UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Un processo di sviluppo *software* per il miglioramento misurabile della qualità di prodotto

Relatore:
Prof. TULLIO VARDANEGA

Autore:
NICOLA BERTAZZO

ANNO ACCADEMICO 2007-2008

Indirizzo di posta elettronica dell'autore: nicola.bertazzo@gmail.com

Indice dei contenuti

Elenco delle figure

Prefazione

Un programmatore è consapevole che, per sviluppare in modo corretto qualsiasi prodotto *software*, è indispensabile identificare i passi da compiere e l'ordine in base al quale devono essere compiuti. Seguendo un ordine è possibile creare un processo ordinato che può essere correttamente applicato ogni qualvolta sia necessario creare un novo prodotto *software*.

Il processo di sviluppo e gestione di un prodotto *software* viene descritto in tutti i libri di ingegneria del *software*. Le informazioni che troviamo in questi libri sono indispensabili ma solitamente di tipo teorico e risulta difficile metterle in pratica nell'attività quotidiana di un programmatore.

Questa guida non vuole coprire tutte le parti dell'ingegneria del *software* ma vuole focalizzarsi nelle metodologie e tecniche di sviluppo e verifica del *software*.

Lo scopo di questa guida è descrivere in modo pragmatico degli strumenti, delle metodologie e delle tecniche per adottare il processo di sviluppo e verifica di un prodotto *software* in modo automatico e ripetibile. I principi e le *best practice* riportati in questa guida sono il riassunto dell'esperienza di molti sviluppatori, che hanno riportato le loro conoscenze in diversi libri, con lo scopo di aumentare la qualità delle comuni attività di uno sviluppatore e dei prodotti *software*.

Per permettere al lettore di approfondire gli argomenti trattati in questa guida verranno sempre forniti i riferimenti che permetteranno di risalire alla fonte dell'informazione riportata.

Struttura della guida

La guida è suddivisa in sette capitoli che descrivono l'utilizzo di metodologie e tecniche per misurare e aumentare la qualità di un progetto *software*:

- Capitolo 1: vengono descritti alcuni principi base che hanno lo scopo di aiutare un programmatore a lavorare in modo migliore aumentando la qualità del processo di sviluppo e del *software* prodotto;
- Capitolo 2: viene descritto il processo di sviluppo individuato per misurare e migliorare la qualità di un prodotto *software*. Per ogni parte che compone il processo vengono descritte le pratiche adottate e introdotti gli strumenti che permettono di realizzare il processo per la creazione di un progetto sviluppato in linguaggio *Java*;
- Capitolo 3: vengono descritti i principali aspetti per la realizzazione dei test di unità in un progetto sviluppato in linguaggio *Java*;
- Capitolo 4: vengono descritti alcuni esempi per realizzare parti di un progetto sviluppato in linguaggio *Java*. Gli esempi forniti possono essere utilizzati come guida per iniziare ad adottare la pratica di *Test-Driven Development* e realizzare i test di unità utilizzando la libreria *JUnit*;
- Capitolo 5: vengono descritti e forniti alcuni esempi per realizzare il processo di costruzione e verifica di un prodotto *software* in modo automatico, utilizzando gli strumenti *Ant* e *Maven*. Inoltre vengono forniti alcuni esempi che permettono di eseguire degli strumenti per effettuare l'analisi del codice sorgente di un progetto e produrre delle misure di qualità;
- Capitolo 6: vengono descritti e forniti alcuni esempi per realizzare il processo di integrazione continua tramite l'utilizzo di *Cruise Control* e *Continuum*;
- Capitolo 7: vengono descritti alcuni aspetti dei test di integrazione e dei test funzionali. Viene fornito un esempio dove viene descritto come realizzare alcuni test funzionali attraverso l'utilizzo di *Fitnesse*;

Destinatari

Questa guida è rivolta principalmente agli studenti del corso di Laurea Triennale e Specialistica in Informatica e agli sviluppatori *software* e *quality assurance manager*. Può essere utilizzato come *tutorial* iniziale per l'utilizzo di alcuni strumenti:

- *Subversion* per gestire un progetto attraverso un sistema di versionamento;
- *Junit* per effettuare test di unità;
- *Maven* e *Ant* per la gestione di progetti *software*;
- *CruiseControl* e *Continuum* per la realizzazione di un sistema di integrazione continua;
- *Fitnesse* per effettuare test di integrazione e test delle funzionalità;
- *JUnitPerf* per effettuare test delle prestazioni;

In ogni capitolo verranno forniti degli esempi che aiuteranno il lettore a prendere confidenza con gli argomenti trattati. I principi descritti sono indipendenti dal linguaggio di programmazione ma gli esempi e gli strumenti riportati sono rivolti per lo sviluppo di prodotti *software* in linguaggio *Java*.

Convenzioni tipografiche

Nella guida sono state adottate le seguenti convenzioni tipografiche:

- *Termine inglese*: indica delle parole che sono in lingua inglese
- *Programma*: indica il nome di un programma
- `Elemento`: indica un elemento o un'attributo *XML*
- `Comando`: indica un comando o un *path*
- `PaginaWeb`: indica un riferimento a una pagina *Web*

Capitolo 1

Principi ispiratori

1.1 Responsabilità

Uno tra i principi fondamentali per raggiungere il successo è farsi carico delle proprie responsabilità[1].

Il lavoro del programmatore consiste nella contribuzione per lo sviluppo di progetti *software*. L'obiettivo principale di quest'attività è la realizzazione di un prodotto affidabile in modo economico nei tempi e nei termini concordati con il cliente.

La realizzazione di un progetto *software* è un attività di elevata complessità, difficile nella pianificazione e nella conduzione, soggetta a scadenze temporali ed ad alta intensità di lavoro umano.

Comportamento responsabile

Nell'attività quotidiana di un programmatore è facile commettere degli errori. Per non andare contro all'obiettivo principale della realizzazione di un prodotto *software*, quando capita di accorgersi di aver fatto un errore, la cosa più giusta da fare, è agire con professionalità e onestà e ammettere di aver commesso l'errore. Quando si ammette di aver commesso un errore la cosa migliore da fare è quella di offrire delle opzioni e non delle scuse. Tipicamente le scuse vengono interpretate come mancanza di responsabilità, per questo è consigliato offrire delle opzioni, che permettono la correzione parziale o totale dell'errore.

Responsabilità nel *team* di sviluppo

Nella realizzazione di un progetto un programmatore collabora con altre persone. Nell'attività di sviluppo il programmatore produce codice e documentazione. Oltre ad essere responsabile del comportamento delle funzionalità che produce, il programmatore deve permettere agli altri componenti del *team* di sviluppo di accedere alle componenti da lui prodotte.

È quindi necessario che tutto il codice, che viene prodotto dal *team* di sviluppo, sia salvato in un posto sicuro, che periodicamente vengano fatti dei *backup*, per minimizzare i rischi di perdite di dati, e che il codice sia accessibile da tutti i componenti del *team*. Nella sezione 2.2.1 a pagina 16 viene spiegato come utilizzare uno strumento che permette di revisionare e condividere prodotti *software*.

Convenzioni

Per migliorare la collaborazione è consigliato definire delle convenzioni che devono essere rispettate da tutti i componenti del *team* di sviluppo. In questo modo ogni componente dovrà rispettare delle regole che permetteranno di produrre un prodotto in modo standard. Alcune convenzioni che possono essere stabilite e devono essere rispettate sono:

Convenzioni di stili di codice: Definiscono come deve essere scritto il codice. Se tutti gli sviluppatori seguono una convenzione, il codice risulta più leggibile e semplice da interpretare e mantenere. Nel linguaggio di programmazione *Java*, una tra le convenzione di stile di codice più utilizzata è reperibile a questo

[1] l'essere responsabile; il poter essere chiamato a rispondere degli effetti dannosi delle proprie o altrui azioni

indirizzo: `http://java.sun.com/docs/codeconv/`. Nelle sezioni 5.3.7 e 5.4.8 viene descritto uno strumento che permette di verificare il rispetto di convenzioni di stile nel codice di un progetto.

Convenzioni sulla struttura del progetto: Se viene seguita una convenzione che definisce la struttura del progetto, tutti i progetti sviluppati avranno la stessa struttura. In questo modo gli sviluppatori che passano da un progetto all'altro minimizzeranno il tempo di *start-up*.

Convenzioni sulle modalità di verifica: È possibile definire delle convenzioni che permettono di stabilire le modalità con cui viene verificato il progetto. In questo modo tutti i componenti del *team* di sviluppo effettueranno verifiche del comportamento del progetto in modo standard. Così facendo il comportamento di tutti i moduli del progetto, sviluppati da diversi sviluppatori, saranno verificati allo stesso modo.

Responsabilità del comportamento dei prodotti

Visto che un programmatore realizza delle funzionalità, esso è responsabile del loro comportamento. Per assicurare che il comportamento delle funzionalità prodotte sia quello desiderato è consigliato effettuare attività di test.

L'attività di test consiste nell'assicurare che una funzionalità prodotta funzioni come desiderato e nel verificare l'assenza di alcuni errori. Tramite i test quindi un programmatore documenta il comportamento di una funzionalità create, in modo che anche altri sviluppatori riescano a capire come realmente funzioni. Nei successivi capitoli viene descritto come creare i:

- Test di unità
- Test di integrazione
- Test funzionali
- Test delle prestazioni

È quindi necessario che in un *team* di sviluppo vengano stabilite delle regole e delle convenzioni che tutti i componenti devono rispettare in modo responsabile.

1.2 Ricerca dell'ordine

Il termine entropia deriva dalla fisica e si riferisce alla quantità di disordine e caos presente in un sistema. Il termine entropia si riferisce alla grandezza termodinamica il cui aumento misura la diminuzione dell'energia posseduta dà un sistema isolato.

In tutte le città alcune costruzioni abbandonate sono curate e di bell'aspetto altre invece sono rovinate e brutte. Ricerche sulla criminologia e sull'urbanistica [2] hanno scoperto una cosa molto curiosa, la principale causa che tende a portare al degrado un edificio disabitato e di bell'aspetto è la presenza di una finestra rotta.

Una finestra lasciata rotta per diverso tempo, dà ad un edificio inabitato un senso di abbandono. Per questo con il passare del tempo altre finestre iniziano a rompersi e tipicamente vandali e criminali iniziano a rovinare sempre di più l'edificio. L'edificio quindi diventa rovinato e di brutto aspetto e dà un senso di degrado alla zona dove si trova.

"The Broken Window Theory" ha ispirato il dipartimento di polizia di New York a prevenire il degrado degli edifici abbandonati cercando di sistemare subito i piccoli degradi. Questa azione ha avuto effetti positivi diminuendo seriamente il livello della criminalità.

Il disordine è una delle principali cause del fallimento di un progetto *software*. Un programmatore deve trarre ispirazione da *"The Broken Window Theory"* e non lasciare disordine nell'attività giornaliera che sta svolgendo.

Si deve quindi cercare di correggere ogni imperfezione trovata il prima possibile. Se per caso non si trova tempo per correggere l'imperfezione, si dovrebbe evidenziala (con un commento o un messaggio) in modo che gli altri possano capire che l'imperfezione è stata trovata e verrà corretta, così continueranno a percepire che il progetto è in ordine.

Quando si lavora in un progetto dove l'entropia sta crescendo è meglio fermarsi e cercare di ridurre il disordine. Se si ha l'impressione che sia impossibile togliere tutto il disordine dal progetto sarebbe meglio rifarlo da capo in modo ordinato.

In contrapposizione se si sta lavorando in un progetto ordinato, ben progettato, dove tutto il codice è commentato e scritto in modo professionale, tutto il *team* del progetto lavorerà in modo professionale e ordinato (tutti avranno paura di essere la causa dell'aumento dell'entropia del progetto). Se per caso c'è una scadenza (il rilascio di una versione o una dimostrazione) ci potrà essere un aumento dell'entropia, ma se il *team* è abituato all'ordine, dopo la scadenza il progetto dovrà essere riportato ai livelli d'ordine prefissati.

Nella sezione 2.4.2 verranno presentati alcuni strumenti utili per misurare e controllare l'entropia di un progetto.

1.3 Evitare le duplicazioni

Un programmatore crea, colleziona, organizza e mantiene conoscenza in forma di dati. La conoscenza assume varie forme: viene documentata nelle specifiche, viene resa concreta quando si esegue del codice e durante la fase di test viene usata per controllare che le aspettative siano verificate.

Sfortunatamente la conoscenza non è stabile, cambia, a volte anche troppo rapidamente. Ad esempio un requisito può cambiare durante un incontro con il cliente. I test possono evidenziare che un algoritmo scelto non è adatto o non funziona secondo le aspettative.

L'instabilità della conoscenza costringe un programmatore a passare gran parte del suo tempo a mantenere, riorganizzare e cambiare le conoscenze presenti nel sistema. Quando si mantiene la conoscenza, ci si trova a cambiare frequentemente la rappresentazione delle cose. Il problema di questa attività è la facilità di duplicare la conoscenza (p.es. nella specifica, nei processi e mentre si programma). Se lo stesso concetto è espresso in più parti, quando questo viene modificato, è necessario apportare lo stesso cambiamento in tutte le parti dove è espresso. Lavorando in questo modo si aumenta notevolmente la possibilità di avere parti inconsistenti.

Quando si duplica la conoscenza si aumenta la difficoltà a mantenerla e si va ad infrangere il principio DRY[2]:

> *"Ogni fonte di conoscenza deve avere una singola, non ambigua, autorevole rappresentazione all'interno di un sistema."*

Di seguito vengono riportate le più comuni tipologie di duplicazione.

Duplicazioni obbligatorie

In questa sezione vengono descritte le tipologie di duplicazioni che non derivano da errori ma sono imposte dal progetto o dal linguaggio di programmazione.

Rappresentazione multipla dell'informazione

Spesso è necessario avere la stessa informazione rappresentata in diversi formati. Ad esempio:

- Quando si scrive un'applicazione *client server*, utilizzando differenti linguaggi, si può avere la necessità di avere componenti comuni in entrambe le parti
- In un'applicazione, quando si accede ai dati di un *database*, si hanno bisogno di classi e attributi che replicano la struttura del *database*

Questi tipi di duplicazioni possono essere rimosse tramite l'utilizzo di generatori di codice. Parti uguali espresse in differenti linguaggi possono essere create da generatori di codice che partono da metadati comuni. Ogni volta che deve essere apportato un cambiamento, questo viene fatto nei metadati comuni, e viene rigenerato il codice in differenti linguaggi. I generatori possono essere creati ad hoc, oppure si possono utilizzare applicativi *software* specifici[3].

[2] *Don't repeat yourself*

[3] `http://www.codegeneration.net/`

Duplicazione nella documentazione del codice

I programmatori devono commentare il codice che producono. Il codice non commentato e non documentato è difficile da mantenere e da gestire.

Esistono due tipologie di commenti:

1. I commenti che spiegano il comportamento e le funzionalità di una funzione o di un metodo o di una classe

2. I commenti che spiegano come è stato realizzato il codice

La prima tipologia è indispensabile perché spiega ad alto livello il comportamento del componente che si sta sviluppando. Questo tipo di conoscenza tipicamente cambia con una frequenza minore rispetto all'altra tipologia di commenti ed è legata ai requisiti del sistema.

I commenti che spiegano come è stato realizzato il codice, quindi interni al metodo o alle funzione, cambiano con la stessa frequenza con cui cambia il codice (con un approccio *Test-Driven Development* questi commenti cambiano molto spesso).

Il principio DRY spinge a commentare il meno possibile il codice, quindi non eccedere con i commenti del secondo tipo. In questo modo non si duplica conoscenza, e non si ha conoscenza inconsistente. Ogni volta che si apportano dei cambiamenti al codice si dovrebbero modificare anche i commenti, in questo modo si aumenta il rischio di avere delle inconsistenze nelle informazioni. Avere commenti inesatti o inconsistenti con il codice è peggio di non avere commenti. Per questo motivo è consigliato scrivere codice professionale, auto esplicativo, in modo da dover inserire il minor numero di commenti nel codice.

Duplicazione nel codice sorgente

Molti linguaggi di programmazione impongono duplicazioni nel codice sorgente. Questo accade nei linguaggi che separano l'interfaccia dall'implementazione (p.es. nel linguaggio `C` esistono i file di specifica e i file di implementazione). Alcuni ambienti di programmazione permettono di generare automaticamente i file di specifica dai file di implementazione.

Questa duplicazione non genera troppi disturbi visto che, durante la compilazione, se i file sono inconsistenti vengono segnalati degli errori.

Duplicazione per impazienza

A volte la pressione causata da scadenze può indurre a commettere delle duplicazioni per impazienza. Supponiamo che uno sviluppatore abbia bisogno di utilizzare una funzionalità simile ad una già creata in passato. La cosa più rapida da fare è copiare e incollare la funzionalità e modificarla. In questo modo il programmatore ha guadagnato qualche secondo ma è andato contro il principio *DRY*. A causa della duplicazione per impazienza, se la funzionalità copiata conteneva degli errori questi sono stati replicati in altre parti. Per mantenere le due funzionalità verrà impiegato il doppio del tempo, visto che ogni volta che viene trovato un errore, deve essere corretto in due punti differenti.

È quindi buona norma non essere impazienti, spendere più tempo inizialmente e cercare di non duplicare il codice, in questo modo verrà speso molto meno tempo in seguito per la manutenzione.

Duplicazione nel *team* di sviluppo

La più difficile tipologia di duplicazione da prevenire in un progetto è quella causata dal lavoro concorrente di diversi sviluppatori. In questo caso molte funzionalità possono essere duplicate inavvertitamente da diversi sviluppatori. Queste duplicazioni possono non essere mai trovate e quindi la manutenzione può essere duplicata per sempre. Per diminuire le duplicazioni all'interno di un progetto è consigliato renderlo il più possibile ortogonale.

Dopo aver progettato il progetto devono essere suddivise le funzionalità tra i componenti del *team* in modo che non ci siano troppe dipendenze tra le varie parti. Devono essere definite delle convenzioni e delle regole in modo che tutti possano lavorare in modo standard e indipendente senza duplicare mansioni o attività.

Risulta più difficile diminuire le duplicazioni tra diversi moduli di un progetto. Infatti più sviluppatori potrebbero sviluppare in moduli indipendenti le stesse funzionalità. Per evitare ciò è necessario che ci sia una grande comunicazioni tra i componenti del *team* di sviluppo in modo da poter individuare le parti comuni e svilupparle una sola volta.

Le conoscenze sviluppate, codice e documentazione, devono essere accessibili a tutti i componenti del *team* in modo da permettere di verificare se uno stesso problema è già stato affrontato precedentemente e poter riutilizzare parti di codice già create (senza duplicarle).

1.3.1 Automatizzare per ridurre le duplicazioni

Spesso le attività di uno sviluppatore sono ripetitive e vengono effettuate in modo manuale. Se le attività possono essere descritte da una procedura allora è possibile realizzare delle automazioni che svolgano l'attività. Automatizzare un'attività ha i seguenti benefici:

Riduzione del tempo di esecuzione: Se un'attività viene eseguita tramite un'automazione impiega meno tempo rispetto alla sua esecuzione modo manuale. Il tempo impiegato per creare l'automazione verrà ammortizzato nel tempo

Risultati accurati, consistenti e ripetibili: Se una procedura viene svolta da una persona c'è il rischio che non venga eseguita sempre allo stesso modo. I tempi di esecuzione della procedura dipendono dalla velocità della persona che spesso commette degli errori. Se la procedura viene resa automatica i risultati saranno sempre consistenti, verrà impiegato sempre lo stesso tempo per effettuare la procedura e sarà più semplice ripeterla nel tempo

Riduzione della documentazione: Se una persona deve eseguire una procedura deve solo capire come eseguire lo *script* che effettua la procedura, senza dover imparare tutte le attività che la compongono. Se la persona è interessata a capire alcuni aspetti della procedura, può consultare lo *script* che la realizza, dove vengono descritte tutte le attività in dettaglio

Miglioramento dell'attività lavorativa: Oltre a facilitare l'attività lavorativa, permettono di effettuare procedure critiche con maggior frequenza e in modo sicuro

Per questi motivi, ogni volta che un'attività deve essere ripetuta più volte è consigliato creare uno *script* che la renda automatica.

Tipi di automazioni

Le automazioni possono essere classificate in tre categorie:

1. **Automazioni a comando**: permettono di eseguire un insieme di attività in modo consistente e ripetibile
2. **Automazioni programmabili**: Automazioni a comando che vengono eseguite ad intervalli di tempo prefissati
3. **Automazioni ad evento**: Automazioni a comando che vengono eseguite quando avvengono determinati eventi

Nell'attività di sviluppo di un progetto *software* le attività che possono essere automatizzate sono molte. Le più comuni sono:

- Conversione dei sorgenti in codice eseguibile
- Esecuzione della fase di verifica
- Realizzazione della documentazione
- Analisi statiche
- Realizzazione del prodotto
- Messa in opera

Nel capitolo 5 e 6 vengono realizzate alcune attività di un progetto *software* utilizzando vari tipi di automazione.

1.4 Ortogonalità

Ortogonalità è un concetto critico se si vuole produrre un sistema che sia semplice da progettare, costruire, verificare ed estendere. Tipicamente il concetto di ortogonalità viene raramente insegnato direttamente. Quando si impara ad applicare il principio di ortogonalità, si noterà un immediato aumento della qualità del sistema che si sta producendo.

Descrizione

Ortogonalità è un termine che deriva dalla geometria. Due linee sono ortogonali se formano un anglo retto, come gli assi di un piano cartesiano. In termini vettoriali le due linee sono indipendenti. Muovendo una linea, la proiezione sull'altra non cambia.

Nella produzione di *software*, il termine ha il significato di indipendenza e disaccoppiamento. Due o più componenti sono ortogonali se il cambiamento di una non influenza nessun altra componente. In un sistema ben progettato il codice che accede e modifica i dati in un *database* è ortogonale alle altre parti del progetto: Si può cambiare l'interfaccia utente senza cambiare la parte che accede e modifica il *database*, e si può cambiare la base di dati senza cambiare l'interfaccia utente. Un *pattern* di progettazione che permette di rendere indipendente un progetto, che deve accedere e modificare dei dati, dal supporto dove vengono salvati i dati è il *pattern DAO*.[4]

Benefici

Quando i componenti di un sistema sono fortemente indipendenti, l'unica attività da svolgere per mantenerlo è la correzione locale.

È consigliato progettare componenti indipendenti, con uno scopo ben definito. Quando i componenti sono isolati tra loro, se avviene un cambiamento di un componente non c'è il pericolo di rovinare il resto del sistema. I due principali benefici che derivano dal scrivere le componenti di un sistema in modo ortogonale sono:

Incremento della produttività :

- I cambiamenti sono localizzati, quindi il tempo di sviluppo e verifica viene ridotto. Risulta più semplice scrivere piccoli componenti, indipendenti e con un comportamento ben definito rispetto a scrivere grandi componenti. Componenti semplici possono essere progettati, codificati e verificati semplicemente.
- Un approccio ortogonale promuove il riuso. Se le componenti hanno una responsabilità semplice e ben definita, possono essere utilizzate con nuove componenti in altri sistemi. Se un sistema ha poche dipendenze è molto semplice riprogettarlo e cambiarlo.
- C'è una notevole crescita della produttività quando vengono combinate componenti ortogonali. Supponiamo che una componente espone N funzionalità distinte, un'altra componente espone altre M funzionalità distinte. Se le componenti sono ortogonali teoricamente si possono ottenere M x N nuove funzionalità.

Riduzione dei rischi :

- Le parti affette da errori in un sistema sono isolate. Se un modulo ha dei difetti, è più difficile che i sintomi si diffondano in tutto il sistema. È più semplice isolare il modulo e correggerlo o sostituirlo con un modulo senza errori.
- Il sistema è meno fragile. Se effettuando piccoli cambiamenti o riparando alcune funzionalità vengono prodotti dei problemi, questi problemi coinvolgeranno solo la componente modificata e non l'intero sistema.
- Il comportamento di un sistema ortogonale è più semplice da verificare. I test sono più semplici da scrivere e sono mirati a verificare solo il comportamento della componente.

[4] *Data access object*: `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`

Nel *team* di un progetto

Se in un *team* di sviluppo non c'è organizzazione e le responsabilità non sono ben definite i componenti non lavorano bene. Ogni cambiamento nel progetto necessita di una riunione di tutti i componenti del *team*, perché chiunque potrebbe essere interessata del cambiamento.

Non esiste un modo per dividere il *team* in gruppi ben organizzati. Dipende dal progetto e dalle analisi delle aree dove potrebbero avvenire più cambiamenti. Un'attività iniziale potrebbe essere quella di separare le interfacce dall'applicazione. Ogni interfaccia (*database*, interfaccia grafica, livello *middleware*, etc.) dovrebbe essere assegnata ad un sotto *team*. Allo stesso modo dovrebbero essere divise le funzionalità dell'applicazione. Alla fine i sottogruppi devono essere aggiustati in accordo con le persone che compongono il progetto.

Un indicatore dell'ortogonalità del *team* di progetto è il numero di persone che sono coinvolte per ogni cambiamento.

Nella progettazione

Molti sviluppatori sentono la necessità di progettare sistemi ortogonali, e utilizzare termini come modulare, basato su componenti, *layer* per descrivere il progetto.

Un sistema deve essere composto da un insieme di moduli che cooperano, ognuno che fornisce delle funzionalità indipendenti.

A volte le parti di un progetto sono suddivise in livelli, che realizzano un livello di astrazione. L'approccio a livelli è un tipo di progettazione che promuove l'ortogonalità. Questo perché ogni livello utilizza le funzionalità esposte dai livelli vicini, in questo modo il sistema diventa più flessibile e il cambiamento su componenti presenti in un livello non costringono a cambiare le componenti degli altri livelli.

Il miglior modo per realizzare progetti ortogonali è quello di utilizzare *pattern* di progettazione. Il *pattern* di progettazione è una soluzione progettuale generale a un problema ricorrente. Un esempio di *pattern* che permette di realizzare sistemi ortogonali è il *pattern MVC*[5].

Mentre si sta programmando

Ogni volta che si scrive del codice si aumenta il rischio di ridurre l'ortogonalità del progetto che si sta sviluppando. Per questo motivo, ogni volta che si produce codice si dovrebbe monitorare non solo il codice che si sta scrivendo ma anche l'intero contesto dell'applicazione. Introducendo delle modifiche al progetto si potrebbe accidentalmente duplicare le funzionalità presenti in altri moduli o fare qualche altro tipo di duplicazione.

Esistono molte tecniche che si possono usare per mantenere un progetto ortogonale:

- Scrivere codice disaccoppiato e che pubblica solo le funzionalità indispensabili. Creare moduli che pubblicano solo l'essenziale e che non dipendano da altri moduli. Un possibile modo per capire se il codice prodotto è scritto in modo disaccoppiato è quello di rispettare la legge di Demeter[6]

- Evitare oggetti globali. Ogni volta che il codice si riferisce ad oggetti globali, va in concorrenza con gli altri oggetti che utilizzano questi dati. In generale il codice è più chiaro e semplice da mantenere se esplicitamente viene passato il contesto direttamente all'oggetto. In un approccio orientato agli oggetti il contesto può essere passato come uno o più parametri di *input* al costruttore di un oggetto.

- Evitare di creare funzionalità simili. Spesso quando si utilizzano un insieme di funzionalità che sembrano avere comportamento simile, forse duplicano lo stesso codice. la duplicazione del codice è sintomo di problemi strutturali.

Si deve quindi sempre analizzare tutto il codice di un progetto. È opportuno migliorare e correggere il codice appena si trova un'errore o un imperfezione in modo da diminuire l'entropia del progetto e aumentare l'ortogonalità.

[5] *Model View Control*

[6] `http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html`

Nei test

Il comportamento di un sistema progettato e sviluppato in modo ortogonale è facile da verificare. Questo perché la comunicazione tra i vari componenti sono formalizzate e limitate. La maggior parte dei test possono essere fatti a livello delle unità. Questa è una buona cosa, perché i test di unità sono più semplici da creare rispetto ad altre tipologie di test. È consigliato che ogni modulo abbia i propri test di unità, e che questi siano eseguiti automaticamente durante il processo di costruzione del progetto (*build*).

