento del test con Assert.fail()



Linee Guida per il testing

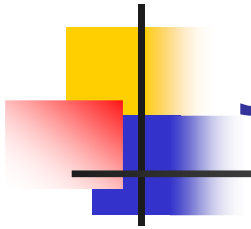
- Linea guida n. 5
  - che stile adottare per i metodi di test ?
  - i test sono codice aggiuntivo, soggetto a sua volta ad errori
  - è opportuno che i metodi di test siano semplici e brevi per aiutare ad identificare le cause di errore e che contengano poche asserzioni ciascuno, idealmente una per metodo
- Linea guida n. 6
  - cosa fare se un test fallisce ?
  - è stato individuato un baco catturato dai test
  - il test fornisce l'indicazione del componente e del metodo errati
  - una volta individuato l'errore è necessario correggerlo, ovvero effettuare il "debugging" del componente con i metodi ordinari
- Linea guida n. 7
  - cosa fare in caso si scopra un errore non catturato dai test ?
  - è possibile che, nonostante i test di unità abbiano successo, attraverso i test funzionali venga riscontrato un errore nel codice
  - in questo caso, prima di correggere l'errore, è opportuno introdurre un nuovo test che "catturi" l'errore (test che fallisce a causa dell'errore)



JUnit - Sintesi

Per riassumere, una **panoramica** delle JUnit:

- A class containing test methods **should subclass the TestCase class.**
- A TestCase can define any number of **public testXXX()** methods.
- When you want to check the expected and actual test results, you **invoke a variation of the assert() method**.
- TestCase subclasses that contain multiple testXXX() methods can use the **setUp() and tearDown() methods to initialize and release any common objects under test**, referred to as the **test fixture**. Each test runs in the context of its own fixture, calling setUp() before and tearDown() after each test method to avoid side effects among test runs.
- TestCase instances can be composed into TestSuite hierarchies that automatically invoke testXXX() methods defined in TestCase instances.
- A **TestSuite** is a composite of other tests, either TestCase instances or other TestSuite instances. The composite behavior exhibited by the TestSuite allows you to assemble test suites and run all the tests automatically and uniformly to yield a single pass or fail status.
- **JUnit is thus designed around two key design patterns**: the Command pattern (a TestCase is a command object) and the Composite pattern.



JUnit - Sintesi

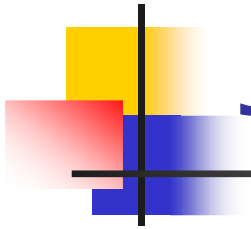
L'utilizzo delle Junit avviene seguendo lo **schema**:

1) Scaricare il package junit, unzipparlo, ed installarlo in una directory da aggiungere al classpath (set CLASSPATH=%CLASSPATH%;%JUNIT_HOME%\junit.jar); testare con *java org.junit.runner.JUnitCore org.junit.tests.AllTests*

2) Write a test case:

- Define a subclass of TestCase.
- **Override the setUp() method** to initialize object(s) under test.
- **Optionally override the tearDown()** method to release object(s).
- **Define one or more public testXXX() methods** that exercise the object(s) under test, usually by invoking some of his methods, and assert expected results. For example, to test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, write:

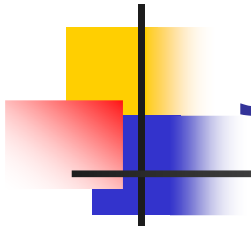
```
@Test public void simpleAdd() {
Money m12CHF= new Money(12, "CHF");
Money m14CHF= new Money(14, "CHF");
Money expected= new Money(26, "CHF");
Money result= m12CHF.add(m14CHF);
assertTrue(expected.equals(result)); }
```



JUnit - Sintesi

- Effettuare l'override di setUp() e tearDown() significa volere creare una fixture. I passi sono:
  - Add a field for each part of the fixture
  - Annotate a method with @org.junit.Before and initialize the variables there
  - Annotate a method with @org.junit.After to release resources allocated in setUp
  - For example, to write several test cases that want to work with different combinations of 12 Swiss Francs, 14 Swiss Francs, and 28 US Dollars, first create a fixture:

```
public class MoneyTest {
private Money f12CHF;
private Money f14CHF;
private Money f28USD;
@Before public void setUp() {
f12CHF= new Money(12, "CHF");
f14CHF= new Money(14, "CHF");
f28USD= new Money(28, "USD");      }
@After public void tearDown(){ … } }
```