Creare test di unità è un modo semplice per capire l'ortogonalità di un sistema. Se risulta abbastanza semplice creare in modo isolato i test per verificare il comportamento delle funzionalità del progetto, significa che il progetto è sviluppato in modo ortogonale. Se invece, per verificare il comportamento di una funzionalità, è necessario coinvolgere più moduli del progetto (e quindi è impossibile isolare alcune funzionalità), significa che alcune parti del progetto non sono ortogonali.

Se in un progetto si trovano dei moduli che non sono disaccoppiati è consigliato eliminare questi difetti il prima possibile per abbassare l'entropia del sistema.

L'attività di correzione degli errori è una buona occasione per misurare l'ortogonalità dell'intero sistema. Se la correzione di un errore obbliga lo sviluppatore a correggere molti moduli significa che il progetto non è ortogonale. Se la correzione di un errore causa la nascita di altri errori allora significa che alcune funzionalità del progetto non sono disaccoppiate e indipendenti.

Per permettere di ricavare una misura sulla ortogonalità del progetto è possibile calcolare il numero di file che vengono modificati per la correzione di ogni errore. Se i file modificati per correggere l'errore sono molti e distribuiti su vari moduli del progetto allora il progetto non è ortogonale. Se la correzione coinvolge pochi file, contenuti in un modulo del progetto allora questo è un'indicatore positivo sull'ortogonalità del progetto.

1.5 Non programmare per coincidenza

Nell'attività di sviluppo, un programmatore è soggetto a commettere errori in ogni momento. Come si può intuire non esiste un modo che assicuri di sviluppare codice senza errori. Di seguito vengono descritte due tipi di atteggiamento che un programmatore può adottare:

- Programmare per coincidenza: atteggiamento sconsigliato che permette ad uno sviluppatore di spendere molto tempo per individuare e correggere errori
- Programmare deliberatamente: atteggiamento consigliato che permette ad uno sviluppatore di produrre del codice più facile da mantenere

Come programmare per coincidenza

Per programmare per coincidenza uno sviluppatore deve seguire le seguenti fasi:

1. Sviluppare una parte del codice del progetto
2. Provare il codice sviluppato
3. Se il codice sembra funzionare riprendere dal punto 1

Se uno sviluppatore produce codice in questo modo, per le prime iterazioni apparentemente non troverà problemi. Con il passare del tempo, dopo aver effettuato molte iterazioni, se viene trovato un errore sarà molto difficile capire dove viene prodotto e come risolverlo. Questo perché sarà difficile capire perché il codice funzionava prima di trovare l'errore, e sarà ancora più difficile trovare la parte del codice affetta dall'errore. Nel punto 2 dell'iterazione il programmatore si limita a provare il codice, senza produrre nessun test che verifica l'assenza di possibili errori e la verifica del comportamento delle funzionalità create nel tempo. In questo modo può succedere che il codice sembra funzionare correttamente, ma in verità funzioni solo per la prova effettuata dallo sviluppatore. Così facendo lo sviluppatore programma per coincidenza, senza essere consapevole che le cose prodotte funzionino correttamente.

Di seguito vengono descritti alcuni incidenti che potrebbero provocare errori nella programmazione per coincidenza.

Incidenti nello sviluppo
Molto spesso accade che gli sviluppatori producono delle funzionalità che non funzionano in modo corretto. Solitamente la causa di questi incidenti può essere la mancanza di documentazione, o l'assenza di test che verificano il comportamento di un componente in condizioni limite.

Supponiamo di utilizzare una funzionalità che è stata sviluppata tramite la programmazione per coincidenza. Supponiamo di passare in *input* ad una funzionalità dei valori non corretti, non considerati dal programmatore che l'ha creata. La funzionalità si comporterà in un modo non previsto dal programmatore, e il codice che utilizzerà questa funzionalità si baserà su questo comportamento (verranno fatte delle assunzioni). Se il programmatore si accorge che la funzionalità, precedentemente realizzata, non si comporta nel modo desiderato, con valori non corretti, potrebbe cambiarla. In questo modo il codice prodotto dagli altri sviluppatori, non funzionerà, poiché le assunzioni fatte verranno a mancare.

Per evitare questi incidenti è quindi necessario fidarsi solo sul comportamento documentato di una funzionalità. Se è necessario utilizzare delle funzionalità che hanno comportamenti non documentati, è buona norma documentare e verificare tramite dei test, le assunzioni che verranno fatte.

Incidenti per assunzioni implicite
A volte capita che gli sviluppatori facciano delle assunzioni e producano codice che si basa su queste. Le assunzioni possono essere fatte ad ogni livello dell'attività di sviluppo, dall'analisi dei requisiti all'attività di verifica. Molto spesso gli sviluppatori operano facendo assunzioni diverse, che spesso sono in conflitto. Raramente queste assunzioni sono documentate e per questo possono verificarsi degli errori. Per evitare questi incidenti è necessario documentare sempre le assunzioni che vengono prese (dalle più banali alle più complesse) in modo che altri sviluppatori possano comprendere il corretto comportamento e utilizzo delle funzionalità realizzata.

Come programmare deliberatamente

Se un programmatore vuole spendere meno tempo per mantenere il codice e per individuare e correggere errori, prima che questi arrivino al cliente, è consigliato che programmi con responsabilità, avendo la certezza che il codice non funzioni per causalità. È consigliato quindi seguire le seguenti linee guida:

- Essere sempre consapevole di ciò che si sta facendo.
- Se per caso non si ha ben compreso un requisito, o il comportamento di una funzionalità, è consigliato non iniziare a produrre codice fino a che non si ha capito cosa si deve sviluppare.
- Seguire sempre una pianificazione. Preferibilmente è consigliato utilizzare strumenti appropriati per pianificare. Lo scopo della pianificazione è identificare e stabilire come procedere e cosa deve produrre l'attività che si sta svolgendo.
- Credere solo nei risultati certi. Non dipendere da assunzioni o coincidenze. È consigliato verificare sempre il comportamento di una funzionalità attraverso dei test (vedi capitolo 3 alla pagina 43).
- Se non è possibile avere un risultato certo, quindi si è obbligati a fare delle assunzioni, è meglio documentare le assunzioni, in modo che anche altri le possano comprendere il comportamento delle funzionalità create.
- Verificare sia il codice che le assunzioni. Se vengono verificate le assunzioni, viene inserito nel codice di test una documentazione implicita. Tramite i test è possibile verificare che se le assunzioni fatte si verificano.
- Dare priorità agli aspetti più importanti per il cliente e per il progetto.
- Pensare sempre al presente. Non scrivere codice che cerca di prevedere e soddisfare requisiti che potranno presentarsi in futuro. È consigliato concentrarsi e risolvere i problemi che sono certi. Se in futuro si presenteranno nuovi requisiti, il codice potrà essere modificato o sostituito per adattarsi ai cambiamenti (vedi 2.1.1).

In questo modo ogni volta che un programmatore ha del codice che sembra funzionare, ma non sa il motivo del funzionamento, deve assicurarsi che non funzioni per coincidenza.

Capitolo 2

Pratiche e strumenti a supporto dello sviluppo

In questo capitolo viene descritto un processo di sviluppo e verifica di un prodotto *software*che cerca di rispettare i principi riportati nel capitolo precedente. Dopo una breve descrizione del processo vengono descritti le pratiche che lo influenzano e gli strumenti necessari per la sua realizzazione al fine di produrre un prodotto *software* sviluppato in linguaggio *Java*.

2.1 Pratiche di processo

Il processo di sviluppo, che viene descritto in questa guida, è rappresentato nella figura 2.1.

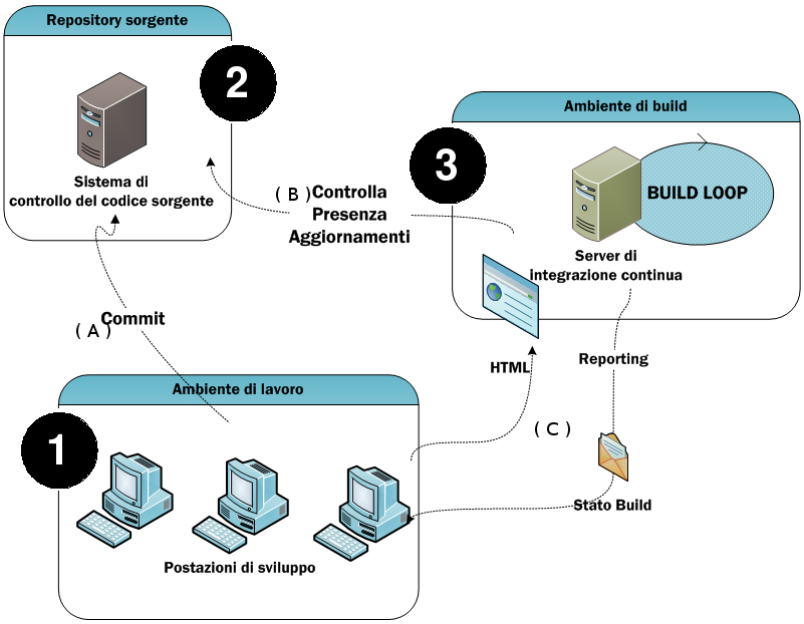

Figura 2.1: Visione generale del processo di sviluppo e verifica di un prodotto *software*

Come si evince dalla figura, il processo è composto da tre componenti. L'ambiente di lavoro (figura 2.1 1) è l'ambiente dove viene sviluppato il prodotto *software* in modo incrementale. Ad ogni realizzazione di una funzionalità, il codice prodotto dagli sviluppatori viene inviato al sistema di controllo del codice sorgente (figura 2.1 A, 2). Questo componente contiene la versione corrente del progetto e ha lo scopo di mantenere uno storico di tutte le modifiche apportate dagli sviluppatori.

Il *server* di integrazione continua (figura 2.1 3, B) controlla periodicamente se sono avvenuti cambiamenti alla versione del prodotto gestita dal sistema di controllo del codice sorgente. Se viene identificata una modifica viene scaricata una copia della versione corrente del prodotto nel ambiente di *build* e viene effettuato il processo

di costruzione e verifica del prodotto. Al termine del processo di costruzione e verifica del prodotto, vengono pubblicati i risultati che vengono consultati dagli sviluppatori (figura 2.1 C). I risultati possono evidenziare che il processo è terminato con successo o la presenza di problemi durante la fase di costruzione e verifica del prodotto. Se i risultati evidenziano dei problemi, gli sviluppatori interromperanno la fase di sviluppo fino a che questi non sono stati corretti.

Di seguito vengono descritte le pratiche che caratterizzano il processo. Nelle successive sezioni di questo capitolo vengono descritti gli strumenti che caratterizzano ogni componente del processo per creare e verificare un prodotto *software* sviluppato in linguaggio *Java*.

2.1.1 *Refactoring*

La pratica di *refactoring* si riferisce all'attività di riscrittura, riprogettazione e riscrittura di alcune parti funzionanti di un progetto. In [12] questa attività viene definita come:

> *"Refactoring is the process of making changes to existing, working code without changing its external behavior. In other words, changing how it does it, but not what it does. The goal is to improve the internal structure."*

La realizzazione di un progetto *software* è un'attività di elevata complessità, soggetta a scadenze temporali e a cambiamenti nel tempo. Per questo un progetto deve essere mantenuto nel tempo e deve adattarsi a nuovi requisiti.

L'attività di *refactoring* è quindi una delle attività fondamentali, svolte da uno sviluppatore, per permettere il successo di un progetto *software* nel tempo. Senza questa attività un progetto tende a diventare presto obsoleto, l'entropia al suo interno aumenta e spesso diventa ingestibile.

Di seguito viene descritto quando è necessario effettuare attività di *refactoring* e alcuni consigli su come effettuare questa attività.

Quando fare attività di *refactoring*

Quando si trova del codice che ha delle problematiche, la cosa più giusta da fare è modificarlo il prima possibile. Le problematiche principali che possono portare a dover effettuare attività di *refactoring* sono:

- Duplicazione: Si scopre una violazione del principio *DRY*.
- Progettazione non ortogonale: Si scopre del codice o una parte del progetto che può essere riprogettata e modificata in modo da migliorare l'ortogonalità del progetto.
- Conoscenza obsoleta: Poiché le cose cambiano, e cambiano anche i requisiti del sistema, è necessario sempre modificare il codice del progetto in modo che sia sincronizzato con i requisiti attuali.
- Prestazioni: Può essere necessario modificare, o spostare una funzionalità per aumentare le prestazioni del sistema

L'attività di *refactoring* viene spesso vista come un'attività pericolosa. Per molti è un'attività critica, che viene sempre posticipata. Per molti viene interpretata come una cosa da evitare, infatti porta a cambiare delle cose che funzionano, con la possibilità di introdurre nuovi errori. L'attività di *refactoring* deve essere vista come un'attività necessaria per migliorare il progetto che si sta sviluppando e diminuire l'entropia al suo interno. Deve essere fatta non appena viene incontrata una problematica, in modo da effettuare attività di *refactoring* per risolvere un problema per volta. Se viene effettuata l'attività di *refactoring* frequentemente, il *team* di sviluppo si abituerà ad effettuare questa attività e sarà vista come un'attività normale.

Nel processo illustrato in figura 2.2 b, questa attività viene svolta ad ogni iterazione della fase di *Design and Code*. In questo modo gli sviluppatori migliorano costantemente la struttura del progetto e prendono confidenza con questa pratica.

Come effettuare attività di *refactoring*

L'attività di *refactoring* consiste nella riprogettazione e riscrittura di parti di un progetto. Questa attività deve essere fatta per migliorare la qualità del progetto in modo da soddisfare i requisiti e facilitarne la manutenzione. Deve essere pianificata, decisa e condivisa da tutto il *team* di sviluppo ed effettuata con molta cautela. Martin Fowler ha riportato nel libro [6] tre consigli per effettuare l'attività di *refactoring*:

1. Non effettuare contemporaneamente attività di *refactoring* e introdurre nuove funzionalità al progetto.
2. Avere dei test che verificano il comportamento delle funzionalità del progetto. In questo modo è possibile riscrivere la funzionalità con le stesse caratteristiche funzionali della precedente. Durante l'attività di *refactoring* è necessario eseguire frequentemente tutti test del progetto. In questo modo si potrà individuare subito la presenza di nuovi errori.
3. Dividere l'attività di *refactoring* in sotto attività ed effettuare un attività alla volta. In questo modo si effettueranno cambiamenti più semplici che coinvolgeranno pochi moduli del progetto. Cosi facendo si riuscirà a localizzare subito gli errori risultati da questa pratica e diminuire l'attività di *debugging*.

Per maggiori informazioni riguardo l'attività di *refactoring* è consigliato consultare:
`http://www.refactoring.com/`

2.1.2 *Test-Driven Development*

Il *Test-Driven-Development* è una pratica utilizzata nelle metodologie agili[1] che influenza l'attività di progettazione e sviluppo di un progetto *software*. Consiste nel suddividere un progetto in piccoli parti. Ogni parte viene realizzata come illustrato in figura 2.2 b, *Design and Code* e si svolge nel seguente modo:

1. Creare un test di unità che verifica un aspetto della componente da realizzare
2. Eseguire tutti i test presenti nel programma e verificare il fallimento dell'ultimo test introdotto
3. Sviluppare il codice di produzione in modo da far superare il test
4. Eseguire tutti i test presenti nel progetto e verificarne il successo
5. Effettuare attività di *refactoring* del codice di produzione e di test

Seguendo questa pratica lo sviluppatore è costretto a sviluppare prima i test e poi il codice di produzione. In questo modo, attraverso i test viene definito il comportamento della componente che successivamente viene realizzata. Così facendo ogni volta che vengono eseguiti tutti i test (possibilmente in modo automatico) viene controllato che, con l'introduzione di nuove modifiche, non sia cambiato il comportamento dell'intero progetto.

Come si può intuire, questa è una pratica che influenza pesantemente l'attività di sviluppo e progettazione e non solo la fase di test. Adottando questa pratica viene creato un sistema di non regressione che permette di effettuare l'attività di *refactoring* in modo semplice e sicuro.

Un'attività fondamentale per adottare la pratica di *Test-Driven Development* è l'attività di test di unità. Questa attività consiste nel verificare il comportamento di una funzionalità atomica di un progetto. Questa attività viene descritta più in dettaglio nel capitolo 3.

2.1.3 *Continuous Integration*

Continuous Integration è una pratica, che può essere adottata durante la realizzazione di un progetto *software*, dove i componenti di un *team* di sviluppo integrano il loro lavoro frequentemente (da una a più volte in un giorno). Ogni integrazione è verificata attraverso l'esecuzione automatica del processo di *build*, dove vengono eseguiti tutti i test, in modo da trovare e correggere il prima possibile gli errori causati durante l'integrazione. Come viene rappresentato in figura 2.1 B, il sistema di *Continuous Integration* controlla, ad intervalli prestabiliti, se sono avvenuti cambiamenti nei sorgenti del progetto presenti nel sistema di versionamento. Ogni

[1] `http://agilemanifesto.org/`

volta che viene trovato un cambiamento viene eseguito il processo di *build* e il risultato viene pubblicato agli sviluppatori.

In questo modo, ogni volta che viene completata un'attività, viene compilato e creato il prodotto e assicurato il superamento di tutti i test. Appena il sistema di *Continuous Integration* segnala un problema (quindi il processo di *build* non termina con successo) il *team* di sviluppo interromperà l'attività fino a che non vengono corretti tutti gli errori.

Gli errori tipici che vengono trovati dal sistema di *Continuous Integration* sono:

- Errori di compilazione dei sorgenti
- Fallimento di un test
- Mancanza di file (dovuta ad una dimenticanza nell'attività di *commit*)
- Errori segnalati dagli strumenti di analisi

Adottando questa pratica si hanno i seguenti benefici:

- Stabilità del progetto: È sempre disponibile una versione funzionante del progetto. Appena viene individuato un errore, l'attività di sviluppo viene sospesa fino a che non viene corretto l'errore.
- Stabilità del processo di *build*: Il processo di *build* deve essere automatico e viene eseguito molto frequentemente. In questo modo è più semplice individuare e correggere errori o imperfezioni presenti nel processo o nelle automazioni che lo realizzano.
- Aumento delle informazioni sullo stato del progetto: Il *team* di sviluppo è sempre informato sullo stato del processo di *build*. È possibile eseguire in questo processo anche degli strumenti di analisi, in modo da misurare e controllare lo stato di qualità del progetto.
- Facile individuazione e isolamento degli errori: Tramite l'utilizzo della integrazione continua è possibile sempre individuare l'attività che introduce l'errore. In questo modo è molto più semplice individuare e isolare l'errore visto che questo è stato introdotto dall'ultima attività e non è presente nel integrazione precedente.

È possibile trovare maggiori informazioni su questa pratica a questo indirizzo:
`http://www.martinfowler.com/articles/continuousIntegration.html`

2.2 *Repository* dei sorgenti

Una delle componenti principali del progetto è il *repository* dei sorgenti. Questa componente ha lo scopo di pubblicare e gestire la versione principale del progetto.

Gestire il codice sorgente di un progetto in un unico archivio ha il vantaggio di diminuire le duplicazioni durante le attività di sviluppo. Tutti gli sviluppatori faranno sempre riferimento alla versione contenuta nel sistema di controllo del codice sorgente in questo modo esisterà sempre e solo un'unica versione ufficiale del prodotto che si sta realizzando. Tutte le modifiche, effettuate dagli sviluppatori, verranno considerate solo quando verranno inviate al sistema di controllo del codice sorgente.

Per realizzare questa componente è necessario utilizzare un'applicazione che permetta di realizzare il sistema di controllo del codice sorgente.

Nelle successive sezioni vengono descritte le caratteristiche del sistema di controllo del codice sorgente e viene descritto un esempio di realizzazione di questo sistema utilizzando *Subversion*.

2.2.1 Sistema di controllo del codice sorgente

Il sistema di versionamento, a volte chiamato *Source Code Control*, è uno degli elementi fondamentali per la realizzazione di un progetto. Offre vantaggi sia per lo sviluppo di un progetto individuale che per lo sviluppo di un progetto da parte di un *team* di sviluppo:

- Mette a disposizione al *team* di un progetto una funzionalità simile al "*UNDO*", presente in molti *editor*, che permette di recuperare il lavoro realizzato in passato da ogni componente del *team*. Avendo la possibilità di ritornare in dietro nel tempo, gli sviluppatori, quando commettono degli errori in un'attività, possono ritornare ad una versione consistente e ripartire nuovamente con l'attività, oppure confrontare le due versioni per individuare e isolare più velocemente l'errore.

- Permette a più sviluppatori di lavorare contemporaneamente sullo stesso codice in modo sicuro. Le versioni prodotte verranno salvate nel tempo, in questo modo, quando qualcuno sovrascriverà del codice realizzato in passato, questo non verrà perso perché sarà sempre possibile scaricare la versione del progetto prima delle modifiche.

- Il sistema di versionamento memorizza tutti i cambiamenti avvenuti durante la vita di un progetto. Quando vengono trovati degli errori o del "codice sospetto" è sempre possibile risalire a chi ha introdotto gli errori, quando è avvenuto il cambiamento e le motivazioni del cambiamento.

- Permette di supportare versioni multiple del *software* contemporaneamente. È possibile creare e gestire rilasci diversi dello stesso prodotto. Utilizzando questo strumento non è necessario che il *team* di sviluppo si fermi quando si sta rilasciando una nuova versione del prodotto.

- Può essere visto come una macchina del tempo che permette di ritornare in una specifica data del passato e vedere come era il progetto a quel tempo. Permette di fare statistiche sull'avanzamento del progetto, ed è essenziale per identificare problemi presenti in versioni passate che sono ancora utilizzate dagli utenti.

Cosa gestire con il sistema di versionamento

Nel sistema di versionamento è consigliato gestire tutti i file che possono essere cambiati direttamente dagli sviluppatori e quindi:

- Tutti i file sorgenti
- Tutti i file sorgenti di test
- Tutti i file che permettono di creare il progetto (file di *build*)
- Tutti i file di documentazione del progetto
- A volte tutte le librerie utilizzate dal progetto

Non devono essere gestiti nel sistema di versionamento tutti i file che possono essere prodotti dai file gestiti dal sistema di versionamento e quindi:

- Tutti i compilati
- Gli archivi creati per distribuire il prodotto
- I file di documentazione creati automaticamente dal codice (p.es. i file *HTML* creati tramite il comando `javadoc`)
- I risultati dei test
- Tutti i file che possono essere creati tramite elaborazioni dei file precedentemente citati

Termini utilizzati in un sistema di versionamento

In questa sezione vengono descritti i termini che verranno utilizzati per descrivere il funzionamento del sistema di controllo del codice sorgente.

Repository: Luogo centralizzato dove vengono salvate le copie principali di tutte le versioni dei file che compongono un progetto. Vista la notevole importanza del *repository* è consigliato collocarlo in una macchina sicura e accessibile a tutti i componenti del *team*. È consigliato salvare regolarmente delle copie di *backup* del

repository. Il *repository* normalmente è accessibile anche remotamente, in modo che i componenti del *team* di sviluppo, in base ai permessi d'accesso, possano accedere a tutto il codice del progetto attraverso la rete.

Working copy: È una copia locale dei file, presenti nel *repository*, che serve ad un componente del *team* di sviluppo per lavorare. Nei progetti di piccole medie dimensioni la *working copy* può contenere tutti i file presenti nel *repository*. Nei progetti di grandi dimensioni la *working copy* è un sottoinsieme dei file presenti nel *repository* (p.es. uno o più moduli che compongono il progetto).

Checking out: È l'azione che permette di prelevare delle copie di file dal *repository* nella *working copy*. Il processo di *check out* permette di avere una copia aggiornata dei file richiesti; I file richiesti sono copiati in una *directory* replicando la struttura del *repository*. Anche se può succedere di lavorare nella stessa macchina che contiene il *repository* del progetto, si deve lo stesso creare una *working copy* tramite un azione di *checking out*.

Committing: Mentre si sta lavorando ad un progetto si effettuano dei cambiamenti nella propria *working copy*. Ogni volta che si vuole fare un salvataggio delle funzionalità e dei cambiamenti realizzati si deve effettuare un azione di *committing*. Mentre si stanno facendo dei cambiamenti nella propria *working copy* più persone del *team* possono effettuare dei cambiamenti negli stessi file ed effettuare azione di *committing*. I cambiamenti apportati dagli altri componenti, che vengono salvati nel *repository*, ma non vengono riportati anche nella *working copy* locale. Perciò prima di effettuare azione di *committing* si deve aggiornare la *working copy* alla versione corrente, presente nel *repository* e verificare che non ci siano delle collisioni. l'aggiornamento della *working copy* locale avviene tramite un azione detta ***update***.

Versioning: Oltre a salvare ogni modifica apportata in un file, da uno sviluppatore, nella copia presente nel *repository*, il sistema di versionamento aggiunge un etichetta incrementale che specifica la versione dei file. Esistono due tipologie di versionamento:

- Numerazione specifica per file: In questa tipologia di versionamento viene tenuta traccia delle versioni di ogni singolo file presente nel *repository*. Se per esempio uno sviluppatore effettua un'azione di *checking out* sul file pippo.txt e sul file pluto.txt e nel *repository* questi file erano stati etichettati come pippo.txt versione 1.2 e pluto.txt versione 1.4. Se lo sviluppatore modifica entrambe i file ed effettui un azione di *committing* (su entrambe i file) nel *repository* verranno salvati come pippo.txt versione 1.3 e pluto.txt versione 1.5.

- Numerazione specifica per *repository*: in questa tipologia di versionamento il *repository* parte dalla versione 0. Ogni volta che uno sviluppatore fa un azione di *committing* in uno o più file, il numero di versione del *repository* viene incrementato di un unità. In questo modo una versione identifica tutti i file contenuti nel *repository* in quella versione. Per esempio, in un progetto composto da due file pippo.txt e pluto.txt versione 0, se uno sviluppatore fa attività di *checking out* del file pluto.txt ed effettua una modifica ed effettua una azione di *committing*, tutti i file del progetto verranno etichettati alla versione 1. In questo modo uno sviluppatore si riferirà alla versione del progetto e non alla versione del file. Questo metodo è utilizzato anche da *subversion*.

Tag: Il numero di versionamento è incrementale. Tipicamente è più semplice ricordare dei nomi che ricordare dei numeri. Per questo motivo il sistema di versionamento permette di assegnare un nome ad un particolare gruppo di file. In questo modo è possibile fare azioni di *checking out*, specificando il nome del *tag* per ricavare il gruppo di file etichettato con uno specifico nome. I *tag* vengono utilizzati per tener traccia di particolari versioni di un progetto.

Trunk: Più sviluppatori lavorano contemporaneamente su più file di un progetto. Ogni sviluppatore effettuerà azione di *checking out*, effettuerà delle modifiche, ed effettuerà azione di *checking in*. In questo modo ogni sviluppatore potrà condividere il lavoro realizzato. Il progetto principale, dove sono contenute il progetto e tutte le modifiche inviate dagli sviluppatori viene chiamato *trunk*.

Branching: Si consideri il momento di un nuovo rilascio del progetto. Un parte del *team* di sviluppo, sarà impegnato nella realizzazione del rilascio, e collaborerà con i *release engineers* e il *QA team*. Durante questo periodo, la parte del *team* coinvolta nella realizzazione del rilascio del progetto avrà bisogno di stabilità, e nessun altro sviluppatore potrà apportare nuove funzionalità al progetto. Questo comporta:

- Bloccaggio del lavoro degli altri *team* di sviluppo

- Impossibilità di dare supporto alla nuova versione rilasciata

Il sistema di versionamento permette di effettuare l'attività di *Branching*. Questa attività consiste nel dividere il progetto in due parti. Una parte principale (*trunk*) e una parte indipendente che può essere vista dal *team* di sviluppo come un progetto a se stante. In questo modo quando gli sviluppatori devono rilasciare un nuovo rilascio effettueranno un azione di *branching*, creando cosi una copia indipendente del *trunk* corrente. In questa nuova copia, gli sviluppatori potranno effettuare azioni di *committing* senza modificare il codice del *trunk* principale e viceversa. Così facendo il *team* di sviluppo e il *team* di release potranno lavorare in modo parallelo e indipendente in due progetti distinti. Quando saranno distribuiti più rilasci, e un utente troverà un errore, gli sviluppatori potranno recuperare dal *repository* il *branch* del progetto che è utilizzato dall'utente e dare supporto (quindi effettuare modifiche e operazioni di *committing*) senza modificare il *trunk* principale e gli altri rilasci.