JUnit - Sintesi

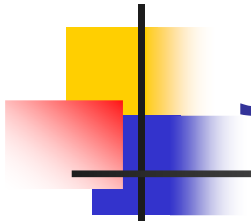
3) Successivamente, **si esegue il test**:

- from a Java program:
  - *org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);*
- from command line, (with both test class and junit on the classpath):
  - *java org.junit.runner.JUnitCore TestClass1.class [...other test classes...]*
- By design, the tree of Test instances is built in one pass, then the tests are executed in a second pass. The test runner holds strong references to all Test instances for the duration of the test execution. This means that for a very long test run with many Test instances, **none of the tests may be garbage collected until the end of the entire test run**. Therefore, if you allocate external or limited resources in a test, **you are responsible for freeing those resources**. Explicitly setting an object to null in the tearDown() method, for example, allows it to be garbage collected before the end of the entire test run.

JUnit - Sintesi

4) **Eventually, write a test suite:** write a Java class that defines a static suite() factory method that creates a TestSuite containing all the tests. Optionally define a main() method that runs the TestSuite in batch mode. An example test suite:

- *import junit.framework.Test;*
- *import junit.framework.TestSuite;*
- *public class ExampleTestSuite {*
  - *public static Test suite() {*
    - *TestSuite suite = new TestSuite();*
    - *suite.addTestSuite(MoneyTest.class);*
    - *// // Add more tests here //*
    - *return suite; }*
    - */** * Runs the test suite using the textual runner. */*
  - *public static void main(String[] args) {org.junit.runner.JUnitCore.run(suite()); }*
  }
- By creating a TestSuite in each Java package, at various levels of packaging, you can run a TestSuite **at any level of abstraction**. For example, you can define a com.mydotcom.AllTests that runs all the tests in the system and a com.mydotcom.ecommerce.EcommerceTestSuite that runs only those tests validating the e-commerce components.



JUnit - Sintesi

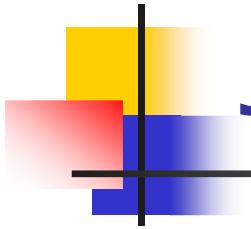
- Infine, qualche raccomandazione:
- Place tests in the same package/directory as the classes under test. For example:

```
Src
    Com
        Xyz
            SomeClass.java
            SomeClassTest.java
```

- To avoid combining application and testing code in your source directories, create a mirrored directory structure aligned with the package structure that contains the test code. For example:

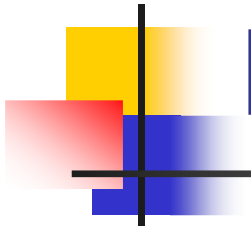
```
src
    com
        xyz
            SomeClass.java
Test
    com
        xyz
            SomeClassTest.java
```

These approaches allow the tests to access to all the public and package visible methods of the classes under test. To **test protected methods** instead**, p**lace your tests in the same package as the classes under test. To **test private methods** instead**,** you can use reflection to subvert the access control mechanism with the aid of the PrivilegedAccessor.



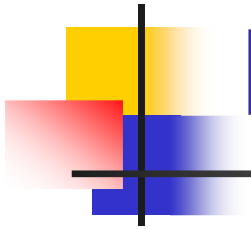
JUnit - Sintesi

- For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package.
- Make sure your build process includes the compilation of all tests. This helps to ensure that your tests are always up-to-date with the latest code and keeps the tests fresh.
- The software does well those things that the tests check.
- Test a little, code a little, test a little, code a little...
- Make sure all tests always run at 100%.
- Run all the tests in the system at least once per day (or night).
- Write tests for the areas of code with the highest probability of breakage.
- Write tests that have the highest possible return on your testing investment.
- If you find yourself debugging using System.out.println(), write a test to automatically check the result instead.
- When a bug is reported, write a test to expose the bug.
- The next time someone asks you for help debugging, help them write a test.
- Write unit tests before writing the code and only write new code when a test is failing.
- Leggere le FAQ su www.junit.org



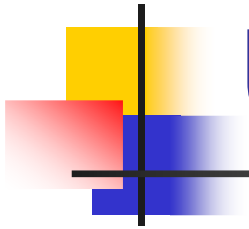
Debugging

- Debugging
  - ricerca della causa e correzione di un errore individuato attraverso un test (manuale o automatizzato)
- l problema fondamentale è la ricerca dell'errore
- Due tecniche fondamentali
  - stampe di debugging
  - utilizzo del debugger
- Stampe di debugging
  - istruzioni di stampa sullo standard output dei valori delle variabili
  - possono essere aggiunte al codice del componente da correggere o anche al codice dei test
  - per capire meglio le cause reali dell'errore
  - metodo rapido e molto diffuso



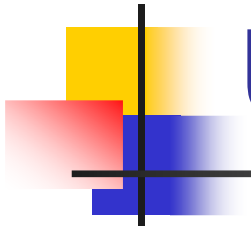
Debugging

- Utilizzo del debugger
  - il debugger consente di eseguire il codice dell'applicazione
  - impostando "breakpoint", linee di codice sorgente in corrispondenza delle quali l'esecuzione si interrompe
  - in modo da ispezionare la pila di attivazione e poter procedere passo passo
- Debugger ("Correttore")
  - strumento che consente di eseguire passo passo il codice
  - per ispezionare lo stack e lo heap durante il funzionamento di un programma
  - verificando i valori delle variabili
  - i valori dei parametri
  - lo stato dei componenti



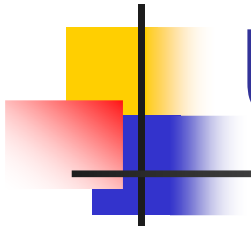
Uso del Debugger

- Le operazioni tipiche del debugger
  - definire i punti di interruzione
  - comandare l'esecuzione del codice
- Definire i punti di interruzione
  - "breakpoint", ovvero righe di codice
  - "guardie" (watch), ovvero espressioni che rappresentano condizioni sullo stato del programma in cui fermarsi
- Comandare l'esecuzione del codice
  - avviare l'esecuzione fino al punto di interruzione successivo
  - continuare sorpassando un punto di esecuzione
  - procedere passo passo
  - esplorare la pila
  - stampare i valori di espressioni complesse



Uso del Debugger

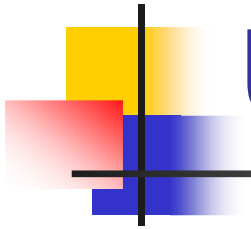
- Compilazione del codice
  - al solito, perchè le informazioni di debugging siano disponibili, è necessario compilare opportunamente il codice
- In Java
  - il compilatore javac prevede l'opzione –g per specificare le opzioni relative alle informazioni di debug



Uso del Debugger

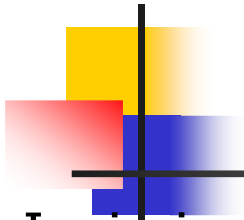
Varie opzioni:

- javac –g
  - produce nel file .class tutte le informazioni di debug
- javac –g:none
  - nessuna info di debug
- javac –g:{lines, vars, source}
  - produce informazioni solo su numeri di linea, variabili nella pila e file di codice sorgente
- il valore standard è –g:lines, source



Uso del Debugger

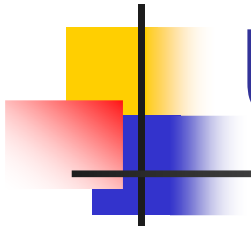
- Il debugger fornito con l'SDK
  - jdb.exe
  - debugger puramente testuale e non grafico eseguento jdb si accede ad un prompt attraverso il quale è possibile eseguire vari comandi
  - per un elenco: help



Uso del Debugger

I principali comandi di jdb:

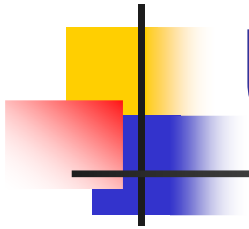
- **run** <classe>= Use **this** command to execute <classe> after starting jdb
- **print** <espressione>= display Java objects and primitive values. For instance: print TempClass.myStaticField, print myObj.myInstanceField, print myObj.myMethod()
- **Dump**= similar to print command. For objects, it is used to print the current value of each field defined in the object including Static and instance fields.
- **Locals**= visualizza il contenuto dello stackframe corrente
- **Cont/Step/next=** continues the execution of the debugged application after a breakpoint, exception, step executes a line at a time, next steps over a method.
- **Stepi=** executes the current instruction. In other words, the code at the => will execute but the current line will not advance to the next instruction.
- **step up=** executes until the current method returns to its caller. Simply put, this stepper executes a method and nothing else.
- **Threads**= lists the threads that are currently running.



Uso del Debugger

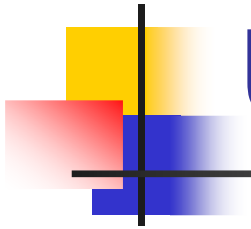
I principali comandi di jdb:

- **Where=** used to dump the stack of the current thread. **where all** is used to dump the stack of all threads in the current thread group, and **where thread index** command is used to dump the stack of the specified thread.
- **Stop**= usato per i breakpoint, i.e.: stop at TempClass:14 (breakpoint at the first instruction for line 14 of the source file), o stop in TempClass.<init> (TempClass constructor). Specify the argument types for overloaded methods to select the proper one, i.e. "TempClass.m1(int,java.lang.String)", or "TempClass.m1()".
- **Clear=** remove breakpoints using a syntax as in "clear TempClass:20"
- **Methods**= fornisce una lista dei metodi disponibili per una data classe
- **List**= display the code at the breakpoint.



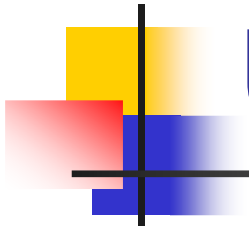
Uso del Debugger

- Here's **the secret to debugging** a program. Every time you execute a statement, make a prediction about what it will do. Will a variable's value change? Will you run a method? Will the 'if' statement happen or not? Then, when you execute it, see whether your prediction was right.
- If your prediction was right, then the program did exactly what you thought it should, and there's no point wasting time examining that statement. If everything in your program worked the way you thought it did you wouldn't have a bug! On the other hand, if your prediction is wrong, you've just found a problem that needs to be solved.



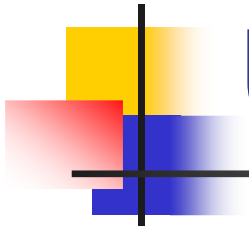
Uso del Debugger

- You may have noticed that there's **a lot of repetition** in a command-line debugger. Every time you want to check up on a variable you have to type print variable, for example.
- JDB offers a feature called a **monitor**, which is just a command that gets executed every time the debugger stops. That means every time you hit a breakpoint and every time you step over a line of code (or into a method) your command will get run, i.e. monitor locals
- You can just type monitor by itself to see what monitors you have right now, and use **unmonitor #** with the number listed to turn it off.



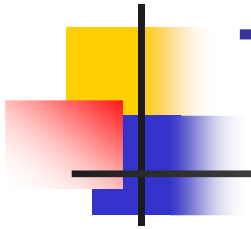
Uso del Debugger

- Another command is useful when for instance a variable would get messed up in the middle of a huge loop (this is entirely realistic). It would also be painful to have to do next 100 times every single time you wanted to test it in the debugger. So, start JDB and tell it to **watch** the state variable by typing the watch <class_name>.<variable> command. Then run the program; you do not need to set any breakpoints: that's the whole point of this.
- That's an important thing to understand. Just as a breakpoint causes the debugger to stop **before** executing the statement on that line, a watch causes the debugger to stop **before** the variable is modified. The language in the output makes this clear: Field (<class>.<var>) is <current_value>, will be <future_value>. Use the list command (and any others you may find helpful) to determine where you are in the code and whether this particular change to the variable is important or not.