2.2.2 Gestione di un progetto tramite *Subversion*

Subversion è un sistema di versionamento *open source*[2] nato a fine agosto del 2001, reperibile all'indirizzo `http://subversion.tigris.org/`, che è stato creato con lo scopo di sostituire l'utilizzo di *CVS* nelle comunità *open source*. Permette di adottare un sistema di controllo del codice sorgente in modo semplice. Nelle prossime sezioni vengono forniti degli esempi che hanno lo scopo di descrivere le principali funzionalità di questo strumento.

Installazione di *Subversion*

Per iniziare ad utilizzare *subversion* è necessario installarlo. Per controllare se *subversion* è già presente nel sistema, da un pront dei comandi eseguire il comando:

```
svn --version
```

Se il *client* di *subversion* è installato correttamente verrà notifica la versione del programma.

```
svn, versione 1.4.4 (r25188)
   compilato Sep 28 2007, 10:50:44

...
```

Per controllare se gli strumenti di amministrazione di *subversion* sono installati correttamente nel sistema, da un pront dei comandi eseguire il comando:

```
svnadmin --version
```

Se gli strumenti di amministrazione di *subversion* sono installati correttamente allora è possibile procedere con la sezione successiva, altrimenti seguire la guida all'installazione a questo indirizzo: `http://subversion.tigris.org/getting.html`.

Creazione di un *repository*

Come descritto precedentemente, *subversion* necessita di un *repository* dove salvare e versionare il progetto. In questa sezione viene descritto come creare un *repository* dove verrà salvato e gestito il progetto d'esempio che verrà utilizzato in tutta la guida. Creare una *directory* vuota (`/home/nicola/svn-repo`) dove verrà posizionata il *repository*:

```
mkdir /home/nicola/svn-repo
```

Creare attraverso il comando `svnadmin create` il *repository* nella *directory* appena creata:

```
svnadmin create /home/nicola/svn-repo
```

Al termine della esecuzione del comando la *directory* creata verrà popolata con alcuni file. In questa guida si utilizzerà la *directory* contenente il *repository* come un *black-box* e non si andrà ad analizzare i file che essa contiene.

[2] Un programma è detto *open source* se la sua licenza di distribuzione è approvata dalla *Open Source Initiative*

Creazione del progetto

Di seguito viene creato il *repository* con un nuovo progetto che realizza un programma di calcolo del punteggio di una partita di bowling. In questa sezione viene descritto come creare il progetto, che si chiamerà BowlingScore (bowlingscore nel *repository*), e vengono inseriti alcuni file sotto il controllo del sistema di versionamento.

Per procedere con l'esempio è necessario creare una *directory* temporanea chiamata `tmpdir`. All'interno di questa *directory*, utilizzando un *editor* di testo, creare i file `README.txt` e `LICENSE.txt` contenenti il testo:

Listing 2.1: README.txt

```
1 README TODO
```

Listing 2.2: LICENSE.txt

```
1 LICENSE TODO
```

Anche se questi file non sono completi e non sono codice sorgente del progetto, serviranno per la documentazione, quindi devono essere gestiti dal sistema di versionamento.

Per importare questi file nel *repository* del progetto è necessario posizionarsi nella *directory* `tmpdir` (dove sono presenti i file) ed eseguire il comando:

```
tmpdir> svn import -m "importo il progetto BowlingScore" \
file:///home/nicola/svn-repo/bowlingscore/trunk
Aggiungo       LICENSE.txt
Aggiungo       README.txt
Commit della Revisione 1 eseguito.
```

Il comando `import` permette di importare alcuni file nel *repository* specificata. l'opzione `-m` permette di associare un messaggio all'azione di `import`. L'ultimo parametro specifica la locazione del *repository* dove si vogliono inserire i file.

Dopo che il comando è stato eseguito, *subversion* notifica l'inserimento dei file specificati nel *repository* e ne riporta la versione.

Dopo l'esecuzione del precedente comando i file sono salvati all'interno del *repository*. Di seguito verrà descritto come recuperare una *working copy* dal *repository*.

Iniziare a lavorare con un progetto

Per iniziare a lavorare con qualsiasi progetto gestito dal sistema di versionamento è necessario:

- Decidere dove posizionare la *working copy*
- Effettuare l'azione di *cheking out* del progetto dal *repository* nella cartella destinata a contenere la *working copy*

In questo esempio viene salvata la *working copy* in una cartella chiamata `work`. Prima di effettuare l'azione di *cheking out* è necessario creare la la *directory* `work`:

```
mkdir /home/nicola/work
```

Dopo di che è possibile effettuare l'azione di *checking out* del progetto nella *directory* `work`:

```
cd /home/nicola/work
work> svn co file:///home/nicola/svn-repo/bowlingscore/trunk bowlingscore
A    bowlingscore/LICENSE.txt
A    bowlingscore/README.txt
Estratta revisione 1.
```

l'argomento `co` specifica che il *client* di *subversion* deve effettuare un azione di *checking out*. Il secondo argomento specifica i file che si vogliono copiare dal *repository*. l'ultimo argomento specifica in quale *directory* si vogliono salvare i file che verranno copiati dal *repository*. Tramite questi comandi è stata creata una *working copy* del progetto BowlingScore nella cartella `/home/nicola/work/bowlingscore`. Da questo momento è possibile lavorare con i file copiati nella *working copy*, e visto che *subversion* sta gestendo correttamente i file di questo progetto, è possibile eliminare la cartella temporanea precedentemente creata (`tmpdir` contenente la copia iniziale di `README.txt` e `LICENSE.txt`).

Effettuare una modifica

Supponiamo di dover iniziare a modificare il file `README.txt`. Tramite un *editor* di testo aprire il file `/home/nicola/work/bowlingscore/README.txt` e modificarlo come segue:

Listing 2.3: /home/nicola/work/bowlingscore/README.txt

```
1 README TODO
  Bowling Score - README
3 17 Apr 2008
```

Dopo aver salvato i cambiamenti, è possibile interrogare *subversion* per ottenere informazioni sullo stato corrente di uno o più file contenuti nella *working copy*. Per ottenere informazioni sullo stato dei file è necessario posizionarsi nella *directory* `/home/nicola/work/bowlingscore/`ed eseguire il comando:

```
cd /home/nicola/work/bowlingscore
bowlingscore> svn status README.txt
M      README.txt
```

Subversion riporterà un messaggio dove viene notificato, attraverso una M, che il file `README.txt` è stato modificato. Se si dimentica il motivo per cui il file è stato modificato, è possibile eseguire il comando `diff` che riporta i cambiamenti apportati ad un file presente nella *working copy* rispetto al file presente nel *repository*:

```
bowlingscore> svn diff README.txt
Index: README.txt
===================================================================
--- README.txt  (revisione 1)
+++ README.txt  (copia locale)
@@ -1 +1,3 @@
 README TODO
+Bowling Score - README
+17 Apr 2008
```

La notifica riportata consiste:

- La prima riga specifica il nome del file analizzato
- Le righe seguenti specificano il numero della revisione presente nel *repository* e il file che si sta confrontando (in questo caso il file nella *working copy*)
- La riga che inizia con "@@ -1 +1,3 @@" specifica in che riga sono avvenute le modifiche
- Nelle successive righe vengono specificate con un + le righe che sono state modificate, con un - quelle che sono state rimosse

Aggiornare il *repository*

Visto che sono state apportate delle modifiche al file `README.txt` è possibile salvare e versionare il file nel *repository*. Per un *team* di progetto composto da un singolo individuo è necessario posizionarsi nella *working copy* del progetto:

```
cd /home/nicola/work/bowlingscore
```

ed effettuare l'azione di *committing*:

```
bowlingscore>  svn commit -m "Inserito titolo e data nel file README.txt"
Trasmetto      README.txt
Trasmissione dati .
Commit della Revisione 2 eseguito.
```

Il comando `commit` permette di salvare ogni cambiamento effettuato (nella *working copy*) nel *repository*. L'opzione `-m` è utilizzata per inserire un messaggio di descrizione al azione di *committing*. Al termine dell'operazione *subversion* notifica i file che sono stati registrati nel *repository* (solo `README.txt` visto che `LICENSE.txt` non è stato modificato) e notifica il nuovo numero di revisione dell'intero *repository*. E' importante notare che se fosse stato modificato anche il file `LICENSE.txt`, visto che il versionamento di *subversion* è specifico per *repository* e non per file, il numero di revisione del *repository* sarebbe stato sempre 2.

Per verificare che l'azione di committing è avvenuta con successo è possibile eseguire il comando:

```
bowlingscore> svn log README.txt
------------------------------------------------------------------------
r2 | nicola | 2008-04-17 10:13:53 +0200 (gio, 17 apr 2008) | 1 line

Inserito titolo e data nel file README.txt
------------------------------------------------------------------------
r1 | nicola | 2008-04-16 17:50:07 +0200 (mer, 16 apr 2008) | 1 line

importo il progetto BowlingScore
------------------------------------------------------------------------
```

Dalla notifica prodotta da questo comando si può ricavare che nicola è lo sviluppatore che ha effettuato l'ultima modifica nella revisione 2 (r2). Come viene riportato nella notifica, il file `README.txt` è stato modificato anche nella revisione 1 dallo sviluppatore nicola. Tramite il seguente comando:

```
svn log --verbose README.txt
```

è possibile ottenere la lista dei file che sono stati aggiunti, modificati o rimossi in ogni versione del progetto.

Gestione delle collisioni

Di norma un *team* di sviluppo di un progetto è composto da più di una persona. Poiché *subversion* non permette di modificare i file in mutua esclusione, tra gli sviluppatori del *team* di sviluppo, possono verificarsi delle collisioni. In questa sezione viene descritto come lavorare parallelamente allo sviluppo dello stesso file, e come *subversion* gestisce le collisioni che possono verificarsi.

Subversion, quando si verificano dei conflitti, cerca di verificare che differenti *working copy* siano consistenti con la copia presente nel *repository*. Per creare un esempio di conflitto vengono utilizzati due utenti con i dati descritti nella tabella 2.1:

Utente	***Working copy***
nicola	`/home/nicola/work/bowlingscore`
davide	`/home/davide/wcopy/bs`

Tabella 2.1: utenti e *working copy* esempio *subversion*

Autenticarsi nel sistema con l'utente davide ed effettuare le seguenti operazioni: Creare la *directory* contenente la *working copy*

```
mkdir /home/davide/wcopy
```

Posizionarsi nella *working copy* ed effettuare l'azione di *checking out* del progetto

```
cd /home/davide/wcopy
wcopy>  svn co file:///home/nicola/svn-repo/bowlingscore/trunk bs
A    bs/LICENSE.txt
A    bs/README.txt
Estratta revisione 2.
```

in questo modo è stata creata una *working copy* del progetto BowlingScore (revisione 2) per lo sviluppatore davide nella *directory* `/home/davide/wcopy/bs`.
Supponiamo che entrambe gli sviluppatori vogliano modificare il file `README.txt` del progetto. L'utente nicola modifica il file aggiungendo una sezione "DESCRIZIONE GENERALE".

Listing 2.4: /home/nicola/work/bowlingscore/README.txt

```
1 README TODO
  Bowling Score - README
3 17 Apr 2008

5 DESCRIZIONE GENERALE
```

ed effettua attività di *committing* per inviare i cambiamenti nel *repository*:

```
bowlingscore> svn commit -m "Inserita sezione DESCRIZIONE GENERALE a README.txt"
Trasmetto      README.txt
Trasmissione dati .
Commit della Revisione 3 eseguito.
```

Dopo questa azione la versione dello sviluppatore davide non è sincronizzata con la versione nel *repository*. Per verificare il cambiamento l'utente davide deve posizionarsi nella *working copy* del progetto (`/home/davide/wcopy/bs`) ed eseguire il seguente comando:

```
cd /home/davide/wcopy/bs
bs> svn status --show-updates
       *        2   README.txt
Stato rispetto alla revisione:      3
```

Tramite questo comando *subversion* verifica se sono disponibili delle versioni più recenti della versione presente nella *working copy* del utente davide. l'asterisco notifica che nel *repository* è presente una versione più aggiornata (il file `README.txt` presente nella *working copy* locale è alla versione 2 mentre la versione nel *repository* è la 3).

Per verificare le differenze tra il file `README.txt` presente nella *working copy* del'utente davide e quella presente nel *repository* si deve eseguire il seguente comando:

```
bs> svn diff -rHEAD README.txt
Index: README.txt
===================================================================
--- README.txt  (revisione 3)
+++ README.txt  (copia locale)
@@ -1,5 +1,3 @@
 README TODO
 Bowling Score - README
 17 Apr 2008
-
-DESCRIZIONE GENERALE
```

L'opzione `-rHead` permette di specificare a *subversion* di verificare le differenze tra il file presente nella *working copy* e il file corrente presente nel *repository* (revisione 3).

Visto che lo sviluppatore nicola ha effettuato delle modifiche lo sviluppatore davide può sincronizzarsi con il file presente nel *repository* tramite il comando:

```
bs> svn update
U    README.txt
Aggiornato alla revisione 3.
```

In questo modo i file del progetto presenti nella *working copy* dello sviluppatore davide vengono aggiornati (notificato tramite la lettera U) e sincronizzati con quelli presenti nel *repository*.

Risoluzione dei conflitti

In questa sezione viene descritto cosa succede quando due persone modificano lo stesso file contemporaneamente. In *subversion*, durante la modifica da parte di due sviluppatori sullo stesso file, possono verificarsi due situazioni:

1. I cambiamenti non riguardano le stesse parte del file
2. I cambiamenti riguardano le stesse parti del file

Di seguito viene ricreata una situazione dove i cambiamenti non riguardano le stesse parti di un file.

Supponiamo che lo sviluppatore nicola modifica la prima riga del file `README.txt`.

Listing 2.5: /home/nicola/work/bowlingscore/README.txt

```
1 README.txt TODO
  Bowling Score - README
3 17 Apr 2008

5 DESCRIZIONE GENERALE
```

Supponiamo che contemporaneamente lo sviluppatore davide modifica la quinta riga del file `README.txt`.

Listing 2.6: /home/davide/wcopy/bs/README.txt

```
1 README TODO
  Bowling Score - README
3 17 Apr 2008

5 DESCRIZIONE DEL PROGETTO
```

In questo modo si è simulato un cambiamento di un file (README.txt revisione 3) da parte di due sviluppatori in due *working copy* distinte. I due cambiamenti sono indipendenti e i due sviluppatori non hanno ancora effettuato l'azione di *committing*. Supponiamo che lo sviluppatore davide effettua azione di *committing* per primo:

```
cd /home/davide/wcopy/bs
bs>  svn commit -m "Modificata la sezione DESCRIZIONE GENERALE\
4 in DESCRIZIONE DEL PROGETTO del file README.txt"
Trasmetto      README.txt
Trasmissione dati .
Commit della Revisione 4 eseguito.
```

e successivamente anche lo sviluppatore nicola effettui azione di *committing*:

```
cd /home/nicola/work/bowlingscore
bowlingscore> svn commit -m "Modificata la prima riga del file README.txt"
Trasmetto      README.txt
svn: Commit fallito (seguono dettagli):
svn: Scaduta: '/bowlingscore/trunk/README.txt' nella transazione '4-1'
```

Subversion notifica che sta cercando di effettuare l'azione di *committing* ma il file README.txt è scaduto. È quindi necessario effettuare un azione di *update* per sincronizzare il file dello sviluppatore nicola con quello presente nel *repository*:

```
bowlingscore> svn update
G    README.txt
Aggiornato alla revisione 4.
```

Subversion notifica (tramite la lettera G) che è stata aggiornata la *working copy* dello sviluppatore nicola e che nel file README.txt sono stati aggiunti i cambiamenti presenti nel *repository*. Per verificare le modifiche apportate da *subversion* aprire, con un *editor* di testo, il file README.txt presente nella *working copy*.

Listing 2.7: /home/nicola/work/bowlingscore/README.txt

```
1 README.txt TODO
  Bowling Score - README
3 17 Apr 2008

5 DESCRIZIONE DEL PROGETTO
```

Come si può notare la riga 1 contiene i cambiamenti effettuati dallo sviluppatore nicola, mentre la riga quattro contiene i cambiamenti dello sviluppatore davide. Come si può notare in queste modifiche non sono avvenute collisioni e *subversion* ha correttamente apportato le modifiche effettuate dai due sviluppatori al file README.txt.

Lo sviluppatore nicola potrà quindi effettuare l'azione di *committing* per apportare le modifiche al progetto.

```
bowlingscore> svn commit -m "Modificata la prima riga del file README.txt"
Trasmetto      README.txt
Trasmissione dati .
Commit della Revisione 5 eseguito.
```

La prossima volta che lo sviluppatore davide inizierà a sviluppare potrà effettuare l'operazione di *update* per sincronizzarsi con la versione contenuta nel *repository*.

```
bs> svn update
U    README.txt
Aggiornato alla revisione 5.
```

Nel caso precedente due sviluppatori hanno modificato due parti distinte dello stesso file. Questo non ha comportato grossi problemi e *subversion* è riuscito a ricostruire in modo corretto il file modificato (README.txt). Di seguito viene analizzato il caso in cui due sviluppatori effettuano dei cambiamenti alla stessa parte di un file e quindi si presenti una collisione tra le due versioni.

Supponiamo che lo sviluppatore nicola voglia modificare la data del file README.txt nella forma *"yyyy month dd"* e che lo sviluppatore davide voglia modificare la data nella forma *"month dd, yyyy"*.

Lo sviluppatore nicola quindi modificherà il file README.txt presente nel suo *working copy* nel seguente modo:

Listing 2.8: /home/nicola/work/bowlingscore/README.txt

```
README.txt TODO
Bowling Score - README
2008 Apr 17

DESCRIZIONE DEL PROGETTO
```

Supponiamo che lo sviluppatore nicola non effettui azione di *committing*.

Lo sviluppatore davide modificherà il file `README.txt` presente nel suo *working copy* nel seguente modo:

Listing 2.9: /home/davide/wcopy/bs/README.txt

```
README.txt TODO
Bowling Score - README
Apr 17, 2008

DESCRIZIONE DEL PROGETTO
```

Supponiamo che lo sviluppatore davide effettui azione di *committing* per primo:

```
bs> svn commit -m "Modificata data README.txt 'month dd, yyyy'"
Trasmetto      README.txt
Trasmissione dati .
Commit della Revisione 6 eseguito.
```

Supponiamo ora che anche lo sviluppatore nicola effettui azione di *committing* (per secondo):

```
bowlingscore> svn commit -m "Modificata data README.txt 'yyyy month dd'"
Trasmetto      README.txt
svn: Commit fallito (seguono dettagli):
svn: Scaduta: '/bowlingscore/trunk/README.txt' nella transazione '6-1'
```

Come accaduto precedentemente, *subversion* avvisa che il file `README.txt` deve essere aggiornato prima di effettuare azione di *committing* visto che la versione nella *working copy* è precedente rispetto alla versione contenuta nel *repository*. Per risolvere questo problema lo sviluppatore nicola deve effettuare un azione di aggiornamento:

```
bowlingscore> svn update
C    README.txt
Aggiornato alla revisione 6.
```

Subversion notifica che è stato aggiornato il file `README.txt` alla versione 6 presente nel *repository*, in oltre segnala un conflitto tra la versione presente nel *working copy* e la versione presente nel *repository* (tramite una C). L'aggiornamento ha cancellato il lavoro svolto dallo sviluppatore nicola? la risposta è ovviamente negativa. Se si apre, con un *editor* di testo, il file `README.txt` presente nel *working copy* dello sviluppatore nicola si potrà vedere il seguente contenuto:

Listing 2.10: /home/nicola/work/bowlingscore/README.txt

```
README.txt TODO
Bowling Score - README
<<<<<<< .mine
2008 Apr 17
=======
Apr 17, 2008
>>>>>>> .r6

DESCRIZIONE DEL PROGETTO
```

Non potendo risolvere il conflitto, *subversion* riporta il contenuto presente nel file `README.txt` del *working copy* corrente e il contenuto presente nel *repository* versione 6 (versione corrente). Le `<<<<<<<` e le `>>>>>>>` indicano in che punto del file è stato trovato il conflitto. Per permettere di capire le cause del conflitto, lo sviluppatore nicola può effettuare il seguente comando:

```
bowlingscore> svn log -r6 README.txt
------------------------------------------------------------------------
r6 | davide | 2008-04-17 14:39:34 +0200 (gio, 17 apr 2008) | 1 line

Modificata data README.txt 'month dd, yyyy'
------------------------------------------------------------------------
```

Tramite questo comando è possibile ottenere il nome dell'autore che ha effettuato la revisione 6 (in questo caso davide) e il messaggio associato all'attività di *committing*.

In questo caso i due sviluppatori possono mettersi d'accordo e decidere come deve essere espressa la data nel file `README.txt` (p.es. *"month dd yyyy"*) quindi lo sviluppatore nicola potrà modificare il file `README.txt` (togliendo le segnalazioni di conflitto) nel seguente modo:

Listing 2.11: /home/nicola/work/bowlingscore/README.txt

```
1 README.txt TODO
  Bowling Score - README
3 Apr 17 2008

5 DESCRIZIONE DEL PROGETTO
```

Quindi ora lo sviluppatore nicola può comunicare a *subversion* che ha risolto il conflitto tramite il seguente comando:

```
bowlingscore> svn resolved README.txt
Risolto lo stato di conflitto di 'README.txt'
```

ed effettuare azione di *committing*

```
bowlingscore> svn commit -m "Modificata data README.txt 'month dd yyyy'"
Trasmetto      README.txt
Trasmissione dati .
Commit della Revisione 7 eseguito.
```

In questo modo il conflitto è stato risolto e nel *repository* è stata salvata la versione corretta del progetto.

Aggiunta di file e *directory* al sistema di versionamanto

Si supponga che lo sviluppatore nicola voglia aggiungere una *directory* contenente i sorgenti del progetto. Per effettuare questa operazione posizionarsi sul *working copy* del progetto BowlingScore.

```
cd /home/nicola/work/bowlingscore
```

Creare una *directory* src (vedi sezione 5.4.1 pagina 88) contenente i seguenti file:

```
src
src/main
src/main/java
src/main/java/it
src/main/java/it/unipd
src/main/java/it/unipd/app
src/main/java/it/unipd/app/App.java
src/test
src/test/java
src/test/java/it
src/test/java/it/unipd
src/test/java/it/unipd/app
src/test/java/it/unipd/app/AppTest.java
```

Eseguire il comando che permette di creare la *directory* `src` nel *repository*

```
bowlingscore>svn add src
A         src
A         src/main
A         src/main/java
A         src/main/java/it
A         src/main/java/it/unipd
A         src/main/java/it/unipd/app
A         src/main/java/it/unipd/app/App.java
A         src/test
A         src/test/java
A         src/test/java/it
A         src/test/java/it/unipd
A         src/test/java/it/unipd/app
A         src/test/java/it/unipd/app/AppTest.java
```

Tramite il comando `add` si notifica a *subversion* di voler aggiungere la *directory* `src`, e il suo contenuto nel *repository*. In questo modo *subversion* memorizza tutti i nomi dei file che si vogliono aggiungere ma non vengono effettivamente aggiunti nel *repository*.

Per aggiungere i file nel *repository* è necessario effettuare un azione di *committing*

```
svn commit -m "Creata cartella sorgenti bowlingscore"
Aggiungo       src
Aggiungo       src/main
Aggiungo       src/main/java
Aggiungo       src/main/java/it
Aggiungo       src/main/java/it/unipd
Aggiungo       src/main/java/it/unipd/app
Aggiungo       src/main/java/it/unipd/app/App.java
Aggiungo       src/test
Aggiungo       src/test/java
Aggiungo       src/test/java/it
Aggiungo       src/test/java/it/unipd
Aggiungo       src/test/java/it/unipd/app
Aggiungo       src/test/java/it/unipd/app/AppTest.java
Trasmissione dati ..
Commit della Revisione 8 eseguito.
```

In questo modo vengo aggiunti tutti i file, contenuti nella cartella `src`, nel *repository* che viene revisionato alla versione 8.

2.3 Ambiente di lavoro

L'ambiente di lavoro è il luogo dove gli sviluppatori realizzano il prodotto. Questo ambiente è composto da delle postazioni di lavoro che devono possedere tutti gli strumenti che permettono la realizzazione di un prodotto in linguaggio *Java* e permettere l'accesso e la gestione del codice del prodotto gestito dal sistema di controllo del codice sorgente. Il processo di sviluppo che viene adottato da questo processo è influenzato dalla pratica di *Test-Driven Development* e viene rappresentato nella figura 2.2 b.

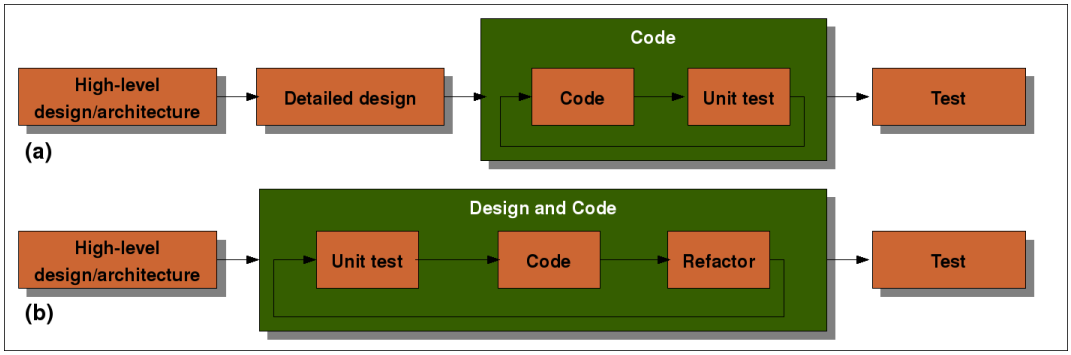

Figura 2.2: Processo di sviluppo: (a) tradizionale *test-last* (b) *test-driven development* (pag. 78 di [19])

A differenza dell'approccio tradizionale (figura 2.2 a), che prevede la fase di progettazione architetturale generale e progettazione in dettaglio prima della fase di codifica, il processo di sviluppo che deriva dall'adozione della pratica di *Test-Driven Development*, prevede solo la fase di progettazione architetturale generale. La fase di progettazione in dettaglio viene spostata all'interno della fase di codifica e viene rivista e migliorata ad ogni iterazione di questa fase. In entrambe i processi, alla fine della realizzazione del prodotto, avviene la fase di test di sistema.

Nelle successive sezioni viene descritto uno strumento che permette di realizzare i test di unità in linguaggio *Java* e vengono dati alcuni consigli su come gestire la conoscenza in un progetto.

Nel capitolo 3 viene descritto come realizzare i test di unità, attività necessaria per poter iniziare ad utilizzare il processo di sviluppo precedentemente descritto. Nel capitolo 4 viene descritto come realizzare parti di un progetto seguendo il processo descritto dalla figura 2.2 b.

2.3.1 *JUnit*

In questa sezione vengono descritte le principali caratteristiche della libreria *JUnit*[3], utilizzata nei successivi capitoli per sviluppare alcuni test di unità di esempio. *JUnit* è un *framework* per sviluppare test di unità ripetibili,

[3] `http://www.junit.org/`

sviluppato da Erich Gamma e Kent Beck. Appartiene alla famiglia di *framework* di test *XUnit*[4] e permette di realizzare test di unità in linguaggio *Java*.

Tutti i test di unità realizzati con *JUnit* (e in qualsiasi *framework* della famiglia *XUnit*) seguono il seguente processo:

1. Riproduzione di tutte le condizioni che servono per eseguire il test (creazione e inizializzazione di oggetti, allocazione di risorse, etc.)
2. Esecuzione dell'unità che deve essere testata
3. Verifica del comportamento dell'unità testata
4. Pulizia dell'ambiente di test

Il *framework* di test *JUnit* permette di realizzare questo processo nel seguente modo:

- le fasi 1 e 4 vengono realizzate tramite i metodi `setUp` e `tearDown`.
- le fasi 2 e 3 vengono realizzate tramite i metodi di asserzione.

Il codice per realizzare i test di unità con *JUnit* è codice *Java*, che viene scritto, compilato, ed eseguito nello stesso modo di qualsiasi classe *Java*.