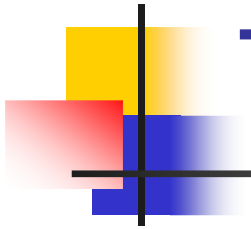
Uso del Debugger

- For all JDB's great features, **there are a few key areas it's not very strong**. For example, it would be nice to have it print out the entire call stack (i.e. the list of functions currently executing: main() called func1() called func2() called...) so you could see how you got to where you are (particularly if you call the same method from multiple locations. JDB doesn't offer such a feature, but other debuggers definitely do.
- **The power of JDB is largely its ubiquity**. Everywhere there's a Java compiler there ought to be JDB, so even if you don't have access to a full-featured development environment you can still use some powerful debugging tools.
  - A development environment like jGrasp (http://www.eng.auburn.edu/department/cse/research/grasp/) or the NetBeans integrated debugger will contain a debugger that does everything JDB does, and usually more.
- Esercitazione: Verificare che strumenti offre il C# per il debug



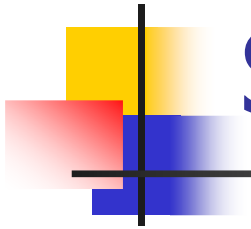
Stampe di Debugging Tradizionali

- L'approccio tradizionale alle stampe
  - inserire nel codice istruzioni di stampa; es: System.out.println()
  - per visualizzare sullo schermo durante l'esecuzione il valore delle variabili, dei parametri e l'evoluzione del flusso di controllo
  - le istruzioni vengono commentate dopo aver trovato l'errore
  - ed eventualmente ripristinate per altri errori



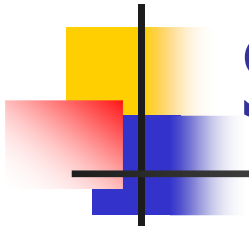
Stampe di Debugging Tradizionali

- Il vantaggio di questo approccio
- è un approccio semplice e immediato (rispetto all'uso del debugger)
- Gli svantaggi di questo approccio sono però numerosi:
  - è necessario modificare il codice per commentare – decommentare le stampe di debug (introduce potenzialmente errori)
  - le stampe vengono prodotte tutte sullo standard output e possono facilmente produrre lo scorrimento veloce dello schermo pregiudicandone la leggibilità, d'altro canto produrre le stampe in un flusso diverso (es: file) complicherebbe decisamente le istruzioni di stampa



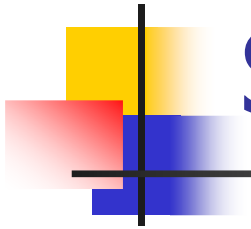
Sistema di Logging

- Sistema di logging
  - sistema che permette di includere nel codice stampe di debug, che descrivono il funzionamento dell'applicazione
  - e di produrle solo su richiesta dell'utente (durante le sessioni di correzione)
  - oltre che in formati e su supporti diversi
  - escludendole nelle versioni in produzione



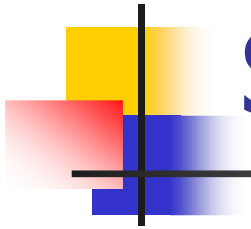
Sistema di Logging

- Sistema di logging
  - libreria di classi per la registrazione di informazioni sull'esecuzione del codice
- Caratteristiche fondamentali
  - consente registrazioni a diversi livelli di "gravità"
  - consente di registrare su dispositivi diversi
  - consente di registrare in formati diversi
- Il sistema di logging standard di Java
  - il package java.util.logging (da Jdk1.4)
- Un sistema di logging molto diffuso
  - Log4j (org.apache.log4j)
  - open source (Apache Software Foundation) scaricabile da http://logging.apache.org utilizzato da moltissimi progetti industriali



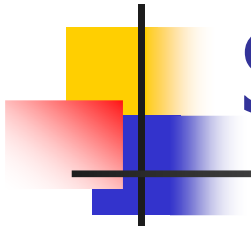
Sistema di Logging

- Concetti principali di un sistema di logging:
  - messaggio di logging
  - logger
  - handler
  - livello di logging
- Messaggio di logging ("log record")
  - stringa registrata durante l'esecuzione di un metodo
- Logger
  - componente responsabile della creazione dei messaggi di logging
  - è possibile utilizzare più di un Logger all'interno della stessa applicazione; es: un logger per package o per classe
  - ogni logger ha un nome
  - i logger sono organizzati in una gerarchia, con un logger principale, detto "root" logger



Sistema di Logging

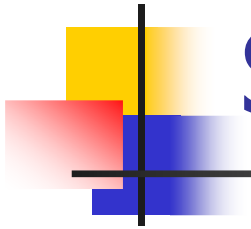
- Handler (o Appender)
  - componente responsabile di registrare un messaggio di logging prodotto da un logger su un flusso o dispositivo
  - con un formato specificato
  - esempio: console, file, dbms ecc.
  - a ciascuna categoria di logger può essere associato uno o più handler
- Livello di logging
  - descrive la gravità e l'urgenza di un messaggio
  - Alcuni livelli tipici, in ordine decrescente
    - SEVERE: problema molto grave
    - WARNING: avvertimento potenz. grave
    - INFO: messaggio informativo sul funzionam.
    - FINE: messaggio di "tracing"
    - FINEST: messaggio di "tracing" dettagliato



Sistema di Logging

Una caratteristica fondamentale:

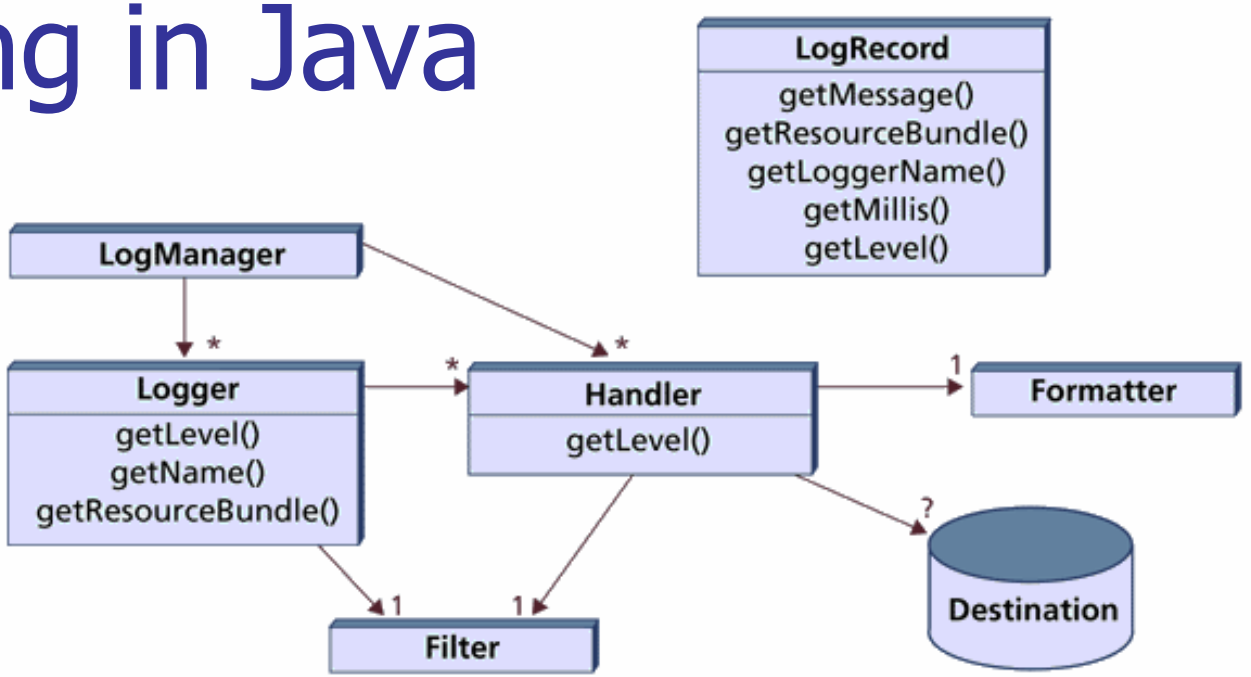
- è possibile associare un livello a ciascun logger, in modo da abilitare o disabilitare i messaggi prodotti dal logger
- il livello standard è tipicamente INFO, ma è configurabile attraverso un file di configurazione, che consente di controllare quali messaggi vengono effettivamente prodotti, eventualmente disabilitandoli completamente
- le istruzioni di creazione dei messaggi vengono introdotte e restano nel codice (non c'è bisogno di commentarle); a seconda del livello abilitato, le registrazioni vengono effettuate o meno senza dover modificare il codice