Di seguito vengono descritti i principali metodi che permettono di realizzare i test di unità in *JUnit*. Per motivi di compatibilità, per realizzare gli esempi, è stata utilizzata la versione 3.8 della libreria *JUnit* che può essere reperita ed installata come descritto nel sito `www.junit.org`.

Tipi di asserzioni

JUnit fornisce dei metodi che permettono di verificare se un'unità sotto test si comporta nella maniera desiderata. Questi metodi sono chiamati asserzioni (*asserts*) e permettono di verificare che il valore ritornato da un'unità sia uguale al valore aspettato. Le asserzioni permettono di segnalare:

Il fallimento di un test (*failtures*): quando il valore atteso è diverso dal valore ritornato dall'unità che si sta testando

La presenza di un errore (*errors*): quando viene lanciata un'eccezione non gestita all'interno di un metodo di test

Sia che venga segnalato un'errore o un fallimento all'interno di un test questo viene terminato mentre gli atri test, all'interno della *suite*, continuano la loro esecuzione. Di seguito vengono descritti i principali tipi di asserzioni che vengono forniti da *JUnit*.

assertEquals

```
assertEquals([messaggio], valore atteso, valore prodotto)
```

È l'asserzione più utilizzata, permette di verificare se il valore atteso è uguale al valore prodotto dall'unità. Il valore atteso è il valore che ci si aspetta venga ritornato dal metodo a cui si vuole verificare il comportamento. Il valore prodotto è il valore che viene ritornato dal metodo dell'unità che si sta verificando. Il messaggio è un parametro opzionale che viene ritornato in *output* quando il test fallisce. Questa asserzione verifica che il valore atteso e il valore prodotto, se confrontati tramite il metodo `equals` dell'oggetto, risultino uguali.

In alcuni casi bisogna prestare attenzione ad utilizzare questa asserzione per confrontare l'uguaglianza di determinati oggetti. Per esempio se si confrontano due `array` tramite `asserEquals`, verrà utilizzato il metodo `equals` per il tipo nativo `array`, che permette di verificare che i due `array` abbiano lo stesso riferimento. In questo caso `assertEquals` non verifica l'uguaglianza dei valori all'interno dell'`array`.

A volte capita di dover confrontare valori decimali (`floats` o `dubles` in *Java*). Per questi tipi è possibile specificare un parametro aggiuntivo al metodo `assertEquals`, che permette di specificare la tolleranza. In questo modo si può verificare se due valori sono uguali rispetto ad una tolleranza desiderata.

[4] `http://www.martinfowler.com/bliki/Xunit.html`

```
assertEquals([messaggio], valore atteso, valore prodotto, tolleranza)
```

Per esempio se si deve confrontare che 3,14 è uguale a `java.lang.Math.PI` con una tolleranza di 0,01 si può realizzare il seguente test:

```
1     public void testPiGreco() {
          assertEquals("Should be 3,14", java.lang.Math.PI, 3.14, 0.01);
3     }
```

assertNull

```
assertNull([messaggio], java.lang.Object object)
assertNotNull([messaggio], java.lang.Object object)
```

Permette di verificare se un oggetto è nullo (o non nullo). Fallisce altrimenti. Il messaggio è opzionale.

assertSame

```
assertSame([messaggio], valore atteso, valore prodotto)
assertNotSame([messaggio], valore atteso, valore prodotto)
```

Permette di verificare che il valore atteso e il valore prodotto dall'unità si riferiscono (o non si riferiscono) allo stesso oggetto. Fallisce altrimenti. Il messaggio è opzionale.

assertTrue e assertFalse

```
assertTrue([messaggio], condizione booleana)
assertFalse([messaggio], condizione booleana)
```

Permette di verificare se una condizione booleana è vera o falsa. Fallisce altrimenti. Questo tipo di asserzione con valore impostato a vero

```
assertTrue(True)
```

viene spesso usata per forzare situazioni di errore e verificare che un metodo lanci un eccezione (vedi esempio "Test di un'eccezione").

fail

```
fail([messaggio])
```

Permette di far fallire il test immediatamente e ritornare il messaggio opzionale. È spesso utilizzata per verificare che un test termini la sua esecuzione in situazioni d'errore (può essere inserita dopo un metodo che avrebbe dovuto lanciare un'eccezione)

Listing 2.12: Test di un'eccezzione

```
1     public void testException() {
          try{
3             Object myObject = null;
              myObject.toString();
5             fail("Should have thrown an exception");
          } catch (NullPointerException e) {
7          assertTrue(true);
          }
9     }
```

In questo test viene verificato che venga lanciata un eccezione durante l'invocazione del metodo `toString` su un oggetto nullo, se questo non avviene il metodo fallirà riportando il messaggio specificato nel metodo `fail`, altrimenti il test verrà superato.

SetUp e Tear-down

Ogni test di unità deve eseguire in modo indipendente dagli altri test. In questo modo i test daranno sempre lo stesso risultato, indipendentemente dall'ordine in cui vengono eseguiti. A volte, prima e dopo l'esecuzione di un test, sono necessarie alcune operazioni (p.es. la creazione e l'eliminazione di dati in un database, la creazione di alcune variabili, etc.). Per effettuare queste operazioni la classe `junit.framework.TestCase` fornisce due metodi:

protected void setUp(): questo metodo viene chiamato prima dell'esecuzione di ogni metodo di test presente in una classe.

protected void tearDown(): questo metodo viene chiamato dopo l'esecuzione di ogni metodo di test presente in una classe.

Ridefinendo, all'interno di una classe di test che deriva da `TestCase`, il metodo `setUp` o `tearDown` e inserendo al suo interno del codice, questo verrà eseguito prima o dopo l'esecuzione di ogni test definito nella classe. Nella sezione 4.1 vengono forniti alcuni esempi di utilizzo di questi metodi.

Per maggiori informazioni sulla libreria *JUnit* è consigliato consultare la guida presente all'indirizzo: `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`.

2.3.2 *Fitnesse*

In questa sezione viene descritto *Fitnesse*[5]. Questo strumento viene utilizzato nell'ultimo capitolo della guida per realizzare alcuni test funzionali di un progetto. *Fitnesse* è uno strumento che permette la realizzazione dei test funzionali di un progetto offrendo la possibilità ai clienti, ai *tester* e agli sviluppatori di verificare se i requisiti funzionali di un progetto sono stati realizzati correttamente. Questo strumento può essere definito come:

1. **Un strumento per verificare le funzionalità di un programma:** È un *framework* che permette di specificare i valori in *input* di una funzionalità e confrontare se i valori di *output* prodotti sono quelli attesi;
2. **Una *wiki*:** possiede delle funzionalità che permettono di creare velocemente pagine *Web* e specificare tramite tabelle i dati di *input* e i dati di *output* attesi che verranno utilizzati durante la fase di test;
3. **Un'applicazione *Web*:** consiste in un'applicazione *Web* che può essere pubblicata. In questo modo gli sviluppatori, i *tester* e i clienti potranno accedere a questa applicazione per consultare, creare e modificare i test funzionali di un programma;

Come viene descritto dall'immagine 2.3 lo sviluppatore realizza il progetto e i test di unità. Quando lo

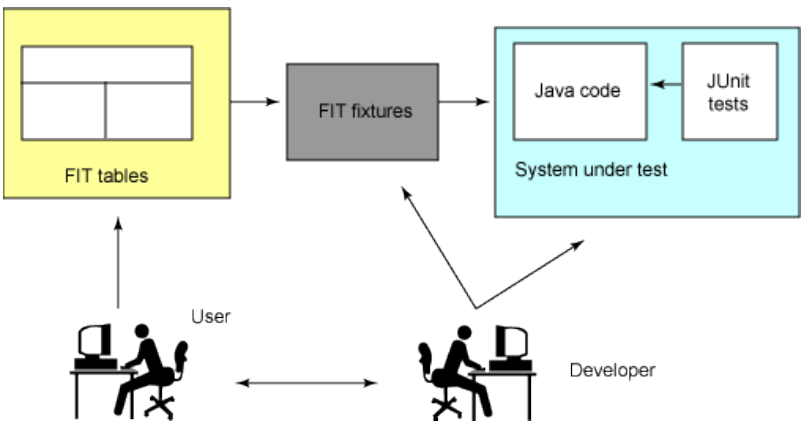

Figura 2.3: Processo di creazione di test con *Fitnesse*

[5] `http://www.fitnesse.org`

sviluppatore ha terminato di realizzare una funzionalità del progetto, e vuole verificarla, deve definire con i *tester* e con il cliente il formato con cui verranno specificati i valori di *input* e i valori attesi di *output* che verranno utilizzati nei test. Solitamente i valori vengono specificati in tabelle.

Concordato con il cliente il formato dei dati di test, lo sviluppatore creerà il codice (chiamato *fixture*) necessario per verificare il comportamento della funzionalità realizzata. Questo codice permette di verificare il comportamento delle funzionalità del programma utilizzando i dati di *input* e confrontando i dati di *output* specificati in tabelle. Il codice realizzato, per funzionare attraverso *Fitnesse*, deve essere realizzato seguendo le specifiche fornite da *Fit*[6] e *Fitnesse*.

Realizzato il codice, lo sviluppatore creerà una suite di test di esempio, dove i valori saranno specificati nella forma concordata con il cliente e con i *tester*. A questo punto, grazie alle funzionalità offerte da *Fitnesse*, sarà possibile pubblicare un'applicazione *Web* che potrà essere utilizzata dal cliente e dai *tester*. I test esposti potranno essere consultati e modificati dagli sviluppatori, dai clienti e dai *tester* attraverso una *wiki*. I dati di test potranno essere eseguiti direttamente dall'applicazione *Web* dove sarà possibile consultare anche i risultati prodotti.

Al termine della sezione di test verranno prodotti alcuni risultati che potranno evidenziare delle problematiche di alcune funzionalità del programma. I test verranno consultati ed eseguiti anche dagli sviluppatori per identificare e correggere i difetti del programma.

Utilizzando questo strumento avviene una collaborazione tra sviluppatori, *tester* e clienti che aumenta le possibilità di realizzare il prodotto in modo da soddisfare nel modo migliore possibile i requisiti funzionali concordati.

Come specificare i dati dei test

I dati dei test, che possono essere utilizzati attraverso *Fitnesse*, posso essere espressi in diverse forme attraverso delle tabelle. I principali tipi di tabelle che possono essere utilizzate sono:

eg.Division		
numerator	denominator	quotient()
1000	10	100.0000
-1000	10	-100.0000
1000	7	142.85715
1000	.00001	100000000
4195835	3145729	1.3338196

```
public class Division extends ColumnFixture {
  public float numerator;
  public float denominator;
  public float quotient() {
    return numerator / denominator;
  }
}
```

fit.ActionFixture		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	37

```
import fit.*;

public class Browser extends Fixture {

    public void library (String path) throws Exception {
        MusicLibrary.load(path);
    }

    public int totalSongs() {
        return MusicLibrary.library.length;
    }

    ....
```

Figura 2.4: Esempio di ColumFixture e ActionFixture

- ColumFixture: Permette di creare delle tabelle per specificare i dati delle funzionalità che possono essere eseguite con pochi valori di *input* e che producono pochi valori di *output*. Nell'intestazione della tabella sono presenti il nome delle variabili di *input* e il nome dei metodi da eseguire. Ogni riga rappresenta un'istanza di un test. Ogni cella specifica il valore di *input* della variabile e il valore di *output* che produrrà il metodo;

- RowFixture: Permette di creare delle tabelle per specificare i dati delle funzionalità che producono molti valori in *output*. Come ad esempio l'esecuzione di una interrogazione in un database;

[6] http://fit.c2.com/

- ActionFixture: Permette di eseguire e controllare il valore di una serie di metodi di una classe che vengono eseguiti in un ordine. Ogni tabella inizia con una cella `start` che permette di inizializzare un'istanza di una classe. Successivamente è possibile eseguire e controllare l'esecuzione dei metodi, invocati dall'istanza della classe creata, con tre diverse modalità:
  - `enter`: permette di invocare un metodo che dei parametri in *input*;
  - `press`: permette di invocare un metodo che non ha parametri in *input*;
  - `check`: permette di controllare il risultato prodotto da un metodo che non ha parametri;
- DoFixture: Ogni riga della tabella permette di eseguire un metodo di una classe;

Definito il formato dei test lo sviluppatore produrrà del codice che permetterà di utilizzare i dati espressi nelle tabelle per eseguire i test funzionali.

2.3.3 Gestione della conoscenza di un progetto

Mentre si programma si crea, si utilizza e si manipola conoscenza. Il miglior modo per esprimere, mantenere e rendere la conoscenza consistente nel tempo è quello di esprimerla attraverso il *plain text* utilizzando linguaggi di *mark-up*. Utilizzando questi strumenti la conoscenza può essere facilmente trasformata in diverse forme, tramite la modifica manuale o la trasformazione attraverso strumenti in modo automatico.

Che cosa è il *plain text*

Il *plain text* è testo (formato da caratteri) espresso in una forma che può essere direttamente compresa e letta dalle persone. Per esempio se si incontra un dato espresso nel seguente modo:

```
lbl = 8a1b
```

Il lettore non riuscirà ad interpretare il significato di `8a1b`. È quindi consigliato esprimere la conoscenza in modo che sia leggibile e interpretabile dalle persone.

```
name = Mario
surname = Rossi
```

In questo esempio il lettore riesce ad interpretare che il dato riguarda il nome e il cognome di una persona. Questi dati potrebbe essere espressi tramite un linguaggio di *mark-up* nel seguente modo

```
<name>Mario</name>
<surname>Rossi</surname>
```

Il *plain text* ha un livello di astrazione più alto rispetto ad una rappresentazione binaria. La rappresentazione binaria infatti non può essere interpretata dalle persone, e i dati rappresentati vengono separati dal contesto (non contengono nessuna informazione della semantica). Tramite il *plain text*, è quindi possibile salvare sia i dati che la semantica. In questo modo i dati sono indipendenti dall'applicazione con cui sono stati creati, e possono essere interpretati e modificati da molti strumenti.

Svantaggi del *plain text*

Utilizzare *plain text* per esprime i dati ha due svantaggi:

1. I dati occupano maggiore spazio rispetto ad una forma binaria. Oltre al dato viene salvata la sua informazione semantica
2. l'interpretazione da parte di un'elaboratore può essere più lenta rispetto ad una forma binaria

Vantaggi del *plain text*

Utilizzare dati espressi tramite *plain text* ha molti vantaggi tra cui:

1. Persistenza
2. Portabilità
3. Benefici nella fase di test e analisi

Di seguito vengono descritti in dettaglio questi vantaggi:

Persistenza

I dati possono sempre essere interpretati e modificati direttamente dallo sviluppatore utilizzando anche strumenti diversi da quello utilizzato per creare il dato. In questo modo i dati sono indipendenti dall'applicazione e finché la conoscenza non cambia, è possibile usarla e modificarla per un periodo potenzialmente più lungo dell'applicazione con cui è stato creato il dato. Attraverso l'utilizzo di trasformatori è possibile filtrare alcune informazioni conoscendo solo la semantica del dato.

Portabilità

Molte applicazione possono utilizzare *plain text*. Tramite qualsiasi *editor* (di qualsiasi sistema operativo) è possibile modificare il contenuto dei dati salvati in questo formato. I dati possono essere salvati in un sistema di versionamento e i cambiamenti possono essere versionati nel tempo. In questo modo è più semplice mantenere e identificare gli errori o i cambiamenti (p.es. tramite l'utilizzo di strumenti come *diff* è possibile identificare i cambiamenti tra le versioni del dato).

Utilizzando *script* e trasformatori, è possibile modificare ed esprimere i dati in varie forme e formati. In questo modo è più difficile infrangere il principio *DRY*.

Benefici nella fase di test e analisi

Se i dati dei test vengono espressi tramite *plain text* è possibile modificare, e gestire i dati in modo semplice. In questo modo, i dati di test possono essere espresse da persone che non conoscono il linguaggio di programmazione con cui sono espressi i test.

Allo stesso modo, se i risultati dei test e di analisi sono espressi tramite *plain text* è molto semplice esprimerli in varie forme e analizzarli. In questo modo è più semplice effettuare sintesi ed elaborazioni dei risultati prodotti.

Come esprimere il *plain text*

Il miglior modo per esprimere conoscenza in forma di *plain text* è tramite l'utilizzo di un linguaggio di *mark-up*. Il linguaggio di *mark-up* è testo che viene aggiunto ad un documento per inserire informazioni riguardanti la sua semantica.

Ogni parte di un documento, che ha lo stesso significato, è racchiusa attraverso *tag* di tipo elemento. Ogni *tag* di tipo elemento può a sua volta contenere degli altri *tag* di tipo elemento. Gli elementi possono avere degli attributi che permettono di descrivere delle informazioni che caratterizzano un elemento. Per rappresentare graficamente un documento espresso utilizzando un linguaggio di *mark-up* viene utilizzata la struttura ad albero.

Il linguaggio di *mark-up* più conosciuto ed utilizzato è *XML*. Di seguito viene fornito un esempio dove vengono rappresentati i dati di un catalogo di cd utilizzando il linguaggio *XML*:

```
<CATALOG>
  <CD>
    <TITLE>Eros</TITLE>
    <ARTIST>Eros Ramazzotti</ARTIST>
    <COUNTRY>EU</COUNTRY>
    <COMPANY>BMG</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1997</YEAR>
  </CD>
```

```
10   <CD>
       <TITLE>Romanza</TITLE>
12     <ARTIST>Andrea Bocelli</ARTIST>
       <COUNTRY>EU</COUNTRY>
14     <COMPANY>Polydor</COMPANY>
       <PRICE>10.80</PRICE>
16     <YEAR>1996</YEAR>
   </CD>
18 </CATALOG>
```

In questo esempio l'elemento `CATALOG` ha al suo interno più elementi `CD`. L'elemento `CD` ha al suo interno più elementi di diversi tipi. I dati espressi in questo modo possono essere letti ed interpretati e modificati in modo semplice.

È possibile descrivere la grammatica di un documento, espresso attraverso *XML*. In questo modo è possibile verificare che un documento rispetti una grammatica. Le grammatiche possono essere espresse attraverso vari linguaggi (*DTD*, *XML schema*). Di seguito viene fornita una possibile grammatica del documento precedente:

```
   <!DOCTYPE CATALOG [
2    <!ELEMENT CATALOG (CD*)>
     <!ELEMENT CD      (TITLE,ARTIST,COUNTRY,COMPANY,PRICE,YEAR)>
4    <!ELEMENT TITLE   (#PCDATA)>
     <!ELEMENT ARTIST  (#PCDATA)>
6    <!ELEMENT COUNTRY (#PCDATA)>
     <!ELEMENT COMPANY (#PCDATA)>
8    <!ELEMENT PRICE   (#PCDATA)>
     <!ELEMENT YEAR    (#PCDATA)>
10 ]>
```

Per maggiori informazioni su *XML* è consigliato consultare `http://www.w3.org/XML/`.

2.4 Ambiente di *build*

L'ambiente di *build* è una delle componenti principali del processo illustrato in figura 2.1, dove avviene la costruzione e la verifica del prodotto. Questa componente deriva dall'adozione della pratica di *Continuous integration* e consiste in una macchina di tipo *server* dove viene installato un'applicazione che permette di effettuare l'attività di integrazione continua del prodotto.

Questa applicazione controlla periodicamente se sono avvenuti cambiamenti al codice del progetto, contenuto nel sistema di controllo del codice sorgente (figura 2.1 B). Se vengono trovati dei cambiamenti, viene scaricata la versione corrente dei sorgenti del progetto nell'ambiente di *build* e viene effettuato il processo di costruzione e verifica del prodotto. Questo processo (rappresentato nella figura 2.1 da *BUILD LOOP*) permette di costruire e verificare la versione corrente del prodotto. In questo processo possono essere eseguiti degli strumenti che permettono di effettuare delle analisi statiche del codice sorgente e produrre delle misure di qualità del prodotto realizzato.

Al termine del processo vengono pubblicati i risultati in modo che possano essere consultati dagli sviluppatori dell'ambiente di lavoro (figura 2.1 C).

Se il processo di costruzione e verifica del prodotto fallisce, l'attività di sviluppo viene interrotta fino a che non viene risolto il problema che ha fatto fallire il processo.

Aggiungendo questo componente al processo di sviluppo, ad ogni iterazione della fase di codifica del processo (figura 2.2 b *Design and Coding*) viene costruito il prodotto e viene verificato che ogni modifica effettuata dagli sviluppatori non introduca degli errori al prodotto.

2.4.1 Strumenti per automatizzare la gestione di un progetto

In questa sezione vengono introdotti *Ant* e *Maven*, due strumenti che permettono di realizzare il processo di costruzione e verifica di un prodotto, sviluppato in linguaggio *Java*, in modo automatico.

Nel capitolo 5 vengono forniti alcuni esempi di utilizzo di questi strumenti per gestire alcuni aspetti di un progetto sviluppato in linguaggio *Java*.

Ant

Ant è uno strumento *open source* che permette di realizzare il processo di costruzione e verifica del prodotto (*build*) principalmente per progetti scritti in linguaggio *Java*. Gli *script*, eseguibili da *Ant*, che rendono automatico il processo di *build*, hanno le seguenti caratteristiche:

- Sono portabili: Permette di risolvere qualsiasi dipendenza del sistema operativo. Se viene eseguito lo *script* che realizza il processo di *build* in diversi sistemi operativi (p.es. *Windows* o *Unix*) questo eseguirà allo stesso modo. *Ant* tradurrà i comandi espressi nello *script* nei comandi specifici del sistema operativo dove si sta eseguendo il processo di *build*.
- Permettono di esprimere strutture con dipendenze: Permette di specificare delle attività che dipendono da altre attività. Per esempio prima di eseguire il processo di *build* è desiderabile eliminare gli *output* prodotti dal processo precedente e ricreare le *directory* di *output*.
- Contengono molte funzionalità indipendenti dall'ambiente di esecuzione: *Ant* definisce un insieme di attività che possono essere inserite, in modo semplice, negli *script*. Per esempio definisce una attività per specificare ed eseguire i test di unità. È in oltre possibile sviluppare attività personalizzate che possono essere eseguite da *Ant*.
- Sono auto descrittivi: Se viene consultato il contenuto di un file, che può essere eseguito da *Ant*, è possibile capire quali sono le attività che compongono il processo di *build*.

Nel manuale, reperibile all'indirizzo `http://ant.apache.org/manual/index.html`, questo strumento viene presentato come:

> *Apache Ant is a Java-based build tool. In theory, it is kind of like make, without make's wrinkles.*

Gli *script*, che vengono interpretati da *Ant* sono realizzati utilizzando il linguaggio *XML* e consistono in una serie di attività (definite dagli elementi `target`) che possono essere eseguite secondo un'ordine prestabilito dallo sviluppatore.

Tutte le attività per gestire un progetto, vengono realizzate dallo sviluppatore utilizzando dei comandi e degli elementi forniti da questo strumento. In questo modo l'attività di creazione e verifica del prodotto non risulta essere standard per ogni progetto, infatti lo sviluppatore può realizzare attività diverse che vengono eseguite con un ordine differente per ogni progetto.

Maven

Maven è un *framework open soruce* per la gestione di progetti *software* in termini di compilazione del codice, distribuzione, documentazione e collaborazione del *team* di sviluppo. Promuove la comprensione e la produttività del *team* coinvolto nello sviluppo, fornendo degli standard che favoriscono l'utilizzo di *best pratice*. Definisce un modello e un linguaggio globale standard che può essere applicato a tutti i progetti *software*. I progetti e i sistemi che usano questo approccio tendono a diventare più trasparenti, riusabili, mantenibili e facili da comprendere.

Ogni progetto gestito con *Maven* possiede un "*Project Object Model*" (*POM*) che definisce, in modo dichiarativo, tutte le caratteristiche del progetto e della sua gestione.

Differentemente da *Ant*, le fasi di un progetto vengono espresse in modo dichiarativo, mediante configurazioni di *plugin* che effettuano l'attività desiderata. In questo modo gli sviluppatori possono definire, configurare ed effettuare le attività tipiche di un progetto *software* (p.es il processo di *build*) senza comprendere la struttura interna di ogni singolo *plugin* che realizza l'attività.

I progetti in *Maven*, adottano una struttura standard:

- Per la struttura delle *directory* di *input* (sorgenti e risorse)
- Per la struttura delle *directory* di *output* (compilati)
- Per la struttura della documentazione prodotta
- Per la gestione delle dipendenze tra progetti
- Per le attività del processo di *build*

I principi adottati da *Maven*

Maven crea una struttura per la gestione di progetti che permette di gestire il progetto ad un livello di astrazione più alto rispetto ad *Ant*.

Di seguito vengono descritti i principi adottati da *Maven* per la gestione di progetti.

Convenzioni sulla configurazione

Uno degli aspetti principali di *Maven* è quello di adottare delle convenzioni standard per la gestione dei progetti. Questo non significa che è impossibile ridefinire queste convenzioni, ma l'utilizzo di queste è consigliato e la ridefinizione è consigliata solo se strettamente necessario.

Utilizzando delle convenzioni lo sviluppatore non deve preoccuparsi di ridefinire, per ogni progetto, tutte le attività comuni (p.es il processo di *build*, la generazione della documentazione, la messa in opera etc.) ma può utilizzare le attività già definite da *Maven*.

Esistono tre principali convenzioni che vengono adottate da *Maven*:

- **Struttura delle *directory* standard**: viene definita una struttura delle *directory* standard per il codice sorgente, le risorse del poggetto e dei test, i file di configurazione, la generazione dei file di *output* e di documentazione. Così facendo, una volta che uno sviluppatore impara questa struttura, riuscirà ad acquisire familiarità, in minor tempo, con qualsiasi progetto gestito da *Maven*. La struttura delle *directory* può essere ridefinita, al costo di aumentare la configurazione e la complessità del *POM*.
- **Un prodotto principale per progetto**: Ogni progetto deve produrre solo un prodotto. Se per caso un progetto è un applicazione *client-server*, è consigliato dividere il progetto in tre sotto progetti separati (uno per le parti in comune, uno per il *client* e uno per il *server*). In questo modo si spinge lo sviluppatore a separare le parti di un progetto. Così facendo diminuirà la complessità di realizzazione e di gestione del progetto e aumenterà il riuso dei sottoprogetti.
- **Nomi standard**: Le convenzioni adottate da *Maven* portano gli sviluppatori ad utilizzare nomi standard. Se vengono utilizzati nomi standard per definire il prodotto, per i nomi delle *directory* e per tutte le parti che compongono il progetto, la comunicazione tra gli sviluppatori risulterà più chiara.

Riuso di logiche di *build*

Viene promosso il riuso delle componenti di un progetto. Tutte le componenti che compongono *Maven* vengono salvate in *plugin*, che possono essere utilizzati ed eseguiti da *Maven*. Esiste un *plugin* per la compilazione del codice sorgente, uno per l'esecuzione dei test di unità, uno per la creazione dei pacchetti (*JAR*, *WAR*, *EAR*, etc.), uno per la creazione della documentazione *Java* e molti altri.

Come si può intuire ogni *plugin* in *Maven* ha un compito ben definito e può essere utilizzato in più progetti. Ogni risultato prodotto da *Maven* deriva dall'esecuzione di uno o più *plugin*. L'esecuzione dei *plugin* è coordinata dal ciclo del processo di *build*, e la configurazione viene definita nell'oggetto di modellazione del progetto. In questo modo, eseguendo gli stessi comandi in progetti diversi, vengono eseguiti gli stessi *plugin*, che eseguono in base alle caratteristiche configurate nel file `pom.xml`(che rappresenta l'oggetto di modellazione del progetto).

Esecuzione dichiarativa

Ogni aspetto di un progetto viene configurato, in modo dichiarativo, attraverso il *POM* rappresentato dal file `pom.xml`. Questo file permette di descrivere ed esprimere le caratteristiche di un progetto. Tramite la lettura di questo file, *Maven* riesce a comprendere le caratteristiche di un progetto ed eseguire le fasi del processo di *build* nella maniera desiderata.