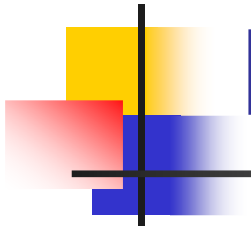
Sistema di Logging

- Caratteristiche essenziali desiderabili per un sistema di log:
  - Efficienza: le registrazioni devono essere prodotte rapidamente per non rallentare il codice e il test per decidere se escludere o meno una registrazione deve essere eseguito rapidamente
  - Robustezza: il codice del sistema di logging viene eseguito assieme al codice dell'applicazione quindi il sistema di logging non deve lanciare eccezioni altrimenti interromperebbe il funzionamento dell'applicazione. Sistemi di questo tipo vengono detti sistemi di tipo "fail-stop"

Logging in Java

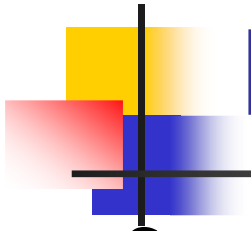


- Il sistema di logging standard di Java è il package java.util.logging
- la classe **LogManager** è una classe in Singleton instance (unica istanza) used to maintain a set of shared state about Loggers and log services. Mediante i metodi getLogManager si ottiene il manager e con addLogger si aggiungono loggers alla cache per poi reperirli mediante il metodo getLogger. The LogManager object manages a hierarchical namespace of Logger objects (all named Loggers are stored in this namespace) and it also manages a set of logging control properties (key-value pairs). The global LogManager object can be retrieved using LogManager.getLogManager(). The LogManager object is created during class initialization and cannot subsequently be changed.



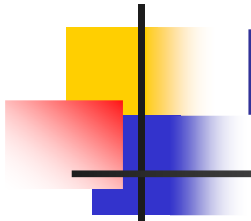
Logging in Java

- I **parametri di configurazione** di default usati dal LogManager vengono letti dal file di properties che si trova nella directory lib del JRE (logging.properties). Questi settings possono essere cambiati a runtime utilizzando dei metodi appropriati o modificando il file di properties; la classe LogManager mette a disposizione dei metodi che consentono di rileggere il file di configurazione. La configurazione di default prevede due Handler globali, uno su file nella home directory utente e un altro su schermo (Console). Il livello di default di questi logger è INFO.
  Per modificare queste impostazioni a runtime si può intervenire in vari modi:
  - Creando nuovi Handler come ad esempio FileHandlers, MemoryHandlers, e PrintHandlers
  - Creando nuovi Logger da associare a specifici Handler
  - Utilizzando il metodo setLevel di Logger o LogManager per cambiare il livello dei Loggers



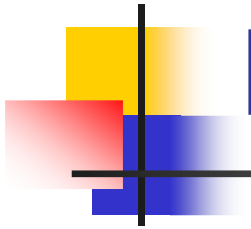
Logging in Java

- Considerando gli altri elementi dello schema della figura…
- Applications make logging calls on **Logger objects**. These Logger objects allocate **LogRecord objects** which are passed to **Handler objects** for publication. Both Loggers and Handlers may use logging **Levels and Filters objects** to decide if they are interested in a particular LogRecord. When it is necessary to publish a LogRecord externally, a Handler can use a **Formatter object** to localize and format the message before publishing it to an I/O stream.
- Each Logger keeps track of a set of output Handlers. Some Handlers may direct output to other Handlers **(piped handlers)**. In such cases, any formatting is done by the last Handler in the chain.
- Each log message has an associated **log Level**. The Level gives a rough guide to the importance and urgency of a log message. The Level class defines seven standard log levels, ranging from FINEST to SEVERE.