Ogni *POM* deriva dal *Super POM* e quindi ne eredita tutte le configurazioni e le caratteristiche. Il *Super POM* contiene tutte le caratteristiche standard di esecuzione di un progetto ed è uguale in ogni installazione *Maven*. In esso sono contenute le realizzazioni delle convenzioni standard adottate da *Maven*. Tramite questa derivazione non è necessario ripetere in ogni *POM* le informazioni contenute nel *Super POM*, ma è necessario solo definire le configurazioni essenziali che caratterizzano il progetto (o le ridefinizioni delle informazioni contenute nel *Super POM*).

Organizzazione coerente delle dipendenze

Le dipendenze nei progetti, gestiti tramite *Maven*, vengono definite nel *POM*. Se viene creato un progetto come descritto nella sezione 5.4.1 si otterrà il seguente *POM*:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>it.unipd.app</groupId>
  <artifactId>bowlingscore</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>bowlingscore</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

In questo *POM* si può comprendere che il progetto BowlingScore dipende da *JUnit* (righe 10-15). La domanda che può sorgere spontanea è: Come *Maven* reperisce le dipendenze?

In *Maven* ogni dipendenza fa riferimento ad un prodotto (*artifact*). Il tipo di prodotto di un progetto gestito da *Maven* viene specificato nel *POM* dall'elemento `packaging`. In progetti sviluppati in *Java*, tipicamente si utilizza il tipo di archivio *JAR*, ma esistono anche prodotti di tipo *WAR*, *SAR*, *EAR*.

Maven, utilizzando le informazioni contenute nell'elemento `dependency`, riesce a reperire le dipendenze da un *repository*(locale o remoto). Le informazioni specificate all'interno dell'elemento `dependency`, che servono a reperire una dipendenza (presenti nelle righe 11-13) sono :

- **`groupid`**: contiene il nome univoco dell'organizzazione che crea il progetto (p.es. `junit`, `org.apache.maven`).
- **`artifactid`**: contiene il nome univoco del prodotto creato dall'organizzazione rappresentata dall'elemento groupid
- **`version`**: contiene la versione del prodotto

Queste tre informazioni premettono di identificare univocamente una dipendenza. Le dipendenze sono espresse in modo dichiarativo, in questo modo lo sviluppatore non deve preoccuparsi in che modo *Maven* reperisce le dipendenze ma deve solo preoccuparsi di specificarle nel modo corretto. Per capire come specificare e reperire le informazioni di una dipendenza nel *POM* è consigliato consultare il sito: `http://mvnrepository.com`

Quando *Maven* cerca di risolvere una dipendenza effettua le seguenti operazioni:

- Analizza gli elementi `dependency` contenuti nel elemento `dependencies` del *POM*
- Cerca di reperire la dipendenza nel *repository* locale
- Se la dipendenza non è presente nel *repository* locale, vengono consultati dei *repository* remoti
- Se la dipendenza viene trovata viene copiata nella *repository* locale

Come si può capire *Maven*, per reperire le dipendenze, interagisce con due tipi di *repository*:

- ***repository* locale**: Creata durante l'installazione di *Maven*, nella *directory* `~/.m2/repository` e popolata dalle dipendenze richieste da tutti i progetti gestiti da *Maven*.
- ***repository* remote**: Queste *repository* possono venir specificate nel *POM*. Se nessuna *repository* remota viene specificata nel *POM*, *Maven* cerca di reperire le dipendenza da `http://repo1.maven.org/maven2/`.

Le dipendenze vengono salvate nelle *repository* nel seguente modo:
`repository/groupid/artifcatid/version/artifact`
Dove:

- `repository` è l'*URL* del `repository` locale o remoto
- `groupid` è un path relativo dove ogni punto del valore `groupid` viene sostituito con una *slash*
- `artifactid` è il nome della dipendenza
- `version` è il numero della versione della dipendenza
- `artifact` è il prodotto che identifica la dipendenza

Confronto tra *Ant* e *Maven*

Come si può capire dalla lettura delle precedenti sezioni è chiaro che *Ant* e *Maven* hanno caratteristiche e obiettivi differenti e un diverso livello di astrazione.

Ant è uno strumento che permette di facilitare la realizzazione del processo di *build* in modo automatico. Tutte le attività che compongono il processo devono essere stabilite e realizzate dallo sviluppatore ogni volta che deve essere iniziato un nuovo progetto.

Maven è uno strumento che permette di gestire più progetti in modo uniforme. In esso sono definite tutte le attività comuni di un progetto. In questo modo si hanno i seguenti vantaggi rispetto ad utilizzare *Ant*:

- Non c'è la necessità di realizzare processi *ad hoc* per realizzare il processo di *build* dei progetti perché viene fornito uno standard, basato su una serie di *best practice* (adottate da tutti i progetti *apache*)
- Viene resa uniforme la struttura dei progetti
- Vengono ridotte le duplicazioni nella gestione di vari progetti
- Dispone di uno standard per la realizzazione la documentazione e la distribuzione di un progetto

Come si evince dalla figura 2.6 del settembre 2007, tratta da `http://www2.mokabyte.it/cms/article.run?articleId=4AM-5X9-CR4-N4R_7f000001_6539645_f36a858e` *Maven* è uno strumento consigliato per la gestione di molti progetti complessi perché permette di rendere questo processo uniforme. Poiché questo progetto è stato realizzato dopo *Ant* risulta essere meno maturo e non è presente moltissima documentazione.

Product	Version	Description / URL	License	Support	Function-ality	Com-munity	Maturity	ER-Rating	Trend
Ant	1.6.5	Automates compilation of J ava programs. Good replacement for make http://ant.apache.org/	Apache License 2.0	Community	✓✓✓✓	✱✱✱✱	★★★★	◆◆◆	➔
Maven	2.0.4	Software management tool, based on the concept of a project object model (POM). Automates compilation of programs, also manages external program library references http://maven.apache.org/	Apache License	Community	✓✓✓	✱✱✱	★★★	◆◆	➚

Figura 2.5: Valutazione *Open Source Catalogue 2007* di *Ant* e *Maven* tratta da `http://www.optaros.com`

Se si osserva la figura 2.5 tratta da *"Open Source Catalogue 2007"* pubblicato da Optaros[7] la tendenza di crescita di *Maven* è in maggiore rispetto ad *Ant* che risulta essere costante, per questo nei prossimi anni si prevede una ampia diffusione e un miglioramento di questo prodotto.

Come si può intuire dal sito di progetto e dalla figura 2.5, *Ant* ha raggiunto un ottimo livello di maturità, la versione ufficiale dal 2006 al 2008 è la 1.7.0 che risulta essere molto stabile e ben documentata. Per questo è consigliato utilizzare questo strumento per iniziare a gestire un singolo progetto non complesso in modo rapido.

Se per realizzare il processo di costruzione e verifica di un prodotto viene scelto di utilizzare *Maven*, oltre a questa guida è consigliato consultare il libro [16] gratuitamente scaricabile.

Nel capitolo 5 vengono realizzati due processi di costruzione e verifica di un prodotto realizzato utilizzando il linguaggio *Java* utilizzando *Ant* (sezione 5.3) e *Maven* (sezione 5.4).

[7] http://www.optaros.com/

Concetto	Ant	Maven
Tempo necessario per scaricare il prodotto	10 minuti	10 minuti
Installazione	Semplice e immediata	Semplice e immediata
Tempo per iniziare un nuovo progetto (semplice)	5 minuti	20 minuti (5 minuti se si riutilizzano le impostazioni di un precedente progetto)
Tempo per iniziare un nuovo progetto (complesso)	Può richiedere molto tempo soprattutto nel contesto di progetti molto complessi/articolati	30 minuti
Aggiungere nuovi target/goal (semplici)	10 minuti (target)	5 minuti (goal esistente)
Aggiungere nuovi target/goal (complessi)	10 minuti (target)	5 minuti (goal esistenti)
Implementazione di un nuovo target/goal (semplice)	10 minuti (target)	Ore
Implementazione di un nuovo target/goal (complessi)	Ore	Giorni
Tempo di apprendimento	Ore (struttura e funzionamento sono assolutamente immediati)	Giorni (comprendere una differente filosofia di intendere il processi di build, comprendere il meccanismo di funzionamento dei repository, etc.)
Struttura standard del progetto	Non definita: ogni team è libero di organizzare il proprio progetto a piacimento. Flessibilità che si paga con la necessità di dover spiegare a ogni nuovo elemento del team struttura, script, etc.	Predefinita. Tuttavia è possibile apportare diverse variazioni anche importanti
Supporto dei progetti complessi basati su più progetti	Sì, ma richiede appositi script (necessario definire progetto “master”)	Sì, nativamente
Disponibilità di tool/plug-in	Elevata	Elevata
Integrazione con IDE	Elevata (alcuni IDE basano il relativo progetto direttamente su script Ant)	Elevata
Definizione del sito di documentazione	Demandato al team e/o ai vari tool	Nativa
Apertura della struttura e meccanismi interni	Elevata	Ridotta
Documentazione	Abbondante e accurata	Limitata e inconsistente (ma questa lacuna si è ridotta grazie ad articoli e libri di recente pubblicazione)
Bugs/Problemi	Quasi inesistenti	Frequenti, tanto che a volte è necessario o modificarli in prima persona o trovare percorsi alternativi

Figura 2.6: Confronto tra *Ant* e *Maven* tratta da `http://www.mokabyte.it`

2.4.2 Misurazione della qualità del codice del progetto

Per migliorare un progetto deve essere possibile quantificare e analizzare alcuni aspetti riguardanti lo stato della sua qualità. Esistono molti strumenti che permettono di analizzare staticamente il codice sorgente di un progetto e ricavare delle misure di qualità che permettono di:

- Imporre il rispetto di convenzioni e stili di codice
- Verificare la congruità della documentazione

- Controllare alcuni aspetti quantitativi riguardanti la qualità di un progetto come: complessità ciclomatica, grafo delle dipendenze, numerosità delle linee di codice
- Ricerca di duplicazioni del codice
- Ricerca di errori comuni
- Ricerca di indicatori di parti incomplete

Analizzando i dati prodotti da questi strumenti è possibile migliorare alcuni aspetti del progetto, aumentandone la qualità.

Strumento	Descrizione	Note	Riferimento
Javadoc	Permette di generare la documentazione del progetto analizzando il Javadoc presente nel codice sorgente	Disponibile sia per Ant che per *Maven*	`http://ant.apache.org/manual/CoreTasks/javadoc.html` e `http://maven.apache.org/plugins/maven-javadoc-plugin/`
Checkstyle	Permette di verificare il rispetto di convenzioni di stile nel codice del progetto. Permette di rendere il codice di un progetto più leggibile	Disponibile sia per Ant che per *Maven*.	`http://checkstyle.sourceforge.net/` e `http://maven.apache.org/plugins/maven-checkstyle-plugin/`
PMD	Permette di verificare la presenza di errori comuni nel codice sorgente	Disponibile sia per Ant che per *Maven*.	`http://pmd.sourceforge.net/` e `http://maven.apache.org/plugins/maven-pmd-plugin/index.html`
CPD	Fa parte del progetto PMD. Permette di verificare la presenza di duplicazioni di codice sorgente di tipo copia e incolla	Disponibile sia per Ant che per *Maven*	`http://pmd.sourceforge.net/cpd.html` e `http://maven.apache.org/plugins/maven-pmd-plugin/index.html`
FindBugs	Permette di effettuare analisi statiche del codice sorgente *Java* al fine di trovare errori comuni	Disponibile sia per Ant che per *Maven*	`http://findbugs.sourceforge.net/` e `http://mojo.codehaus.org/findbugs-maven-plugin/` e `http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2707.pdf`
Tag List	Permette di elencare tutti i marcatori presenti nel codice sorgente di un progetto (p.es. TODO, @todo, FIXME)	Disponibile per *Maven*	`http://mojo.codehaus.org/taglist-maven-plugin/taglist.html`
JavaNCSS	Permette di calcolare alcune misure quantitative e qualitative di un progetto (p.es. numero di classi, righe di codice)	Disponibile sia per Ant che per *Maven*	`http://www.kclee.de/clemens/java/javancss/` e `http://mojo.codehaus.org/javancss-maven-plugin/`
JDepend	Permette di calcolare misure qualitative sull'ortogonalità di un progetto	Disponibile sia per Ant che per *Maven*	`http://clarkware.com/software/JDepend.html` e `http://mojo.codehaus.org/jdepend-maven-plugin/`

Tabella 2.2: Strumenti di analisi del codice

Nella tabella 2.2 vengono riportati alcuni di questi strumenti che possono essere utilizzati per misurare la qualità del codice sorgente di un progetto *Java*.

Come verrà descritto nel capitolo 3, la fase di test di unità di un progetto è una cosa molto importante perché permette di verificare e assicurare nel tempo il comportamento delle parti che compongono il progetto. Visto che i test di unità devono essere scritti in modo professionale, è buona norma applicare gli stessi controlli, effettuati nel codice di produzione, anche al codice di test. Esistono degli strumenti specifici che permettono di dare ulteriori informazioni sullo stato di qualità dei test. Nella tabella 2.3 vengono riportati degli strumenti specifici che permettono di analizzare alcuni aspetti dei test di unità.

Data la numerosità di questi strumenti, per aumentare la qualità del progetto, è necessario selezionare e applicarne solo alcuni. Se in un progetto vengono prodotte troppe misure o segnalazioni di possibili problemi è

Strumento	Descrizione	Note	Riferimento
Cobertura	È uno strumento *open source* che permette di verificare se i test di unità sono esaustivi. Verifica che tutto il codice di un progetto sia eseguito attraverso i test di unità	Disponibile sia per *Ant* che per *Maven*	`http://cobertura.sourceforge.net/`
EMMA	È uno strumento *open source* per misurare e documentare *"Java Code Coverage"*. Può essere utilizzato per verificare se i test di unità sono esaustivi	Disponibile solo per *Ant*. Versione per *Maven* in fase di sviluppo	`http://emma.sourceforge.net/`
Clover	Permette di verificare se i test di unità sono esaustivi. É uno strumento a pagamento e analizza più aspetti rispetto a *Cobertura*	È disponibile una versione di prova per un periodo di 30 giorni. Disponibile sia per *Ant* che per *Maven*.	`http://www.atlassian.com/software/clover/`
Jester	Permette di verificare che i test di unità falliscono se viene specificato un valore atteso diverso da quello specificato nel test	Disponibile per *Ant* ma non per *Maven*	`http://jester.sourceforge.net/`

Tabella 2.3: Strumenti di analisi dei test

molto probabile che queste vengano ignorate dagli sviluppatori. È quindi consigliato selezionare e configurare solo alcuni strumenti che producano misure di interesse per gli sviluppatori, in questo modo sarà più probabile che la qualità di un progetto aumenti nel tempo.

Come viene riportato nella tabella 2.2 e nella tabella 2.3 quasi tutti gli strumenti possono essere eseguiti automaticamente nel processo di costruzione e verifica di un prodotto sia attraverso *Maven* che attraverso *Ant*. In questo modo questi strumenti verranno eseguiti, ad intervalli predefiniti, dal sistema di integrazione continua che permetterà di produrre e pubblicare delle reportistiche che indicheranno delle misure di qualità del prodotto sviluppato.

2.4.3 Strumenti per realizzare il sistema di integrazione continua

In questa sezione vengono presentati due strumenti che vengono utilizzati in questa guida per realizzare il sistema di integrazione continua (figura 2.1 3) per lo sviluppo di un progetto in linguaggio *Java*.

Cruise Control

Cruise Control è uno *framework open source* per la realizzazione di un sistema di integrazione continua. Possiede delle funzionalità che permettono di monitorare e scaricare i sorgenti di un progetto dal sistema di controllo del codice sorgente.

Questo strumento permette di programmare l'esecuzione del processo di costruzione e verifica di un prodotto sia attraverso *Ant* che attraverso Maven (è più adatto per eseguire processi attraverso *Ant*). Dispone di una applicazione *Web* che permette di pubblicare lo stato del processo di costruzione e verifica di un prodotto scritto in *Java* e notificare tramite vari strumenti lo stato del processo agli sviluppatori.

La configurazione di questo strumento viene effettuata attraverso la creazione di un file di configurazione, scritto in *XML*, dove vengono specificate tutte le caratteristiche dell'esecuzione del processo di *build*.

Un esempio di configurazione ed esecuzione di questo strumento, per realizzare un sistema di integrazione continua che esegue il processo di costruzione e verifica di un prodotto *software* attraverso *Ant*, viene fornito nella sezione 6.1 della guida.

Continuum

Continuum è un applicativo *open source* di tipo *server* per realizzare un sistema di integrazione continua per effettuare il processo di costruzione di progetti *Java*. Permette di programmare l'esecuzione del processo di costruzione e verifica di prodotti sia attraverso *Ant* che attraverso *Maven*. Consiste in una applicazione *Web* dove vengono configurati e specificati i progetti da inserire nel sistema di integrazione. Nell'applicazione *Web* vengono pubblicati i risultati dei processi di costruzione e verifica dei prodotti gestiti, inoltre permette di configurare molti aspetti sulla programmazione dell'esecuzione del processo e la pubblicazione dei risultati.

La configurazione di questo strumento risulta più semplice e intuitiva rispetto a *Cruise Control*, ma non è presente molta documentazione.

Un esempio di configurazione ed esecuzione di questo strumento, per realizzare un sistema di integrazione continua che esegue il processo di costruzione e verifica di un prodotto *software* attraverso *Maven*, viene fornito nella sezione 6.2 della guida.

Capitolo 3

Test di unità

3.1 Definizioni utili

I test di unità sono del codice, prodotto dallo sviluppatore, che esercitano un'unità del programma. Per unità si intende una funzionalità atomica che può essere verificata in modo isolato, in modo da assicurare che il risultato del test non sia influenzato da altre unità.

Nella programmazione ad oggetti un'unità può essere uno o più metodi di una classe, o un'istanza di una classe. Nella programmazione procedurale un'unità corrisponde ad una funzione.

Per creare i test di unità sono necessari i seguenti componenti:

- Un modo standard per impostare l'ambiente di esecuzione del test
- Un modo per selezionare un test o un insieme di test
- Un modo per analizzare i valori aspettati, prodotti dalle unità
- Un modo standard per esprimere se il test è stato superato, se è fallito o se sono stati prodotti degli errori

Tipicamente vengono sviluppati dal programmatore che sviluppa le unità, per verificare l'assenza di alcuni errori, e assicurare il comportamento dell'unità prodotta. Come riportato in [1]

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

Tramite i test di unità è impossibile eliminare tutti gli errori da una funzionalità del progetto, ma permettono che gli errori che vengono trovati non si ripresentino in futuro.

3.2 Proprietà desiderabili

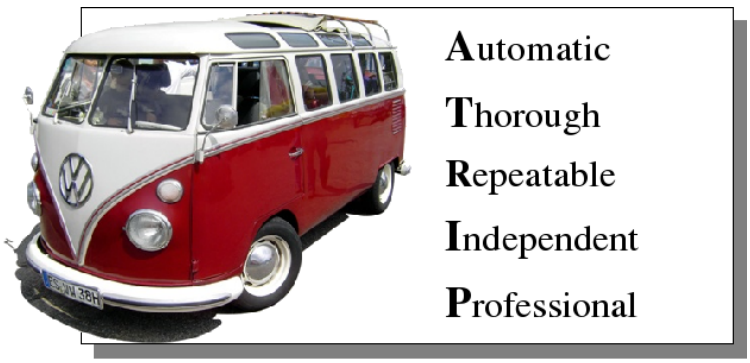

Figura 3.1: *A Trip*: le proprietà dei buoni test di unità

I test di unità sono uno strumento fondamentale per la creazione di un progetto. Permettono di verificare l'assenza di alcuni errori, e il corretto funzionamento di unità che compongono il progetto.

I test di unità devono essere mantenuti nel tempo perciò se non sono semplici e scritti in modo professionale possono far spendere molto tempo prezioso allo sviluppatore. Per questo esistono delle linee guida che descrivono come realizzare dei buoni test di unità e possono essere riassunte nell'acronimo *A-TRIP* (un viaggio) riportato nella figura 3.1. Le linee guida e gli aspetti descritti in questa sezione derivano da [9] e vengono riportate nel capitolo 7 di [10]. Nelle successive sezioni vengono analizzate tutte le proprietà desiderabili dei buoni test di unità.

Automatic

I test di unità devono essere eseguiti automaticamente. In ogni progetto deve essere disponibile un "automazione a comando" che permetta a tutti di invocare e far eseguire tutti o una parte dei test di unità in modo semplice.

Durante la fase di sviluppo del progetto è importante che i test possano eseguire:

- In modo rapido: I test di unità devono essere semplici e la loro esecuzione non deve impiegare più di pochi secondi. Se un test, per eseguire, impiega qualche minuto, deve essere considerato come un problema perché rallenterebbe l'attività lavorativa di tutti gli sviluppatori del progetto.
- Senza richiedere l'interazione umana: Se un test di unità richiede che alcuni parametri siano inseriti, ogni volta, manualmente da uno sviluppatore, questo non permetterebbe di eseguire tutti i test del progetto in modo automatico a determinate ore del giorno.
- In modo autonomo: L'automazione che effettua l'esecuzione dei test di unità deve essere in grado di capire quando e dove i test falliscono ed avvisare gli sviluppatori. In questo modo gli sviluppatori saranno interrotti, dall'attività lavorativa, solo quando uno o più test falliranno. L'automazione dovrà verificare e comunicare agli sviluppatori dove e quali test falliscono. In questo modo gli sviluppatori potranno concentrarsi sullo sviluppo del progetto e ad ogni esecuzione dei test non dovranno controllare tutti i risultati ma solo quelli che segnalano errori.

Nel capitolo 5 vengono spiegati alcuni metodi per far eseguire automaticamente tutti i test di unità, ogni volta che viene effettuata una modifica al codice del progetto.

Thorough

Dei buoni test di unità devono essere esaustivi e accurati, devono verificare il comportamento di qualsiasi parte del progetto che potrebbe creare degli errori. Esistono degli strumenti che permettono di misurare se ogni parte del progetto è stata eseguita durante la fase di test, e possono calcolare:

- La percentuale di righe di codice che vengono esercitate attraverso i test di unità nel progetto
- La percentuale di possibili *branch* che vengono eseguiti dai test di unità
- Il numero di eccezioni che vengono controllate attraverso i test
- Altri dati che permettono di capire dove il progetto è carente di test di unità

Come si può intuire, non è detto che se in un progetto viene eseguito il 100% del codice dai test di unità questo è privo di errori. Come è stato detto in precedenza i test permettono di verificare la presenza di alcuni errori, e non la loro assenza. Questi strumenti permettono di investigare e capire quali parti del progetto non sono eseguite dai test di unità e quindi potrebbero avere un comportamento non atteso. Utilizzando questi strumenti è possibile capire quali parti del codice dovrebbero essere riscritte. Supponiamo che in un progetto vengano segnalati degli errori, e questi errori sono riconducibili sempre alla stessa unità del codice. Supponiamo che l'unità abbia una quantità di test di unità che copre il 100% del codice del progetto. In questi casi, dopo aver riprodotto l'errore tramite il test la cosa più saggia da fare è riscrivere interamente l'unità. L'attività di *refactoring* risulterà semplice, visto che lo sviluppatore avrà compreso meglio quale è il comportamento dell'unità da realizzare, e

i test permetteranno di assicurare che la nuova unità abbia un comportamento identico (e preferibilmente con meno errori) della precedente.

L'installazione e l'esecuzione di un strumento che permette di verificare se i test di unità di un progetto sono esaustivi viene descritta nella sezione 5.4.9.

Repeatable

I test di unità devono produrre sempre lo stesso risultato. Per essere ripetibili, i test di unità devono avere le seguenti caratteristiche:

- Essere indipendenti dall'ordine di esecuzione: L'ordine di esecuzione dei test di unità non deve influenzare il risultato. Per questo è necessario che i test siano indipendenti
- Essere indipendenti dall'ambiente di esecuzione: L'esecuzione dei test non deve dipendere da risorse esterne al progetto o da risorse non gestite dal *team* di sviluppo. Se alcune unità devono utilizzare risorse esterne (p.es. *database*) è consigliato utilizzare la tecnica *Mock Object* per simulare il comportamento di queste componenti. Se è indispensabile che i test utilizzino componenti esterne è consigliato che ogni sviluppatore abbia un'ambiente di esecuzione indipendente (p.es ogni sviluppatore deve avere il suo *database* di test). Le componenti esterne, utilizzate dai test, devono iniziare sempre dallo stesso stato in modo da non compromettere il risultato dei test

Se i test non sono ripetibili, lo sviluppatore potrebbe trovarsi in situazioni dove il test non riesce ad eseguire, e perdere del tempo prezioso per capire le motivazioni che hanno reso il test non ripetibile (p.es. investigare lo stato delle componenti esterne).

Independent

I test di unità devono essere il più possibile indipendenti dall'ambiente di esecuzione e dagli elementi esterni al progetto. Quando si scrive un test è consigliato verificare il comportamento di un singolo aspetto del progetto (in questo modo si riesce ad identificare univocamente un'errore). Questo non significa che un test di un'unità deve avere solo una asserzione, ma deve controllare solo un metodo o più metodi che realizzano un'aspetto di una funzionalità del progetto.

È consigliato avere almeno un test per ogni possibile errore che potrebbe verificarsi nel progetto. In questo modo quando un test fallisce è possibile identificare precisamente quale parte del codice del progetto (quale unità) produce l'errore. Se il test è indipendente il suo comportamento sarà ripetibile nel tempo, perché il suo comportamento non dipenderà dalle altre unità del progetto. La ripetibilità del test è un aspetto che permette di capire se il test è indipendente.

Professional

Poiché i test di unità sono codice, devono essere scritti e mantenuti con la stessa professionalità del codice di produzione del progetto.

I test di unità devono essere scritti rispettando tutti i principi descritti nella prima parte del volume, quindi devono essere ben progettati, non infrangere il principio *DRY*, devono essere ortogonali e devono essere commentati come il codice di produzione.

Visto che i buoni test di unità devono essere esaustivi, è ragionevole che il numero di linee di codice per realizzare i test sia pari o a volte superiore delle linee di codice in produzione. Per questo motivo i test devono essere trattati con la stessa professionalità del codice di produzione, altrimenti si rischia di aumentare l'entropia nei test, e quindi anche l'entropia dell'intero progetto.

Come correggere un'errore

Tipicamente i test di unità vengono creati per assicurare che un componente di un progetto abbia il comportamento atteso. Anche se il codice di un progetto è ampiamente verificato, tutte le unità hanno i propri test, è lo stesso molto probabile che con il tempo vengano travati degli errori (da altri sviluppatori o dagli utenti). Quando si trova un errore in un progetto è importante che:

- L'errore venga corretto
- Durante la correzione non vengano introdotti altri errori
- L'errore non si ripresenti in futuro

Per far ciò è necessario seguire quattro passi:

1. Identificare l'errore (l'unità o le unità affette dall'errore)
2. Scrivere un test che riproduca l'errore (quindi fallisca)
3. Correggere l'unità o le unità in modo da far superare il test
4. Eseguire tutti i test del progetto per verificare di non aver introdotto degli errori durante la correzione

Come si può intuire, con il passare del tempo, il progetto verrà popolato con molti test di unità che permetteranno di verificare che gli errori che vengono trovati non si ripresentino in futuro.

Ogni volta che viene corretto un errore la cosa più giusta da fare è quella di chiedersi se errori simili possono verificarsi in altre unità del progetto, e creare nuovi test che permettano di assicurare che il progetto non abbia determinati tipi di errori.

Verifica dei test

Per verificare se un test è stato scritto in modo corretto è necessario riprodurre nell'unità l'errore che fa fallire il test. In questo modo è possibile verificare che, quando viene trovato l'errore il test fallisca.

Nella pratica *Test-Driven Development* il test viene realizzato prima del codice di produzione. In questo modo è possibile verificare che il test fallisca fino a che il codice di produzione non superi il test. Così facendo viene verificato sia che il codice di test segnali il fallimento, se il codice di produzione ha dei problemi, sia che il test esegua correttamente se il codice di produzione supera il test.

Se non si adotta un approccio *Test-Driven Development*, e si ha il sospetto che alcuni test non sono stati realizzati correttamente, è consigliato riprodurre l'errore nell'unità e verificare che il test fallisca.

3.3 Creazione di test con *JUnit*

Per iniziare a creare dei test di unità per un progetto, sviluppato in *Java*, vengono descritti alcuni esempi di classe, che utilizzano la libreria *Junit*, che permettono di realizzare i test di unità.

```
1  package it.unipd.app;

3  import junit.framework.TestCase;

5  public class FirstTest extends TestCase {

7      public FirstTest(String name) {
           super(name);
9      }

11     public void testMultiplication() {
           assertEquals(64, 8*8);
13     }

15 }
```

Di seguito vengono descritte le righe più significative che compongono il test:

- Riga 1: viene specificato il pacchetto di appartenenza
- Riga 3: viene importata la classe `junit.framework.TestCase`, classe fondamentale per realizzare test utilizzando *JUnit*
- Riga 5: viene specificato il nome della classe che conterrà i test. Questa classe estende `TestCase` che fornisce delle funzionalità che permettono di realizzare i test di unità (tra cui tutti i metodi di asserzioni descritti precedentemente)

- Riga 7: viene creato il costruttore. `TestCase` fornisce un costruttore ad un parametro che riceve in ingresso una stringa (di seguito verrà spiegato il significato del paramento del costruttore).
- Riga 11: viene realizzato il metodo di test. La classe realizzata contiene solo un metodo di test, ma ne potrebbe contenere più di uno. Fino alla versione 4 della libreria *JUnit* i nomi dei metodi di test devono iniziare con la parola `test` (In questo volume, per motivi di compatibilità con *Java 1.4* viene utilizzata la versione 3.8 di *Junit*). Il metodo `testMultiplication` contiene una asserzione che permette di verificare una semplice moltiplicazione. Il metodo avrebbe potuto contenere più di una asserzione

Se viene eseguita una classe di test, *Junit* di default, esegue tutti i metodi di test al suo interno. In alcuni casi non è consigliato eseguire tutti i test contenuti in una classe di test (p.es. nel caso in cui è presente un test che impiega delle ore). Per questo *Junit* permette di definire una *suite* di test dove specificare quali test devono eseguire all'esecuzione di una classe.

```
1  package it.unipd.app;
2
3  import junit.framework.Test;
4  import junit.framework.TestCase;
5  import junit.framework.TestSuite;
6
7  public class SecondTest extends TestCase {
8
9      public SecondTest(String name) {
10         super(name);
11     }
12
13     public void testOnePlusOne() {
14         assertEquals(2, 1 + 1);
15     }
16
17     public void testTwoPlusTwo() {
18         assertEquals(4, 2 + 2);
19     }
20
21     public void testLongRunner() {
22         // same code that take few hours..
23     }
24
25     public static Test suite() {
26         TestSuite suite = new TestSuite();
27         suite.addTest(new SecondTest("testOnePlusOne"));
28         suite.addTest(new SecondTest("testTwoPlusTwo"));
29         return suite;
30     }
31
32 }
```

In questo esempio quando verrà eseguita la classe di test `SecondTest` verranno solo eseguiti i test specificati nella *suite* (`testOnePlusOne`, `testTwoPlusTwo`).

Come si può intuire dalle righe 27-28 dell'esempio, il costruttore di ogni classe che estende `TestCase` permette di specificare una stringa che specifica il nome del metodo di test da eseguire. In una *suite* di test si possono specificare anche test di altre classi.

Supponiamo di voler creare una *suite* di test che esegua tutti i test della classe `FirstTest` e il test `testLongRunner` della classe `SecondTest`.

```
1  package it.unipd.app;
2
3  import junit.framework.Test;
4  import junit.framework.TestCase;
5  import junit.framework.TestSuite;
6
7  public class TestCollection extends TestCase {
8
9      public TestCollection(String testMethodName) {
10         super(testMethodName);
11     }
12
13     static public Test suite() {
14         TestSuite suite = new TestSuite();
15         suite.addTest(new SecondTest("testLongRunner"));
```

```
16          // add all method test in the class
            suite.addTestSuite(FirstTest.class);
18          return suite;
        }
20
}
```

Se viene eseguita la classe `TestCollection` verranno eseguiti:

- Il metodo di test `SecondTest.testLongerRunner()`;
- Tutti i metodi di test presenti nella classe `FirstTest.class`;

Se si vuole inserire alla *suite* solo i metodi specificati nella *suite* della classe `SecondTest` è necessario modificare la *suite* della classe `TestCollection` nel seguente modo:

```
1       static public Test suite() {
            TestSuite suite = new TestSuite();
3           suite.addTestSuite(SecondTest.suite());
            return suite;
5       }
```

3.4 Organizzazione dello spazio di lavoro

In questa sezione viene descritto come deve essere organizzato lo spazio di lavoro per verificare il comportamento di un progetto reale tramite i test di unità.

3.4.1 Dove collocare le procedure di test

In progetti di piccole dimensioni, la locazione dei test non è un problema, ma in progetti di medie e grandi dimensioni è consigliato separare il codice di test dal codice di produzione.

Tipicamente il codice di test non viene distribuito (almeno nei progetti non *open source*), ma deve accedere ai metodi e agli attributi pubblici e protetti delle classi che compongono il progetto.

Per questi motivo è consigliato trovare un modo che permetta di separare il codice di test dal codice di produzione, ma permettere ai metodi di test di accedere agli attributi e ai metodi protetti delle classi del progetto.

Di seguito vengono descritti tre possibili modi dove collocare i test, evidenziando pregi e difetti.

Stessa *directory*

Supponiamo sia necessario verificare i metodi della classe:

`it.unipd.app.App`

tramite la classe di test:

`it.unipd.app.AppTest`

Il modo più semplice per permettere alla classe di test di accedere ai metodi e agli attributi protetti della classe del codice di produzione è quello di adottare la seguente struttura delle *directory*:

```
it/unipd/app/App.java
it/unipd/app/AppTest.java
```

In questo modo i due file sono nella stessa *directory*.

- Vantaggio: La classe di test può accedere ai metodi e agli attributi protetti e pubblici della classe del codice di produzione
- Svantaggio: Non è immediato dividere il codice di test dal codice di produzione per la distribuzione del prodotto

Sotto *directory*

La seconda opzione è quella di creare una *directory* di test:

```
it/unipd/app/App.java
it/unipd/app/test/AppTest.java
```

In questo modo i file di produzioni sono separati dai file di test.

- Vantaggio: Il codice di test è separato dal codice di produzione
- Svantaggio: Il codice di test non può accedere ai metodi e agli attributi protetti della classe del codice di produzione

Stesso *package directory* differenti

La terza opzione, adottata in quasi tutti i progetti *Java*, è quella dividere i file di test dai file di produzione. Per permettere ai metodi delle classi di test di accedere ai metodi e agli attributi delle classi di produzione è necessario replicare nella *directory* di test la stessa struttura dei *package* del codice di produzione, ottenendo il seguente risultato:

```
src/it/unipd/app/App.java
test/it/unipd/app/AppTest.java
```

I progetti che adottano questa struttura delle *directory* hanno le seguenti proprietà:

- Vantaggi:
  - Il codice di test è separato dal codice di produzione
  - Il codice di test può accedere ai metodi e agli attributi protetti della classe in produzione
- Svantaggio: È necessario distinguere il `classpath` per il codice di produzione dal `classpath` del codice di test.
  - Il `classpath` del codice di produzione conterrà le librerie utilizzate dalle classi in produzione
  - Il `classpath` del codice di test conterrà il `classpath` del codice di produzione più le librerie utilizzate dalle classi di test

Come si può intuire la terza opzione è la più conveniente perché permette di rendere ortogonale il codice di produzione dal codice di test. In questo modo sarà semplice dividere il codice di produzioni dal codice di test.

Per progetti di piccole dimensioni anche la prima opzione può essere adottata. la cosa più importante è che dopo aver effettuato una scelta su dove collocare i test è necessario rimanere coerenti con questa scelta.

3.4.2 Frequenza di esecuzione

Quando si lavora in un *team* di sviluppo, il codice prodotto può influenzare il corretto funzionamento delle funzionalità prodotte dagli altri sviluppatori. Per questo per assicurare che non vengano introdotti errori è necessario che ogni unità prodotta abbia i propri test.

Prima di inviare il codice prodotto al sistema di versionamento è buona norma verificare che tutti i test presenti nel progetto vengano sempre superati. In questo modo non si rischierà di rompere il processo di costruzione del progetto.

Quando è necessario inviare il codice al sistema di versionamento è buona norma verificare di non far fallire il processo di costruzione del progetto e quindi verificare di non inviare:

- Codice incompleto (effettuare attività di *commit* solo delle classi ma non delle dipendenze)
- Codice che non compila
- Codice che compila ma non fa compilare altre parti del progetto
- Codice senza test di unità

- Codice con test di unità che falliscono
- Codice che superi i suoi test di unità ma fa fallire altri test di unità presenti nel progetto

È necessario eseguire tutti i test presenti nel progetto quando:

- Si realizza un nuova funzionalità: creare e compilare almeno i test dell'unità realizzata
- Fissaggio di un errore: creare dei test che riproducano l'errore e correggere le unità affette dall'errore
- Ad ogni compilazione di una funzionalità: compilare almeno i test locali per verificare che non si siano introdotti errori
- Ad ogni invio del codice al sistema di versionamento: eseguire tutti i test di unità del progetto

In grandi progetti capita che l'esecuzione di tutti i test del sistema (quindi non solo dei test di unità) possa impiegare anche alcune ore. In questi progetti è necessario adottare un approccio modulare, in modo che l'esecuzione e la compilazione dei test di un modulo venga eseguita continuamente, l'esecuzione di tutti i test dell'intero progetto venga eseguita tramite un'automazione a comando programmata ad eseguire in determinate ore della giornata.

Nel capitolo 5 della guida viene descritto come creare diversi tipi di automazione che permettono di eseguire in modo automatico tutti i test presenti in un progetto. Inoltre viene descritto come creare un sistema di integrazione continua che permette di eseguire automaticamente tutti i test del progetto ad ogni attività di *committing*. In questo modo sarà possibile creare un sistema di non regressione che permette di verificare che gli errori trovati in passato non si ripresentino nelle nuove versioni del progetto che si sta sviluppando.

Capitolo 4

Produrre codice verificabile

È molto difficile capire se una unità veramente funziona. Risulta sopratutto difficile riprodurre le condizioni che permettono di ricreare e trovare un errore nelle unità sviluppate. Ovviamente trovare tutti gli errori presenti in unità è una cosa impossibile.

Per sviluppare delle buone unità, è consigliato seguire delle linee guida che permettono di ricordare quali sono le aree più importanti da verificare. Queste linee guida possono essere utilizzate anche per la creazione di unità utilizzando un approccio *Test-Driven Development*. Come viene riportato in [10], esistono sei aree che devono essere controllate in ogni unità. Queste aree possono essere riassunte con l'acronimo *Right BICEP* (il bicipite destro).

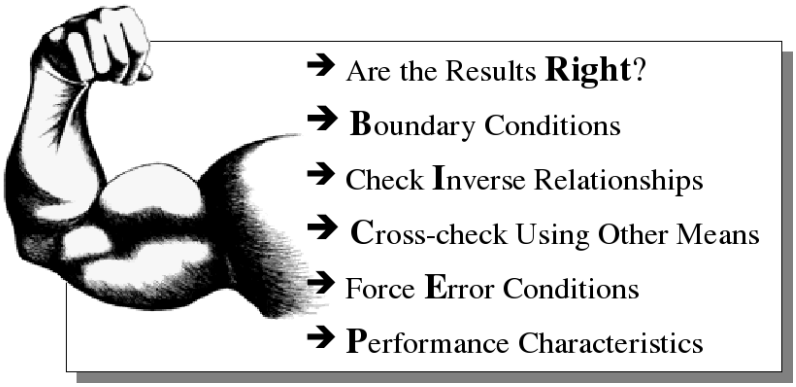

Figura 4.1: *The Right BICEP*

Nelle prossime sezioni vengono descritte in dettaglio ogni aree e vengono riportati alcuni esempi di realizzazione dell'unità centrale del progetto BowlingScore, utilizzando l'approccio *Test-Driven Development*.

4.1 Controllare se i risultati sono corretti

La prima cosa da fare quando si crea un'unità è quella di verificare se i risultati che essa produce sono corretti. Per corretto si intende che il risultato atteso sia uguale al risultato prodotto dall'unità.

Per effettuare questa attività devono essere capiti i requisiti del sistema e quindi sapere quali sono i risultati prodotti dall'unità quando esegue correttamente. Molto spesso però i requisiti del sistema non sono ben definiti e a volte non è molto chiaro come deve essere il comportamento di tutte le parti di un progetto. In questi casi è consigliato consultare l'utente o gli altri sviluppatori per comprendere al meglio i requisiti (e non programmare per coincidenza).

A volte però capita che i requisiti non possono essere definiti a priori, o possono cambiare nel tempo. In questi casi i test di unità sono un buon punto di partenza per documentare nel codice, come uno sviluppatore ha interpretato i requisiti e descrivere il comportamento delle unità realizzate. Se i requisiti cambiano nel tempo o successivamente un altro sviluppatore o l'utente comprende che il comportamento di alcune unità non è quello desiderato, è sempre possibile consultare i test e modificarli. In questo modo, dopo la modifica, si

possono eseguire i tutti i test e, se questi segnalano degli errori, correggere le unità fino a che non si ottiene il comportamento desiderato.

Creazione del gioco del bowling

In questa sezione viene realizzato, seguendo un'approccio *Test-Driven Development*, un modulo che ha lo scopo di calcolare il punteggio di una partita del gioco del bowling (l'unità centrale del progetto BowlingScore). La prima parte di questo esempio è stato tratto da `http://butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt` dove viene specificata anche l'architettura generale.

I requisiti dell'unità sono specificati nel regolamento del WTBA[1] nella sezione 2.1 "Definizione di Partita". Nell'esempio riportato vengono considerati solo i punti 2.1.1 - 2.1.3, 2.1.6, 2.1.7 del regolamento.

Per iniziare la costruzione dell'unità è necessario creare il seguente file che in futuro conterrà i test dell'oggetto `BowlingGame`.

Listing 4.1: src/test/java/it/unipd/app/BowlingGameTest.java

```
1 package it.unipd.app;

3 import junit.framework.TestCase;

5 public class BowlingGameTest extends TestCase {
  }
```

Se vengono eseguiti i test della classe appena creata verrà segnalato che essa non contiene test di unità. Poiché viene seguendo un' approccio *Test-Driven Development* viene creato prima il primo test e poi viene sviluppata la classe `BowlingGame`. Il primo test verifica che il punteggio è zero quando, in una partita, tutti i lanci fanno cadere zero birilli.

Listing 4.2: src/test/java/it/unipd/app/BowlingGameTest.java

```
  package it.unipd.app;
2
  import junit.framework.TestCase;
4
  public class BowlingGameTest extends TestCase {
6
      public void testAllZeros() throws Exception {
8         BowlingGame g = new BowlingGame();
          for (int i = 0; i < 20; i++)
10            g.roll(0);
          assertEquals(0, g.score());
12    }
  }
```

Da questo test si può capire che la classe `BowlingGame` avrà i seguenti metodi:

- Il metodo `roll` che simula il lancio di una palla
- Il metodo `score` che ritorna il punteggio totalizzato in una partita

Se vengono eseguiti nuovamente i test, viene segnalato che la classe `BowlingGame` non è ancora stata creata. Di seguito viene creata la classe `BowlingGame` in modo da far superare, nel modo più semplice possibile, il test precedentemente creato.

Listing 4.3: src/main/java/it/unipd/app/BowlingGame.java

```
1 package it.unipd.app;

3 public class BowlingGame {

5     public void roll(int pins) {
      }
7
      public int score() {
9         return 0;
      }
11 }
```

[1] `http://www.fisb.org/RegoleGioco2006.html`

Se vengono eseguiti nuovamente i test (contenuti nella classe `BowlingGameTest`) questi daranno esito positivo. In questo modo è stato assicurato che il comportamento dell'unità `BowlingGame`, quando un giocatore effettua venti lanci che non colpiscono mai un birillo, calcoli il punteggio zero.

Di seguito viene creato un test che simula una partita dove tutti i lanci, effettuati da un giocatore, colpiscano sempre e solo un birillo. In questo caso l'unità dovrà calcolare che il punteggio realizzato durante la partita è venti.

Listing 4.4: src/test/it/unipd/app/BowlingGameTest.java

```
1  package it.unipd.app;
2
3  import junit.framework.TestCase;
4
5  public class BowlingGameTest extends TestCase {
6
7      public void testAllZeros() throws Exception {
8          BowlingGame g = new BowlingGame();
9          for (int i = 0; i < 20; i++)
10             g.roll(0);
11         assertEquals(0, g.score());
12     }
13
14     public void testAllOnes() throws Exception {
15         BowlingGame g = new BowlingGame();
16          for (int i = 0; i < 20; i++)
17              g.roll(1);
18          assertEquals(20, g.score());
19     }
20 }
```

Come si può notare, nel codice sorgente dei due test, viene violato il principio *DRY* (tra le righe 8-10 e le righe 15-17). In oltre la variabile rappresentante il gioco ha un nome ("g") poco chiaro. È quindi consigliato abbassare l'entropia della classe facendo un piccolo *refactoring* della classe `BowlingGameTest`.

Listing 4.5: src/test/java/it/unipd/app/BowlingGameTest.java

```
1  package it.unipd.app;
2
3  import junit.framework.TestCase;
4
5  public class BowlingGameTest extends TestCase {
6
7      private BowlingGame game;
8
9      protected void setUp() throws Exception {
10         game = new BowlingGame();
11     }
12
13     private void rollMany(int num, int pins) {
14         for (int i = 0; i < num; i++)
15             game.roll(pins);
16     }
17
18     public void testAllZeros() throws Exception {
19         rollMany(20,0);
20         assertEquals(0, game.score());
21     }
22
23     public void testAllOnes() throws Exception {
24         rollMany(20,1);
25         assertEquals(20, game.score());
26     }
27 }
```

In questo esempio sono state apportate le seguenti modifiche:

- Introdotta una variabile `game` di tipo `BowlinGame` che viene creata (tramite il metodo `setUp`) prima dell'esecuzione di ogni metodo di test
- Aggiunto un metodo privato (`rollMany`) che simula dei lanci consecutivi che colpiscono sempre lo stesso numero di birilli

A questo punto se vengono eseguiti nuovamente tutti i test, viene segnalato un fallimento in `testAllOnes` (risultato atteso 20, risultato ottenuto 0).

È quindi necessario modificare la classe `BowlingGame` in modo da far superare il test (sempre nel modo più semplice possibile).

Listing 4.6: src/main/java/it/unipd/app/BowlingGame.java

```
1  package it.unipd.app;
2
3  public class BowlingGame {
4
5      private int score = 0;
6
7      public void roll(int pins) {
8          score += pins;
9      }
10
11     public int score() {
12         return score;
13     }
14 }
```

Se vengono eseguiti nuovamente i test, questi daranno un esito positivo. Di seguito viene creato un nuovo test che simula una partita dove nel primo *frame* venga realizzato uno *spare*.

Listing 4.7: src/test/java/it/unipd/app/BowlingGameTest.java

```
1  package it.unipd.app;
2
3  import junit.framework.TestCase;
4
5  public class BowlingGameTest extends TestCase {
6
7      private BowlingGame game;
8
9      protected void setUp() throws Exception {
10         game = new BowlingGame();
11     }
12
13     private void rollMany(int num, int pins) {
14         for (int i = 0; i < num; i++)
15             game.roll(pins);
16     }
17
18     public void testAllZeros() throws Exception {
19         rollMany(20,0);
20         assertEquals(0, game.score());
21     }
22
23     public void testAllOnes() throws Exception {
24         rollMany(20,1);
25         assertEquals(20, game.score());
26     }
27
28     public void testOneSpare() throws Exception {
29         rollMany(2,5); // spare
30         rollMany(1,3);
31         rollMany(17,0);
32         assertEquals(16, game.score());
33     }
34 }
```

Eseguendo i test viene segnalato un fallimento (il valore atteso in `testOneSpare` è 16 mentre il valore ritornato è 13). È quindi necessario modificare il codice della classe `BowlingGame` in modo da superare il nuovo test.

Listing 4.8: src/main/java/it/unipd/app/BowlingGame.java

```
1  package it.unipd.app;
2
3  public class BowlingGame {
4
5      private int rolls[] = new int[21];
6      private int currentRoll = 0;
```

```
8       public void roll(int pins) {
            rolls[currentRoll++] += pins;
10      }

12      public int score() {
            int score = 0;
14          int frameIndex = 0;
            for (int frame = 0; frame < 10; frame++)  {
16              if (rolls[frameIndex] + rolls[frameIndex + 1] == 10) {// spare
                    score += 10 + rolls[frameIndex + 2];
18              }
                else {
20                  score += rolls[frameIndex] + rolls[frameIndex+1];
                }
22            frameIndex += 2;
            }
24          return score;
        }
26  }
```

Come si può notare, la classe è stata modifica nel seguente modo:

- Aggiunto un `array` contenente il punteggio di ogni singolo lancio (riga 5)
- Aggiunta una variabile che identifica la posizione attuale del giocatore nella partita (riga 6)
- Modificato il metodo `roll` (riga 8-10)
- Modificato il metodo `score` in modo che riconosca e calcoli il punteggio ottenuto dopo aver effettuato uno *spare* (riga 12-25)

Tramite l'esecuzione dei test, è possibile verificare che il comportamento dell'oggetto `BowlingGame` dopo le modifiche non è cambiato rispetto al passato. Se vengono eseguiti nuovamente tutti test, questi non segnaleranno nessun errore, quindi il comportamento della classe `BowlingGame`, dopo le modifiche, è stato preservato.

Il codice della classe dei test e della classe `BowlingGame`, contiene delle imperfezione, infatti la realizzazione di uno *spare* viene evidenziato solo da un commento. È consigliato inserire dei metodi che permettano di riprodurre e verificare il punteggio di uno *spare* (queste modifiche verranno apportate in seguito).

Il prossimo test verifica il comportamento di una partita dove nel primo *frame* viene realizzato uno *strike*.

Listing 4.9: src/test/java/it/unipd/app/BowlingGameTest.java

```
    package it.unipd.app;
2
    import junit.framework.TestCase;
4
    public class BowlingGameTest extends TestCase {
6
        private BowlingGame game;
8
        protected void setUp() throws Exception {
10          game = new BowlingGame();
        }
12
        private void rollMany(int num, int pins) {
14          for (int i = 0; i < num; i++)
                game.roll(pins);
16      }

18      public void testAllZeros() throws Exception {
            rollMany(20,0);
20          assertEquals(0, game.score());
        }
22
        public void testAllOnes() throws Exception {
24          rollMany(20,1);
            assertEquals(20, game.score());
26      }

28      public void testOneSpare() throws Exception {
            rollSpare();
30          rollMany(1,3);
            rollMany(17,0);
```

```
32          assertEquals(16, game.score());
        }
34
        public void testOneStrike() throws Exception {
36          rollStrike();
            game.roll(4);
38          game.roll(3);
            rollMany(16, 0);
40          assertEquals(24, game.score());
          }
42
        private void rollStrike(){
44          rollMany(1,10);
        }
46
        private void rollSpare() {
48          rollMany(2,5);
        }
50  }
```

In questa versione della classe `BowlingGameTest` sono stati aggiunti due metodi che permettono di effettuare uno *strike* e uno *spare* (rollStrike e rollSpare). Se vengono eseguiti nuovamente i test, viene segnalato che il valore atteso in `testOneStrike` (righe 35-41) è 24 mentre il valore ritornato è 17. È quindi necessario modificare il codice della classe `BowlingGame` in modo da far superare `testOneStrike`.

Listing 4.10: src/main/java/it/unipd/app/BowlingGame.java

```
    package it.unipd.app;
2
    public class BowlingGame {
4
        private int rolls[] = new int[21];
6       private int currentRoll = 0;

8       public void roll(int pins) {
            rolls[currentRoll++] += pins;
10      }

12      public int score() {
            int score = 0;
14          int frameIndex = 0;
            for (int frame = 0; frame < 10; frame++)  {
16              if (isStrike(frameIndex)) {
                    score += 10 + rolls[frameIndex+1] + rolls[frameIndex+2];
18                  frameIndex++;
                }
20              else {
                    if (isSpare(frameIndex)) {
22                      score += 10 + rolls[frameIndex + 2];
                    }
24                  else {
                        score += rolls[frameIndex] + rolls[frameIndex+1];
26                  }
                    frameIndex += 2;
28              }
            }
30          return score;
        }
32
        private boolean isSpare(int frameIndex) {
34          return rolls[frameIndex] + rolls[frameIndex + 1] == 10;
        }
36
        private boolean isStrike(int frameIndex) {
38          return rolls[frameIndex] == 10;
        }
40  }
```

Alla classe `BowlingGame` sono state fatte le seguenti modifiche:

- Aggiunti due metodi (`isStrike` e `isSpare`) che permettono di verificare se è stato realizzato uno *strike* o uno *spare*
- Modificato il metodo `score` in modo da identificare e calcolare il punteggio di uno *strike*

Se vengono eseguiti nuovamente i test, questi verranno superati con successo. Nel codice sono presenti ancora delle imperfezioni, infatti viene ripetuto più volte il calcolo del punteggio di un *frame* (queste modifiche verranno apportate in seguito).

Il prossimo test permette di simulare una partita perfetta:

Listing 4.11: src/test/java/it/unipd/app/BowlingGameTest.java

```
package it.unipd.app;

import junit.framework.TestCase;

public class BowlingGameTest extends TestCase {

    private BowlingGame game;

    protected void setUp() throws Exception {
        game = new BowlingGame();
    }

    private void rollMany(int num, int pins) {
        for (int i = 0; i < num; i++)
            game.roll(pins);
    }

    public void testAllZeros() throws Exception {
        rollMany(20,0);
        assertEquals(0, game.score());
    }

    public void testAllOnes() throws Exception {
        rollMany(20,1);
        assertEquals(20, game.score());
    }

    public void testOneSpare() throws Exception {
        rollManySpare(1);
        rollMany(1,3);
        rollMany(17,0);
        assertEquals(16, game.score());
    }

    public void testOneStrike() throws Exception {
        rollManyStrike(1);
        game.roll(4);
        game.roll(3);
        rollMany(16, 0);
        assertEquals(24, game.score());
      }

    public void testPerfetGame() throws Exception {
        rollManyStrike(12);
        assertEquals(300, game.score());
      }

    private void rollManyStrike(int num){
        rollMany(num,10);
    }

    private void rollManySpare(int num) {
        for(int i = 0; i < num; i++)
        rollMany(2,5);
    }
}
```

Se vengono eseguiti tutti i test, si può notare che anche `testPerfectGame` non crea problema. Successivamente viene effettuato un *refactoring* del codice per ridurre le ripetizioni presenti nella classe `BowlingGame`.

Listing 4.12: src/main/java/it/unipd/app/BowlingGame.java

```
package it.unipd.app;

public class BowlingGame {

    private int rolls[] = new int[21];
    private int currentRoll = 0;
```

```
8       public void roll(int pins) {
            rolls[currentRoll++] += pins;
10      }

12      public int score() {
            int score = 0;
14          int frameIndex = 0;
            for (int frame = 0; frame < 10; frame++)  {
16              if (isStrike(frameIndex)) {
                    score += strikeBonus(frameIndex);
18                  frameIndex++;
                }
20              else {
                    if (isSpare(frameIndex)) {
22                      score += spareBonus(frameIndex);
                    }
24                  else {
                        score += sumOfBallsInFrame(frameIndex);
26                  }
                    frameIndex += 2;
28              }
            }
30          return score;
        }
32
        private boolean isSpare(int frameIndex) {
34          return sumOfBallsInFrame(frameIndex) == 10;
        }
36
        private boolean isStrike(int frameIndex) {
38          return rolls[frameIndex] == 10;
        }
40
        private int sumOfBallsInFrame(int frameIndex) {
42          return rolls[frameIndex] + rolls[frameIndex + 1];
        }
44
        private int spareBonus(int frameIndex) {
46          return 10 + rolls[frameIndex + 2];
        }
48
        private int strikeBonus(int frameIndex) {
50          return 10 + sumOfBallsInFrame(frameIndex+1);
        }
52  }
```

Come si è potuto notare, tramite un approccio *Test-Driven Development* è stato abbastanza semplice creare e verificare il comportamento di una classe che simula il calcolo di una partita del gioco del bowling.