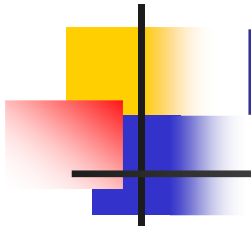
Logging in Java

- Generally, **loggers are named using a dot-separated hierarchical namespace**. Logger names can be arbitrary strings, but they should normally be based on the package name or class name of the logged component, such as java.net or javax.swing. In addition it is possible to create "anonymous" Loggers not stored in the Logger namespace.
- **Logger objects may be obtained by calls on one of the getLogger** methods. These will either create a new Logger or return a suitable existing Logger. Hence **the same logger can be shared among classes belonging to the same package.** The getLogger method finds or creates a logger for a named subsystem. If a logger has already been created with the given name it is returned. Otherwise a new logger is created. If a new logger is created its log level will be configured based on the LogManager configuration and it will configured to also send logging output to its parent's handlers. It will be registered in the LogManager global namespace, thus **any new logger is linked to the LogManager**
- **Logging messages will be forwarded to registered Handler objects, which can forward** the messages to a variety of destinations, including consoles, files, OS logs, etc.



Logging in Java

- **Each Logger has a "Level" associated** with it. This reflects a minimum Level that this logger cares about. If a Logger's level is set to null, then its effective level is inherited from its parent, which may in turn obtain it recursively from its parent, and so on up the tree. The log level can be configured based on the properties from the logging configuration file. However it may also be dynamically changed by calls on the **Logger.setLevel method**.
- The Logger class provides **a large set of convenience methods for generating log messages**. For convenience, there are methods for each logging level, named after the logging level name. Thus rather than calling "logger.log(Constants.WARNING,..." a developer can simply call the convenience method "logger.warning(..."

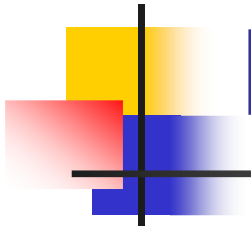


Logging in Java

- There are **two different styles of logging** methods. First, there are methods that take an explicit source class name and source method name. These methods are intended for developers who want to be able to locate the source of messages, i.e
  - *void warning(String srcClass, String srcMethod, String msg);*

  Second, there are a set of methods that do not take explicit source class or source method names. These are intended for developers who want easy-to-use logging and do not require detailed source information.
  - *void warning(String msg);*

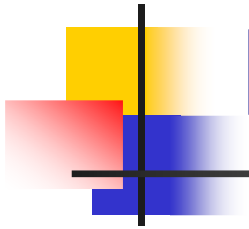
  For this second set of methods, the Logging framework will make a "best effort" to determine which class and method called into the logging framework and will add this information into the LogRecord. However, it is important to realize that this automatically inferred information may only be approximate. The latest generation of virtual machines perform extensive optimizations when JITing and may entirely remove stack frames, making it impossible to reliably locate the calling class and method.



Logging in Java

- This program performs logging using the default configuration. It relies on the root handlers that were established by the LogManager based on the configuration file. It creates its own Logger object and then calls that Logger to report various events.

```
package com.wombat;
public class Nose{
    // Obtain a suitable logger.
    private static Logger logger = Logger.getLogger("com.wombat.nose");
    public static void main(String argv[]) {
        // Log a FINE tracing message
        logger.fine("doing stuff");
        try{
            Wombat.sneeze();
        } catch (Exception ex) {
            // Log the exception
            logger.log(Level.WARNING, "trouble sneezing", ex);
        }
        logger.fine("done");    }
}
```



Logging in Java

- Here's a small program that sets up it own logging Handler and ignores the global configuration.

```
package com.wombat;
import java.util.logging.*;
public class Nose {
    private static Logger logger = Logger.getLogger("com.wombat.nose");
    private static FileHandler fh = new FileHandler("mylog.txt");
    public static void main(String argv[]) {
        // Send logger output to our FileHandler.
        logger.addHandler(fh);
        // Request that every detail gets logged.
        logger.setLevel(Level.ALL);
        // Log a simple INFO message.
        logger.info("doing stuff");
        try {           Wombat.sneeze();        }
     catch (Exception ex) {   logger.log(Level.WARNING, "trouble sneezing", ex);    }
        logger.fine("done");
    }}
```