Specificare i dati di test esternamente

A volte, per uno sviluppatore, non è semplice capire se il comportamento di un'unità è corretto. Nell'esempio precedente, uno sviluppatore potrebbe non essere in grado di calcolare correttamente tutti i possibili risultati di una partita di bowling.

Risulta abbastanza scomodo scrivere molti test per assicurare il comportamento della maggior parte delle situazioni che possono verificarsi in un modulo. In questi casi risulta più comodo specificare i dati di test in file esterni. In questo modo questi dati possono essere specificati, in modo semplice e agevole, da persone che hanno maggior confidenza del dominio del progetto che si sta sviluppando.

Di seguito viene proposto un possibile esempio per specificare in modo agevole i valori di una partita di bowling. Viene realizzata una classe, visibile solo nell'ambiente di test, che permetta di leggere dei valori da dei file, scritti in *plain text*, che hanno le seguenti caratteristiche:

- Il campo # indica un commento
- Possono essere specificati dei valori numerici separati da uno spazio, dove il primo indica il punteggio realizzato durante una partita da bowling e i successivi indicano i punteggi realizzati per ogni tiro

Tramite questa specifica, se si vogliono specificare i valori dei test precedentemente creati, basta creare, con un *editor* di testo, il seguente file:

Listing 4.13: src/test/resoruces/testdata.txt

```
#
#  All zero
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
#  All one
20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
#  One spare
16 5 5 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
#  One strike
24 10 4 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
#  Perfect game
300 10 10 10 10 10 10 10 10 10 10 10 10
```

Di seguito viene creata la classe di utilità che permette di leggere i valori, da specificare nei test, da file scritti nella forma precedentemente descritta.

Listing 4.14: src/test/java/it/unipd/app/DataFromFile.java

```
package it.unipd.app;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class DataFromFile {

    private String line;
    private int expected;
    private BufferedReader reader;

    public DataFromFile(String fileName) throws Exception {
        reader = new BufferedReader(
                new InputStreamReader(
                        getClass().getResourceAsStream( fileName )));
    }

    public int getExpected() {
        return expected;
    }

    public int[] getDataFromFile()throws Exception {
        int[] arguments = null;
        while ((line = reader.readLine()) != null) {
            if(line.startsWith("#")){
                continue;
            }
            StringTokenizer st = new StringTokenizer (line);
            if (!st.hasMoreTokens()){
                continue; //Blank line
            }
            // Get expected value
            String val = st.nextToken();
            expected = Integer.valueOf(val).intValue();
            // And the arguments
            ArrayList <Integer> argument_list = new ArrayList <Integer> ();
            while (st.hasMoreTokens()){
                argument_list.add(Integer.valueOf(st.nextToken()));
            }
            arguments = new int[argument_list.size()];
            for (int i=0; i < argument_list.size(); i++) {
               arguments[i] = ((Integer)argument_list.get(i)).intValue();
            }
            break;
        }
        return arguments;
    }
}
```

La classe espone un costruttore e due metodi:

- Il costruttore permette di creare un oggetto che legge i valori da un file
- Il metodo `getDataFromFile` ritorna i punteggi di ogni lancio di una partita di bowling specificati nella riga corrente di un file

- Il metodo `getExpected` ritorna il valore atteso specificato nella riga corrente di un file

Per creare i test che verificano i valori specificati nel file si può creare la seguente classe:

Listing 4.15: src/test/java/it/unipd/app/FromFileTest.java

```
package it.unipd.app;

import junit.framework.TestCase;

public class FromFileTest extends TestCase{

    public void testFromSimpleFile() throws Exception {
        DataFromFile dataFromFile =
            new DataFromFile("src/test/resources/testdata.txt");
        int [] arguments;
        while((arguments = dataFromFile.getDataFromFile() )!= null) {
            BowlingGame game = new BowlingGame();
            for (int i = 0; i < arguments.length; i++)
                game.roll(arguments[i]);
            assertEquals(dataFromFile.getExpected(),game.score());
        }
    }
}
```

Se vengono eseguiti i test (con i valori specificati nel file `testdata.txt`) della classe appena creata, verranno superati tutti con successo.

In questo modo più persone, anche non programmatori, possono specificare agevolmente i dati dei test e lo sviluppatore in modo molto semplice potrà verificare che il comportamento del modulo che sta realizzando soddisfi i requisiti concordati.

4.2 Considerare le condizioni limite

Solitamente gli errori accadono in condizioni limite. Identificare le condizioni limite è una delle parti più importanti per creare delle buone unità. Un modo semplice per pensare alle condizioni limite che possono presentarsi in un'unità è quello di memorizzare l'acronimo *CORRECT*.

Per ogni elemento creato si dovrebbe verificare che:

- ***C**onformance* - I valori sono conformi al formato atteso?
- ***O**rdering* - I valori seguono o non seguono un ordine?
- ***R**ange* - I valori sono all'interno di un valore di minimo e massimo appropriato?
- ***R**eference* - I valori possono provenire da codice che si riferisce a dati esterni che non sono sotto il controllo del codice?
- ***E**xistence* - I valori esistono (non sono nulli, non sono zero, sono presenti in un determinato insieme)?
- ***C**ardinality* - I valori sono nella quantità desiderata?
- ***T**ime* - I valori rispettano un ordine temporale?

Con il termine valore si fa riferimento sia ai parametri di *input* dei metodi di un'unità, che ai dati interni all'unità e ai risultati che questa produce. Di seguito vengono descritte in dettaglio le principali condizioni di valori limite da considerare.

4.2.1 Rispettare un formato

In un progetto capita di dover ricevere o restituire dei valori conformi ad uno specifico formato. Per esempio un indirizzo *e-mail*, un indirizzo *IP*, un numero di telefono, un *bar code* non sono delle semplici stringhe.
Cosa succede al programma se sbadatamente viene inserito un valore non conforme ad uno specifico formato?
Le componenti sviluppate ritornano dei dati conformi ad uno specifico formato?
Uno sviluppatore deve essere in grado di rispondere a queste domande per tutte le componenti sviluppate in un

progetto. È consigliato che uno sviluppatore crei dei test di unità che permettono di verificare se i valori gestiti dalle varie componenti realizzate sono conformi ad un determinato formato e assicurare che, se un valore non è conforme ad un formato prestabilito, venga segnalato un errore (o nei casi più estremi venga terminato il programma).

Per esempio in una componente che gestisce dei dati che rappresentano un'indirizzo *e-mail* cosa succede se:

- L'indirizzo *e-mail* non contiene il simbolo @
- L'indirizzo *e-mail* non contiene dei punti
- L'indirizzo *e-mail* contiene più di un punto (p.es. mario.rossi@subdomain.xyx.com)
- L'indirizzo *e-mail* contiene dei caratteri non consentiti

Come si può vedere la gestione di un dato apparentemente semplice come l'indirizzo *e-mail* può far sorgere molti dubbi riguardanti il formato. Questi dubbi devono essere colmati e verificati tramite i test di unità, in modo da assicurare che le componenti sviluppate gestiscano correttamente dati conformi ad un determinato formato, e che non si verifichino immissioni di dati errati.

4.2.2 Rispettare un ordine

Un altro aspetto molto importante è l'ordine dei dati, o la posizione in cui accade un determinato evento. Questo aspetto riguarda qualsiasi funzionalità che effettua una ricerca all'interno di una lista di elementi. La ricerca di un elemento dovrebbe essere testa almeno nelle seguenti condizioni:

- L'elemento da ricercare è nella prima posizione della lista
- L'elemento da ricercare è nell'ultima posizione della lista
- L'elemento da ricercare è in una posizione intermedia

Effettuando questi controlli è molto probabile trovare se sono presenti degli errori nella funzionalità di ricerca.

Un'altro aspetto molto importante da considerare è la posizione in cui avviene un determinato evento e la sua influenza nei risultati prodotti dall'unità. Supponiamo di dover scrivere una funzionalità che riceve e schedula gli ordini di un ristorante. Se per caso le persone sedute in un tavolo effettuano le seguenti ordinazioni:

1. Un dolce
2. Un primo
3. Una bibita

In che ordine devono essere servite le ordinazioni?
È giusto imporre un ordine?
Supponiamo che una mamma ordini un primo e un caffè, e il figlio ordini solo un dolce, come ci si deve comportare?
È desiderato quindi dare la maggior flessibilità possibile al programma, e assicurare comportamenti limiti e comportamenti normali attraverso i test di unità.

Esempio di test per verificare aspetti riguardanti l'ordine

Nel progetto BowlingScore il requisito indicato nella regola 2.1.1:

> "Se nel decimo *frame* il giocatore realizza uno *strike* o uno *spare* lancia tre palle."

È un requisito che dipende dal ordine in cui accadono gli eventi, infatti :

- Se nell'ultimo *frame* un giocatore effettua uno *strike* ha diritto ad effettuare altri due lanci
- Se nell'ultimo *frame* un giocatore effettua uno *spare* ha diritto ad effettuare un altro lancio.

La terminazione del gioco dipende dal punteggio realizzato nel decimo *frame*. Il punteggio totale del gioco quindi può essere calcolato solo alla terminazione del gioco. Di seguito viene aggiunto alla classe `BowlingGameTest` un test che verifica che il metodo `score` non possa essere invocata prima della terminazione del gioco.

Listing 4.16: src/test/java/it/unipd/app/BowlingGameTest.java

```
    public void testNotEndGame() {
2     try{
          game.score();
4         fail("Should␣have␣thrown␣an␣exception");
      } catch (RuntimeException e){
6         assertTrue(true);
      }
8   }
```

Se vengono eseguiti tutti i test realizzati, verrà segnalato un errore in `testNotEndGame`. Per superare l'ultimo test creato è necessario modificare la classe `BowlingGame` in modo da verificare che il metodo `score` non venga invocata se non sono avvenuti lanci. Supponiamo si voglia verificare che il metodo `score` non possa essere invocata dopo l'undicesimo lancio di una partita perfetta. Viene quindi aggiunto il seguente test:

Listing 4.17: src/test/java/it/unipd/app/BowlingGameTest.java

```
    public void testNotEndPerfectGame() {
2       try{
            rollManyStrike(11);
4           game.score();
            fail("Should␣have␣thrown␣an␣exception");
6       } catch (RuntimeException e){
            assertTrue(true);
8       }
      }
```

Se vengono eseguiti nuovamente tutti i test, verrà segnalato un errore in `testNotEndPerfectGame`. Per superare l'ultimo test creato è necessario modificare radicalmente la classe `BowlingGame`. La nuova realizzazione che dovrà essere realizzata dovrà avere le seguenti caratteristiche:

- I punteggi di ogni *frame* devono essere salvati in un apposita classe (`Frame`)
- il gioco termina se il giocatore ha raggiunto il decimo *frame* e questo è terminato
- il metodo `score`, prima di eseguire, deve controllare che il gioco sia terminato
- L'ultimo *frame* deve permettere di effettuare tre lanci se è stato realizzato uno *strike* o uno *spare*

La classe `BowlingGame` che permette di superare i nuovi test può essere recuperata dal codice d'esempio del progetto BowlingScore. Anche se le modifiche comportano un cambiamento radicale della classe, il comportamento di questa verrà preservato grazie ai test realizzati fino ad ora.

4.2.3 Appartenere ad un insieme di valori

Un dato presente in un poggetto solitamente rappresenta un dato reale. I linguaggi di programmazione offrono dei tipi, (p.es. interi, reali, booleani, stringhe) che permettono di rappresentare la maggior parte dei dati, senza però permettere di inserire direttamente dei controlli sui valori ammissibili che può assumere il dato.

Se per esempio si utilizza un tipo intero per rappresentare i gradi di un angolo, questo potrebbe assumere valori non ammissibili nella realtà (p.es. numeri negativi o numeri maggiori di 360). Una bona pratica nella programmazione ad oggetti è quella di limitare i valori dei dati che si vogliono rappresentare in un oggetto, in modo da poter definire, in modo esplicito delle proprietà.

Nel progetto BowlingScore, il punteggio di un tiro deve essere compreso tra 0 e 10. È consigliato quindi rappresentare questo dato in un oggetto. Viene creata una nova classe (`RollTest`) contenente due test che verificano che il punteggio realizzato in un tiro non possa essere superiore a 10 ed inferiore a 0.

Listing 4.18: scr/test/java/it/unipd/app/RollTest.java

```
1 package it.unipd.app;

3 import junit.framework.TestCase;
```

```
5  public class RollTest extends TestCase {

7      private Roll roll;

9      protected void setUp() {
           roll = new Roll();
11     }

13     public void testNegativeRoll() {
           try{
15             roll.setPins(-1);
               fail("Should have thrown an exception");
17         }catch(RuntimeException e) {
               assertTrue(true);
19         }
       }
21
       public void testOverTenRoll() {
23         try{
               roll.setPins(11);
25             fail("Should have thrown an exception");
           }catch(RuntimeException e) {
27             assertTrue(true);
           }
29     }

31 }
```

Se viene eseguito il test viene segnalato che la classe `Roll` non esiste. È quindi necessario creare la classe `Roll` che permetta di assicurare che il che il punteggio di un tiro sia compreso tra 0 e 10.

Listing 4.19: scr/main/java/it/unipd/app/Roll.java

```
1  package it.unipd.app;

3  public class Roll {

5      public void setPins(int pins) {
           if (pins < 0 || pins > 10) {
7              throw new RuntimeException("Roll out of range: [0 - 10]");
           }
9      }
   }
```

Se vengono eseguiti nuovamente i test, questi verranno superati dalla classe `Roll`. Di seguito viene creato un nuovo test che verifica che la classe `Roll` ritorni il punteggio totalizzato.

```
       public void testGetPins() {
2          roll.setPins(5);
           assertEquals(5, roll.getPins());
4      }
```

Se vengono eseguiti i test verrà segnalato un errore in `testGetPins`. Di seguito viene modificata la classe `Roll` in modo da superare il test.

Listing 4.20: scr/main/java/it/unipd/app/Test.java

```
   package it.unipd.app;
2
   public class Roll {
4
       private int pins = 0;
6
       public void setPins(int p) {
8          if (p < 0 || p > 10) {
               throw new RuntimeException("Roll out of range: [0 - 10]");
10         }
           pins = p;
12     }

14     public int getPins() {
           return pins;
16     }
   }
```

Se vengono eseguiti nuovamente i test, questi verranno superati con successo dalla classe `Roll`. A questo punto, per assicurare che la calasse `BowlingGame` consenta di effettuare solo punteggi reali, è necessario modificare la classe `Frame`.

Alcuni proprietà limite possono dipendere da più valori. Per esempio nel progetto BowlingScore, il punteggio di due tiri non deve essere superiore a dieci. In queste situazioni è consigliato aggiungere alla classe `Frame` dei metodi di controllo che permettano di verificare la presenza di situazioni di errore. Viene aggiunto un nuovo test alla classe `Frame`.

Listing 4.21: scr/test/java/it/unipd/app/FrameTest.java

```
1     public void testFrameScore() {
          try {
3             frame.roll(6);
              frame.roll(7);
5             fail();
          } catch (RuntimeException e) {
7             assertTrue(true);
          }
9     }
```

Questo test permette di assicurare che, se il punteggio di un *frame* è superiore a dieci, venga lanciata un'eccezione. Se vengono eseguiti i test della classe `Frame` il metodo `testFrameScore` non verrà superato.

È quindi necessario aggiungere alla classe `Frame` un metodo che permetta di controllare che il punteggio realizzato all'interno di un frame sia minore o uguale a dieci.

Listing 4.22: scr/main/java/it/unipd/app/Frame.java

```
1 public void roll(int pins) {
          rolls[numOfRoll].setPins(pins);
3         checkScore();
          addARoll();
5     }
  ..
7 protected void checkScore(){
          if (sumOfBallsInFrame() > 10) {
9             throw new RuntimeException("Score␣out␣of␣range");
          }
11    }
```

Se vengono eseguiti nuovamente tutti i test del poggetto, questi verranno superati con successo.

Come si può vedere, in questo modo, è possibile assicurare attraverso i test di unità, che una classe gestisca solo valori ammissibili.

4.2.4 Considerare lo stato interno ed esterno

A volte capita che un unità funzioni correttamente solo se sono presenti determinate condizioni. Le condizioni che si possono incontrare possono dipendere dallo stato interno e da alcuni componenti esterne all'unità. È buona norma quindi identificare le precondizioni e le postcondizioni che devono verificarsi prima e dopo l'esecuzione di una funzionalità di una componente.

Le precondizioni di una funzionalità specificano come deve essere lo stato interno o esterno alla componente per permettere di far eseguire la funzionalità. Ad esempio le precondizioni dei metodo della classe `BowlingGame` sono:

- Il metodo `roll` può essere eseguito solo se la partita non è terminata
- Il metodo `score` può essere eseguito solo se la partita è terminata

Ad esempio le precondizioni di un metodo che effettua la lettura di un file sono:

- Il metodo può aprire e leggere il file solo se esso esiste
- il metodo può aprire e leggere il file solo se è consentita la lettura

Le postcondizioni di una funzionalità specificano come viene modificato lo stato interno o esterno alla componente al termine della sua esecuzione. Ad esempio le postcondizioni dei metodi della classe `BowlingGame` sono:

- Al termine dell'esecuzione del metodo `roll`, se viene terminato un *frame*, il gioco passa al frame successivo
- Al termine dell'esecuzione del metodo `score` la partita rimane terminata

Ad esempio le postcondizioni del metodo che effettua la lettura di un file sono:

- Al termine dell'esecuzione del metodo il file viene chiuso

Una buona pratica è quella di documentare tutte le precondizioni e le postcondizioni di tutti i metodi di una classe. È quindi necessario realizzare dei test di unità che permettano di riprodurre le precondizioni eseguire la funzionalità e verificarne le post condizioni.

Nel progetto BowlingScore è possibile creare un test che permetta di verificare le precondizioni necessarie e le post condizioni del metodo `roll` nel seguente modo:

```
1     public void testPrePostConditionRoll() {
          for (int i = 0; i < 10; i++){
3             assertFalse(game.finish());
              assertEquals(i,game.getFramePosition());
5             game.roll(1);
              assertFalse(game.finish());
7             assertEquals(i,game.getFramePosition());
              game.roll(1);
9         }
          assertTrue(game.finish());
11    }
```

Se viene eseguito il test verrà segnalato che è impossibile accedere al metodo `finish` e al metodo `getFramePosition`. É quindi necessario modificare la classe `BowlingGame` e rendere pubblico il metodo `finish` e creare un metodo `getFramePosition` che ritorni il *frame* corrente.

```
1     public boolean finish() {
          return currentFrame == frames.length &&
3         frames[currentFrame -1].finish();
      }
5
      public int getFramePosition() {
7         return currentFrame;
      }
```

Se vengono eseguiti nuovamente tutti i test, questi verranno superati con successo.

4.2.5 Considerare l'assenza di dati

Molti errori possono presentarsi se è assente un dato all'interno di un'unità. Per assenza di un dato ci si riferisce alla mancanza di:

- Valori nei parametri di *input*
- Dati interni o esterni che servono all'unità per un corretto funzionamento

È quindi consigliato pensare come si comporta un'unità se non sono presenti alcuni dati. Ad esempio per le funzionalità che ricevono in *input* dei valori si dovrebbe pensare a cosa potrebbe succedere se viene passato un valore nullo, o una stringa vuota o una quantità nulla.

Molte funzionalità presenti nelle librerie *Java*, se ricevono un parametro in *input* che non esiste (p.es. un oggetto `null`), viene lanciata un'eccezione che segnala e permette di gestire l'errore.

Molti metodi, se alcuni dati non esistono, continuano la loro esecuzione, provocando in seguito dei gravi problemi. Molto spesso la non gestione di un errore può provocare problemi molto gravi, che possono compromettere il corretto funzionamento del programma e recare danno a chi utilizza il programma. Per esempio, si supponga di dover realizzare un programma per palmari, che effettua delle ordinazioni in un ristorante. Supponiamo che la funzionalità di inoltro dell'ordinazione alle cucine non segnali un errore se il palmare non è connesso alla rete. In questo caso molte ordinazioni potrebbero andare perse creando dei danni economici ai ristoranti che utilizzano questo programma.

È consigliato verificare il comportamento delle unità nei casi in cui alcuni dati, interni o esterni, non esistono, in modo da prevenire e gestire i possibili errori che si potrebbero verificare.

4.2.6 Considerare la cardinalità

Quando si creano delle unità è necessario considerare almeno due aspetti riguardanti la cardinalità:

- Il calcolo delle quantità
- Il numero di elementi che possono essere gestiti all'interno di un'unità

Per esempio, per assicurare che una funzionalità, che ricerchi il massimo valore all'interno di una lista di elementi, funzioni correttamente è opportuno controllare il comportamento della ricerca con delle liste che contengano zero uno e più di un elemento.

Se si deve realizzare una classifica dei migliori tre elementi presenti in una lista (rispetto a qualche criterio) si dovrebbero considerare le seguenti situazioni:

- È possibile realizzare la classifica con meno di tre elementi?
- È possibile realizzare la classifica quando non ci sono elementi?
- È possibile realizzare la classifica quando c'è solo un elemento?
- È possibile aggiungere un nuovo elemento quando esistono già tre elementi in classifica?
- È possibile aggiungere un nuovo elemento quando non ci sono ancora tre elementi in classifica?

Nel progetto BowlingScore vengono gestiti i seguenti dati:

- Dieci *frame*
- Due lanci per *frame*
- Dieci birilli abbattibili per ogni *frame*

Cosa succede se per caso vengono cambiate le regole del gioco?(undici *frame* al posto di dieci, tre lanci al posto di due, venti birilli al posto di dieci)
L'unità è abbastanza flessibile da poter essere agevolmente cambiata in base ai cambiamenti futuri?
È quindi necessario creare l'unità e i test in modo che questi siano flessibili, e considerare le condizioni limite di cardinalità con quantità 0, 1, e n dove n può essere modificato nel tempo.

4.2.7 Considerare gli aspetti temporali

L'ultima condizione limite che compone l'acronimo *CORRECT* riguarda l'aspetto temporale. Ci sono molti aspetti riguardanti il tempo che devono essere controllati in un'unità. Le più importanti condizioni limite riguardanti il tempo si possono riassumere in tre categorie:

- Tempo relativo (ordine in cui accadono determinati eventi)
- Tempo assoluto (tempo in cui accade un evento rispetto ad un tempo di riferimento)
- Aspetti riguardanti la concorrenza

La prima categoria è già stata trattata nella sezione 4.2.2 relativa all'ordinamento. Nel progetto BowlingScore il metodo `score` della classe `BowlingGame` non può essere invocato prima che non siano stati completati tutti i *frame* che compongono una partita. È quindi necessario considerare cosa accade se un metodo viene invocato senza rispettare un determinato ordine, e permettere di gestire le situazioni d'errore.

Un'altro aspetto riguardante il tempo relativo è il tempo d'attesa che un metodo è disposto ad attenere prima che si verifichi un evento. Se un evento non accade è accertabile aspettare all'infinito?
Se si realizza una funzionalità che deve accedere ad una risorsa in rete, e la rete non è disponibile, per quanto tempo si può attendere che la rete diventi disponibile? È quindi necessario creare e verificare il comportamento di un'unità in modo da considerare gli aspetti riguardanti il tempo relativo.

La seconda categoria riguarda il tempo assoluto. Tutte le applicazioni che operano con un tempo assoluto possono soffrire di alcuni problemi relativi al sistema di riferimento del tempo. Una domanda che potrebbe far sorgere qualche dubbio è la seguente:

Tutti i giorni durano 24 ore?

La risposta non è così scontata, perché secondo la convenzione *UTC*[2] tutti i giorni sono di 24 ore mentre nella convenzione *DST*[3] no (in primavere esiste un giorno di 23 ore e in autunno uno da 25 ore). Le unità che effettuano operazioni aritmetiche con dei dati rappresentanti il tempo dovrebbero essere sviluppate in modo da considerare il sistema di riferimento del tempo e assicurare che il comportamento sia quello desiderato, tenendo in considerazioni i casi limite (p.es. l'ultimo giorno di febbraio, calcolo delle ore in primavera e in autunno, etc.).

L'ultima aspetto del tempo che si dovrebbe considerare è il comportamento di una funzionalità in un'ambiente concorrente. Visto che in alcuni linguaggi di programmazione (come *Java*) è possibile creare programmi con più flussi di controllo, è desiderato pensare a come deve essere il comportamento di un'unità se viene eseguita in un'ambiente concorrente.
Cosa può accadere se un'unità è usata contemporaneamente da più flussi di controllo?
Alcuni metodi devono eseguire in mutua esclusione?
Nel progetto BowlingScore, cosa succede se un istanza della classe `BowlingGame` è condivisa da più flussi di controllo che possono eseguire contemporaneamente la funzione `roll`? È quindi utile pensare e assicurare il comportamento di un'unità considerando che essa può essere eseguita in un ambiente concorrente.

4.3 Controllare le relazioni inverse

Alcune unità possono o devono essere verificati tramite l'applicazione della loro funzionalità inversa. Si supponga di voler creare un'unità che calcoli la radice quadrata di un numero. Per testare se la radice quadrata è corretta è possibile elevare al quadrato il risultato ritornato dall'unità e confrontarlo con il dato di partenza.

Altre volte per verificare il comportamento di alcune funzionalità, è necessario utilizzare le relazioni inverse. Si supponga di voler controllare il risultato dell'inserimento di un un elemento in una pila. Il modo più semplice per verificare se l'inserimento è andato a buon fine è quello di effettuare un prelevamento dalla pila e controllare che l'elemento ritornato sia l'elemento di partenza.

Quando possibile, per verificare il corretto funzionamento dell'unità è consigliato utilizzare un metodo inverso esterno ed affidabile. In questo modo si minimizzano le possibilità che il metodo inverso contenga degli errori.

4.4 Verificare il comportamento utilizzando altri strumenti

A volte un'unità può replicare alcune funzionalità realizzate in precedenza o da librerie esterne. Può accadere che lo sviluppatore crei un algoritmo in modo che sia più performante o che abbia delle caratteristiche diverse da quelle presenti in versioni precedenti o in altre implementazioni. Supponiamo che in una versione precedente un'unità realizzi un algoritmo con complessità esponenziale. Supponiamo che il comportamento di questa unità sia controllato dai test di unità e che funzioni correttamente per anni. Dopo alcuni anni viene realizzata una versione dello stesso algoritmo con complessità lineare, e che lo sviluppatore, per migliorare le prestazioni del programma, voglia ricreare l'unità. Per verificare che la nuova unità funzioni correttamente è consigliato confrontare i risultati delle due versioni, in modo da verificare di non introdurre nuovi errori.

In questi casi è consigliato, attraverso i test di unità, confrontare i risultati dell'algoritmo realizzato con quelli presenti in altre librerie, che molto probabilmente producono risultati affidabili. In questo modo è possibile verificare che le due unità, anche se realizzate diversamente, funzionino allo stesso modo.

4.5 Forzare condizioni d'errore

Nel mondo reale gli errori accadono. Una buona norma, per creare un buon progetto, è quello di ricreare le condizioni di errore e verificare che il progetto funzioni come ci si aspetta in queste condizioni.

[2] *Universal Coordinated Time*
[3] *Daylight Saving Time*

Esistono due tipologie di errori, errori interni e errori esterni al progetto. Per errori interni, ci si riferisce a tutti gli errori che possono essere ricreati all'interno del progetto. Nel progetto BowlingScore alcuni errori interni sono:

- Il punteggio di un *frame* è superiore a dieci
- Il punteggio di un lancio è superiore a dieci
- Viene richiesto il punteggio totale prima che il gioco sia terminato
- Viene effettuato un lancio dopo che la partita è terminata

Come spiegato precedentemente, tramite i test di unità, è possibile verificare che il progetto realizzato gestisca nella maniera desiderata i possibili errori che si possono verificare.

Gli errori esterni, sono gli errori che vengono causati da componenti esterni al progetto. Per esempio alcuni errori esterni comuni possono essere:

- Terminato lo spazio nell'hard disk
- Problemi legati alla rete
- Problemi legati alla memoria
- Errori legati al sistema operativo

Nel capitolo 4.7 viene spiegata una tecnica per testare le unità che dipendono da componenti esterne, in modo da poter simulare e gestire le situazioni d'errore.

4.6 Misurare le prestazioni

Un aspetto da non trascurare è il tempo d'esecuzione e la complessità di un'unità. Risulta utile creare dei test che permettano di misurare alcuni aspetti riguardanti le prestazioni di alcune unità.

Molto spesso capita che una funzionalità di un progetto, dopo alcune versioni inizi a diventare lenta. Altre volte capita che il tempo di esecuzione di un metodo eseguito con pochi valori di *input* risulta accettabile ma con molti elementi risulti inaccettabile.

Esistono alcuni strumenti che permettono di controllare il tempo di esecuzione dei test di unità. In questo modo si riesce a controllare e verificare che il tempo di esecuzione di una funzionalità, realizzata da un'unità, in differenti versioni del progetto o in diversi ambienti di esecuzione rimanga inferiore ad un certo limite temporale prestabilito. In questo modo quando si identifica un rallentamento, o una diminuzione delle prestazioni è possibile intervenire ed effettuare un attività di *refactorning* che permetta di migliorare il progetto.

Nella successiva sezione viene descritto uno strumento che permette di misurare le prestazioni delle unità di un programma.

4.6.1 *JunitPerf*

JunitPerf[4] è uno strumento che permette di misurare le prestazioni e la scalabilità delle funzionalità di un programma che sono verificate attraverso i test di unità.

Questo strumento è stato creato adottando il *pattern decorator* e fornisce due funzionalità che permettono di eseguire i test, realizzati attraverso *JUnit*, e verificarne le prestazioni e la scalabilità delle funzionalità e delle unità che compongono un programma. Le due principali funzionalità di questo strumento sono:

TimedTest: Test decorator che permette di misurare il tempo di esecuzione di un test;

LoadTest: Test decorator che simula l'esecuzione di un test da un numero concorrente di utenti per un numero di iterazioni;

[4] `http://clarkware.com/software/JUnitPerf.html`

Come viene descritto nella documentazione, reperibile all'indirizzo:
`http://clarkware.com/software/JUnitPerf.html`, per verificare se l'esecuzione di un test impiega un tempo inferiore ad un certo limite è possibile realizzare la seguente classe di test:

Listing 4.23: Esempio tratto da `http://clarkware.com/software/JUnitPerf.html`

```
import com.clarkware.junitperf.*;
import junit.framework.Test;

public class ExampleTimedTest {

    public static Test suite() {

        long maxElapsedTime = 1000;

        Test testCase = new ExampleTestCase("testOneSecondResponse");
        Test timedTest = new TimedTest(testCase, maxElapsedTime);

        return timedTest;
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }

}
```

Come si può vedere dal codice la classe utilizza le librerie di *JUnit* e le librerie di *JUnitPerf*. Nella classe viene realizzata una suite dove viene creato un test che permette di eseguire attraverso *JUnit* il metodo `testOneSecondResponse` della classe `ExampleTestCase`. Nella riga 11 viene decorato il test con la classe `TimedTest`. In questo modo all'esecuzione della classe di test verrà controllato che l'esecuzione di `testOneSecondResponse` (incluso il tempo di esecuzione dei metodi di `setUp` e `tearDown`) impieghi meno di un secondo, altrimenti il test fallirà.

Per verificare se un'unità è scalabile (il suo comportamento non cambia in base al numero di esecuzioni parallele) è possibile creare il seguente test:

Listing 4.24: Esempio tratto da `http://clarkware.com/software/JUnitPerf.html`

```
import com.clarkware.junitperf.*;
import junit.framework.Test;

public class ExampleLoadTest {

    public static Test suite() {

        int users = 10;
        int iterations = 20;
        Timer timer = new ConstantTimer(1000);
        Test testCase = new ExampleTestCase("testOneSecondResponse");
        Test loadTest = new LoadTest(testCase, users, iterations, timer);
        return loadTest;
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }

}
```

In questo esempio viene verificato che il test `testOneSecondResponse` eseguito contemporaneamente da 10 utente (che iniziano la loro esecuzione a distanza di un secondo) per 20 volte funzioni sempre correttamente.

Come viene descritto nella documentazione di *JUnitPerf*, combinando le classi `TimedTest` e `LoadTest` è possibile creare:

- Test di carico: Permette di verificare che il tempo di esecuzione di ogni test, eseguiti parallelamente da un numero stabilito di utenti, rimanga sempre inferiore ad una certo limite temporale. Questa tipologia di test si può ottenere decorando un `TimedTest` con un `LoadTest`;

- Test di *Throughput*: Permette di verificare che il tempo globale di esecuzione di tutti i test, eseguiti parallelamente da un numero stabilito di utenti, sia inferiore ad un certo limite temporale. Questa tipologia di test si può ottenere decorando un `LoadTest` con un `TimedTest`;

Attraverso questo strumento è quindi possibile creare dei test *JUnit* che possono verificare se le prestazioni di un programma rimangono costanti nel tempo.

4.7 Verificare le unità in un ambienti isolati

I test di unità hanno l'obiettivo di esercitare una funzionalità del programma per volta, in un'ambiente isolato. A volte capita che le unità dipendono da altre unità o da componenti esterne difficili da controllare e con comportamenti non simulabili (p.es. un *database*, problemi di connessione della rete, etc.).

In questa sezione vengono descritti alcuni metodi utili per verificare in modo isolato il comportamento delle unità che hanno dipendenze con altre unità o componenti estere.

Mock Object

il *Mock Object* è un pattern di test che permetti di verificare in modo efficace il comportamento di una unità che dipende da altre unità. La tecnica *Mock Object* consiste nel sostituire la componente esterna all'unità da testare con una realizzazione "finta" che ne simuli il comportamento. Questa tecnica risulta utile per creare test di unità indipendenti, per le unità che dipendono da:

- Componenti che hanno un comportamento non deterministico
- Componenti che hanno comportamenti difficili da riprodurre (p.es. la produzione di un errore in una rete)
- Componenti lente
- Componenti *hardware*
- Componenti che ancora non esistono o sono in fase di sviluppo

I tre passi per creare ed utilizzare un *Mock Object* sono:

1. Creare un'interfaccia che descrive il comportamento della componente
2. Realizzare la componente per il codice di produzione con i metodi specificati nell'interfaccia
3. Realizzare il *Mock Object* con i metodi specificati nell'interfaccia per il test

Il codice in produzione sotto test farà riferimento alla componente esterna solo attraverso la sua interfaccia, in questo modo sarà più indipendente e potrà usare senza problemi qualsiasi realizzazione dell'interfaccia.

Esempio di applicazione del *pattern Mock Object*

Supponiamo si voglia creare una classe che permette di effettuare una partita di bowling tra più giocatori. Supponiamo che la classe `BowlingGame` non sia stata ancora terminata e che uno sviluppatore stia sviluppando parallelamente la classe per effettuare una partita tra più giocatori. In questo caso è opportuno utilizzare la tecnica del *Mock Object* per permettere allo sviluppatore, che dovrà creare la nuova classe, di poter lavorare parallelamente allo sviluppatore che sta lavorando sulla classe `BowlingGame`.

L'interfaccia che definisce il comportamento della classe `BowlingGame` è la seguente:

Listing 4.25: src/main/java/it/unipd/app/BowlingGameInterface.java

```
1 package it.unipd.app;
2
3 public interface BowlingGameInterface {
4
5     public void roll(int pins);
6
7     public int score();
```

```
8
      public boolean finish();
10
      public int getFramePosition();
12
}
```

I metodi hanno le seguenti funzionalità:

- `roll` permette di simulare il lancio e specificare il numero di birilli che sono stati abbattuti
- `score` permette di ritornare il risultato finale della partita
- `finish` permette di verificare se una partita è finita
- `getFramePosition` ritorna il *frame* che il giocatore ha raggiunto

Lo sviluppatore, che dovrà creare la classe che permetterà di simulare una partita tra più giocatori, potrà creare una nuova realizzazione dell'interfaccia `BowlingGameInterface` e utilizzarla per verificare alcuni aspetti della nuova classe.

Listing 4.26: src/test/java/it/unipd/app/MockBowlingGame.java

```
1  package it.unipd.app;

3  public class MockBowlingGame implements BowlingGameInterface {

5      static private int numFrame = 3;
       private int frame = 0;
7      private int score = 0;

9      public boolean finish() {
           return frame == numFrame;
11     }

13     public int getFramePosition() {
           return frame;
15     }

17     public void roll(int pins) {
           score += pins;
19         frame ++;
       }
21
       public int score() {
23         return score;
       }
25
}
```

La classe di test realizzata, deve avere solo alcune caratteristiche che interessano allo sviluppatore. In questo caso lo sviluppatore che realizza il *multiplayer* è interessato che:

- Si possa simulare un lancio e il punteggio realizzato con il lancio influenzi il risultato finale (righe 17-20)
- Ci sia un avanzamento crescente dei *frame* (riga 19)
- Sia possibile verificare quando il gioco è terminato (righe 9-11)
- Sia possibile ricavare il punteggio totalizzato nella partita (righe 22-24)

Non è detto che questa realizzazione sia sufficiente per tutti i test della nuova classe, ma lo sviluppatore potrà sempre creare nuove implementazioni dell'interfaccia per simulare il comportamento di alcune funzionalità dell'oggetto `BowlingGame`. In questo modo lo sviluppatore potrà iniziare a creare la nuova classe che dipenderà solo dall'interfaccia `BolingGameInterface`. I test utilizzeranno gli oggetti di simulazione, mentre il codice in produzione utilizzerà la classe reale.

Grazie alla classe `mockBowlingGame` lo sviluppatore potrà creare il primo test per iniziare a sviluppare la nuova classe:

Listing 4.27: src/test/java/it/unipd/app/MultiplayerBowlingGameTest.java

```
package it.unipd.app;

import junit.framework.TestCase;

public class MultiplayerBowlingGameTest extends TestCase {

    public MultiplayerBowlingGameTest(String name) {
        super(name);
    }

    public void testGetWinner() {
        MultiplayerBowlingGame mpGame = new MultiplayerBowlingGame();
        String player1Name = "player 1";
        String player2Name = "player 2";
        mpGame.addPlayer(player1Name, new MockBowlingGame());
        mpGame.addPlayer(player2Name, new MockBowlingGame());
        while (!mpGame.finish()) {
            mpGame.roll(1);
            mpGame.roll(2);
        }
        assertEquals(player2Name, mpGame.getWinner());
    }

}
```

Questo test permette di simulare il *multiplayer* di due giocatori e verificare il nome del vincitore. come si può vedere è stato deciso di creare una metodo `add` che permette di aggiungere un giocatore ad un'istanza di gioco. Una prima realizzazione della classe `MultiplayerBowlingGame` per superare questo test è la seguente:

Listing 4.28: src/main/java/it/unipd/app/MultiplayerBowlingGame.java

```
package it.unipd.app;

import java.util.ArrayList;

public class MultiplayerBowlingGame {

    private ArrayList<BowlingGameInterface> playersGame;
    private ArrayList<String> playersName;
    private int currentPlayerIndex = 0;

    public MultiplayerBowlingGame() {
        playersGame = new ArrayList<BowlingGameInterface>();
        playersName = new ArrayList<String>();
    }

    public void addPlayer(String playerName,
            BowlingGameInterface bowlingGame) {
        playersGame.add(bowlingGame);
        playersName.add(playerName);
    }

    public boolean finish() {
        boolean finish = true;
        for (int i = 0; i < playersGame.size(); i++) {
            finish = finish && playersGame.get(i).finish();
        }
        return finish;
    }

    public void roll(int i) {
        int currentFrame = playersGame.get(currentPlayerIndex).getFramePosition();
        playersGame.get(currentPlayerIndex).roll(i);
        if (currentFrame != playersGame.get(currentPlayerIndex).getFramePosition())
            currentPlayerIndex = (currentPlayerIndex + 1) % playersGame.size();
    }

    public String getWinner() {
        int topScore = Integer.MIN_VALUE;
        int topPlayerIndex = 0;
        for (int i =0; i < playersGame.size(); i++)
            if (topScore < playersGame.get(i).score()){
                topPlayerIndex = i;
                topScore = playersGame.get(i).score();
```

```
44              }
        return playersName.get(topPlayerIndex);
46      }
}
```

Come si può vedere la classe dipende solo dall'interfaccia `BowlingGameInterface`.
In questo modo quando la classe andrà in produzione potrà utilizzare indipendentemente qualsiasi realizzazione di `BowlingGameInterface` (quindi anche la classe `BowlingGame`). Se viene eseguito nuovamente il test, questo verrà superato. Grazie all'utilizzo della tecnica *Mock Object* lo sviluppatore potrà realizzare e verificare il comportamento della classe `MultiplayerBowlingGame` senza dipendere dalla classe `BowlingGame`.

Tipicamente questa tecnica viene utilizzata per verificare il comportamento delle unità che dipendono da classi che sono in stretto contatto con componenti *hardware*. In questo modo, realizzando i metodi esposti dall'interfaccia della classe in contatto con il componente *hardware* si possono riprodurre le condizioni d'errore che potrebbero essere riprodotte dal componente reale.

Realizzazione rapida dei *Mock Object*

Utilizzare *Mock Object* per verificare il comportamento delle unità di un programma è una cosa molto utile, ma ha lo svantaggio di aumentare la quantità del codice di test.

Esistono delle librerie che permettono di realizzare in modo semplice e rapido *Mock Object*. Una delle più note e semplici librerie per realizzare rapidamente *Mock Object* in *Java* è *Easy-Mock*[5]. *Easy-Mock* mette a disposizione un modo interessante per creare e simulare metodi specificati in un'interfaccia. La creazione dell'oggetto *mock* può avvenire direttamente nella classe di test. Per creare un nuovo oggetto *mock* è necessario:

1. creare un oggetto *mock* per l'interfaccia che si vuole simulare (senza realizzare la classe)
2. registrare il comportamento desiderato
3. cambiare lo stato in "*ready*" e utilizzare l'oggetto per il test

Questi tre passi vengono realizzati direttamente nel codice di test senza la necessità di creare la classe del oggetto *mock*.

Esempio di utilizzo della libreria *Easy-Mock*

Nell'esempio del gioco del bowling, se si vuole creare e verificare il comportamento di un metodo che calcoli il punteggio del vincitore di una partita *multiplayer* di bowling è possibile realizzare il seguente test:

Listing 4.29: src/test/java/it/unipd/app/MultiplayerBowlingGameTest.java

```
1  package it.unipd.app;

3  import junit.framework.TestCase;
   import static org.easymock.EasyMock.*;
5
   public class MultiplayerBowlingGameTest extends TestCase {
7
       public MultiplayerBowlingGameTest(String name) {
9          super(name);
       }
11
       public void testGetWinnerScore() {
13         // create player1
           BowlingGameInterface mockPlayer1 =
15             createMock(BowlingGameInterface.class);
           expect(mockPlayer1.score()).andReturn(30).atLeastOnce();
17         replay(mockPlayer1);

19         MultiplayerBowlingGame mpGame = new MultiplayerBowlingGame();
           mpGame.addPlayer("player1", mockPlayer1);
21         assertEquals(30, mpGame.getWinnerScore());
           verify(mockPlayer1);
23     }
...
```

[5] http://www.easymock.org/

Il codice del metodo `testGetWinnerScore` è stato realizzato nel seguente modo:

- Riga 4: vengono importati i metodi statici della classe `org.easymock.EasyMock`.
- Righe 14-15: viene creato un oggetto *mock*, tramite il metodo `createMock`, che realizza l'interfaccia `BowligGameInterface`.
- Riga 16: viene registrato il comportamento del metodo `BowlingGameInterface.score`. In questa riga viene specificato che il metodo deve essere chiamato almeno una volta prima della terminazione del test e che ritornerà il risultato 30.
- Riga 17: viene impostato l'oggetto `mockPlayer1` nella modalità di risposta. D'ora in poi se viene invocato il metodo `score` esso ritornerà il valore 30. Se vengono invocati gli altri metodi, che non sono stati registrati, verrà lanciata un'eccezione.
- Righe 18-21: viene creato il test che consiste nel creare un istanza di `MultiplayerBowlingGame` dove è presente solo un giocatore che effettua la partita. Viene verificato quindi che il metodo `getWinnerScore` ritorni il valore registrato nell'oggetto *mock*.
- Riga 22: viene invocato il metodo `verify` della classe `EasyMock`. Questo metodo permette di verificare che il metodo `score` dell'oggetto `mockPlayer` sia stato invocato nella modalità in cui è stato registrato (in questo caso controlla che il metodo sia stato invocato almeno una volta).

A questo punto lo sviluppatore potrà creare il metodo `getWinnerScore` della classe `MultiplaerBowlingGame` in modo da far superare il nuovo test.

Come si è potuto vedere dall'esempio, *Easy-Mock* è una libreria potente e semplice da utilizzare. Fornisce molte funzionalità che possono essere utili durante la fase di test. Dopo la registrazione dell'oggetto *mock* permette di utilizzarlo per creare il test e alla fine dell'esecuzione permette di verificare se l'oggetto si è comportato come desiderato.

Considerazioni sui *Mock Object*

Come si evince dagli esempi, utilizzare la tecnica di *Mock Object* ha i seguenti vantaggi:

- Rende i test di unità indipendenti e semplici. In questo modo si riesce ad identificare molto velocemente quale è l'unità affetta dall'errore
- Permette di ricreare condizioni difficili o scomode da riprodurre
- Permette di creare test per le unità che dipendono da componenti non ancora esistenti
- Promuove l'utilizzo di interfacce

Usare questa tecnica porta i seguenti svantaggi:

- Aumentano le dimensioni del codice di test
- Comporta una maggior manutenzione per i test. I test di unità, utilizzando questa tecnica, sono più specifici, quindi se viene cambiato il codice di produzione con molto probabilmente devono essere modificati anche gli oggetti *mock*.
- Non è sempre possibile adottare questa tecnica: a volte non è possibile creare un'interfaccia che descrive il comportamento di alcune componenti

È lo sviluppatore che deve capire quando è utile utilizzare questa tecnica per rendere i test indipendenti. Esistono altri strumenti, specifici per la tipologia di applicazione che si sta sviluppando, che permettono di rendere i test indipendenti. Per esempio, per testare applicazioni che utilizzano *database*, è possibile utilizzare *database* sviluppati in *Java* (p.es *HSQL* e *Derby*) che permettono di verificare il comportamento dell'applicazione velocemente.

Capitolo 5

Automatizzare il processo di *build*

Il processo di *build* permetti di creare un prodotto *software* a partire dai sorgenti. Questo processo viene eseguito ogni volta che si vuole creare e verificare il prodotto che si sta realizzando. Visto che è un attività ripetitiva, se viene fatta ogni volta manualmente, possono essere commessi degli errori. In questo capitolo viene descritto come automatizzare questo processo, e creare un'automazione a comando che permette la creazione di un prodotto *software* in modo automatico.

5.1 Definizioni utili

Il processo di *build* è un insieme di passi che trasformano gli *script* di *build*, il codice sorgente, i file di configurazione, la documentazione e i test in un prodotto *software* distribuibile.

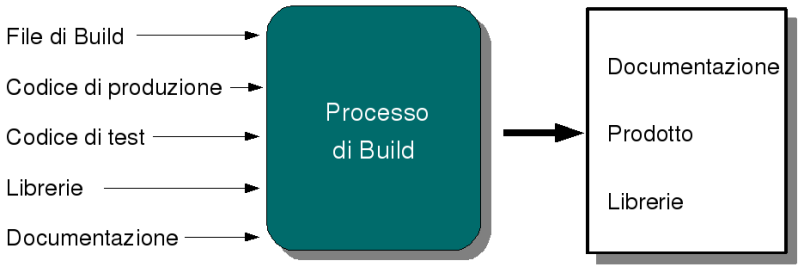

Figura 5.1: Processo di *build*

Grazie ad un'automazione a comando, che permette di realizzare il processo di *build*, è possibile creare e verificare velocemente il comportamento di un prodotto *software* distribuibile. In questo modo, ogni volta che si sono fatti dei cambiamenti al codice sorgente, è possibile verificare e creare un nuovo prodotto *software* che contenga questi cambiamenti.

Il processo di *build* deve avere le seguenti caratteristiche, riportate in [11], che possono essere riassunte dall'acronimo *CRISP*:

- ***Complete***: Indipendente da fonti non specificate nello *script* di *build*. Il processo di *build* deve essere autosufficiente quindi tutte le risorse necessarie per realizzare il prodotto devono essere specificate e accessibili dallo *script* che realizza l'automazione;
- ***Repeatable***: Gli *script* di *build* devono essere salvati nel sistema di gestione del codice sorgente, con tutte le altre risorse necessarie ad effettuare il processo di *build*. In questo modo è possibile ricostruire qualsiasi versione del progetto sviluppato e ottenere in ogni momento sempre gli stessi risultati;
- ***Informative***: Fornisce informazioni sullo stato del prodotto. Il processo di *build* deve sempre eseguire tutti i test di unità presenti nel progetto in modo da verificare che i cambiamenti apportati in una revisione non abbiano introdotto vecchi errori. I test devono produrre delle reportistiche che possono essere consultate in ogni momento dagli sviluppatori;

- ***Schedulable***: Se il processo di *build* è completo e ripetibile può essere schedulato (da uno schedulatore) a determinate ore del giorno (p.es. dopo l'orario lavorativo per verificare che i cambiamenti apportati dal *team* di sviluppo non facciano fallire il processo di *build*) o eseguito quando avvengono determinati eventi (p.es. ogni volta che viene fatta un attività di *committing* del codice sorgente);

- ***Portable***: Indipendente il più possibile dall'ambiente di esecuzione. Se vengono scaricate, dal sistema di versionamento, tutte le risorse necessarie per effettuare una *build*, deve essere possibile eseguire il processo di *build* (sia nell'ambiente di produzione che di lavoro);

Prima di descrivere come realizzare un'automazione a comando, per realizzare il processo di *build*, viene descritta la struttura delle *directory* adotta per realizzare il progetto BowlingScore.

5.2 Struttura delle *directory* di un progetto

Per realizzare un'automazione a comando, che realizzi il processo di *build*, è necessario definire le *directory* che conterranno gli elementi di *input* e dove verranno salvati gli elementi di *output* prodotti dal processo.

Directory di *input*

La struttura delle *directory* scelta per gli elementi di *input* è quella adottata da *Maven*[1].

```
src
|-- main
|   `-- java
|-- test
|   |-- java
|   `-- resources
vendor
`-- lib
    |-- prod
    |-- test
    `-- report
```

Le *directory* `src/main/` e `src/test` contengono i sorgenti di *input* per il progetto. Tutti i file sorgenti *Java* del progetto sono contenuti nella *directory* `src/main/java`. I file sorgenti *Java* dei test di unità sono contenuti nella *directory* `src/test/java`. In entrambe le *directory*, che contengono i sorgenti, viene replicata la struttura dei *package Java* del progetto, questo per permettere che le classi di test siano nello stesso *package* delle classi di produzione. In questo modo i metodi di test possono accedere ai metodi protetti del codice in produzione ed è molto più semplice mantenere separate le due parti di codice (vedi sezione 3.4).

La cartella `src/test/resouces` contiene i file di configurazione utilizzati dai test (p.es. i file che contengono i dati dei test). Tutto il contenuto della cartella `src` deve essere sotto il controllo del sistema di versionamento.

Le cartelle `vendor/lib/prod`, `vendor/lib/test` e `vendor/lib/report` contengono tutte le librerie che sono usate dal codice di produzione, dal codice di test e dagli strumenti di reportistica. Poiché il codice sorgente di un progetto potrebbe dipendere da specifiche versioni di alcune librerie, è necessario mantenere anche queste sotto il controllo del sistema di versionamento. In questo modo il processo di *build* viene reso completo e ripetibile.

Directory di *output*

Quando viene eseguito il processo di *build*, vengono compilati tutti i sorgenti *Java*, sia di produzione che di test.

```
build
|-- prod
|-- reports
`-- test
```

[1] `http://maven.apache.org/`

I file prodotti dalla compilazione vengono salvati nella *directory* `build/prod` e `build/test`. Anche queste *directory* replicano la struttura dei *package* *Java* dei file sorgenti. Separare l'*output* in due *directory* permette di distinguere i file (.class) di produzione dai file di test.

I file di reportistica verranno creati e salvati nella cartella `build/reports`.

La *directory* `build` conterrà solo file che possono essere generati dagli elementi di *input*. Poiché gli elementi di *input* vengono gestiti dal sistema di versionamento, è possibile rigenerare ogni volta i file di *output* e quindi non è necessario gestire la *directory* `build` attraverso il sistema di versionamento.

Il primo processo di *build*

Di seguito viene descritto come realizzare il processo di *build* del progetto BowlingScore. Il primo processo di *build* verrà realizzato tramite dei comandi eseguiti da un terminale *Unix*.

Per realizzare il primo processo di *build* è necessario posizionarsi nella *directory* contenente il progetto.

```
cd home/nicola/work/bowlingscore
```

Poiché la *directory* `build/prod` non è sotto il controllo del sistema di versionamento è necessario crearla.

```
mkdir -p build/prod
```

Compilare i sorgenti usando il compilatore *javac*.

```
javac -d build/prod src/main/java/it/unipd/app/*.java
```

Il comando *javac* permette di compilare i sorgenti *Java*, l'opzione `-d` permette di specificare la destinazione dove vengono salvati i file .class prodotti dalla compilazione. Di seguito vengono specificati i file che devono essere compilati.

Se per caso nel codice del progetto viene usata qualche libreria, è possibile specificarla tramite l'opzione `-classpath` separando il *path* di ogni libreria con il carattere **:** in *Unix* e con **;** in *Windows*.

Dopo che questo comando è stato eseguito, la *directory* `build/prod` conterrà i file prodotti dalla compilazione.

```
ls -R build/prod
```

In questo modo è stato possibile realizzare il primo processo di *build*.

Come si può osservare questo processo di *build* non è ripetibile. Infatti se un componente del *team* di sviluppo vuole eseguire nuovamente il processo di *build* deve ricordarsi tutti i comandi, e ripetere ogni volta tutti i passi appena descritti. Poiché il processo di *build* deve essere eseguito molte volte, risulta scomodo ripetere ogni volta tutti questi passi manualmente (aumentando il rischio di commettere errori).

Questo processo di *build* non è portatile. Se deve essere eseguito in diversi sistemi operativi devono essere modificati alcuni comandi.

Per rendere il processo di *build* ripetibile e portatile si potrebbero creare due file: Un file *batch* con i comandi di *Windows* e un file *shell script* con i comandi *Unix*. In questo modo vengono create delle automazioni a comando che permettono di ripetere, sempre allo stesso modo, una serie di attività. Così facendo si andrebbe contro il principio *DRY* e quindi risulterebbe più difficile mantenere consistente l'informazione contenuta in questi due file.

Di seguito vengono descritti due strumenti specifici per realizzare il processo di *build* per progetti *Java*. Questi strumenti sono *Ant* e *Maven*.

5.3 Automazione con *Ant*

I file che possono essere interpretati ed eseguiti da *Ant* sono espressi tramite il linguaggio *XML*[2].

Per proseguire con i successivi esempi è necessario installare *Ant* come viene descritto al seguente indirizzo: `http://ant.apache.org/manual/index.html`

Di seguito verrà creato un file di *build* eseguibile da *Ant* che effettuerà le attività descritte nella sezione precedente. Per convenzione *Ant*, quando viene eseguito (attraverso il comando `ant`), cerca nella *directory* corrente, un file chiamato `build.xml`. Il file `build.xml` è il file che permette di specificare le attività del processo di *build* che il comando `ant` deve eseguire.

[2]Extensible Markup Language `http://www.w3.org/XML/`

5.3.1 Definizione del progetto

Per realizzare il processo di *build* attraverso *Ant* è necessario creare un file chiamato `build.xml` che verrà salvato nella *directory* principale del progetto Bowilng Score. In questo file viene definito il progetto che *Ant* dovrà eseguire.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="bowlingscore" default="compile" basedir=".">
</project>
```

- Nella prima riga viene specificato che il contenuto del file è scritto in linguaggio *XML*
- Nella seconda riga viene definito il progetto, tramite l'elemento `project`. In questo elemento vengono specificati:
  - L'attributo `name` che specifica il nome del progetto
  - L'attributo `default` che specifica l'attività che viene eseguita quando viene invocato il comando `ant` dalla linea di comando. In questo caso verranno eseguiti i passi specificate nell'attività `compile` del progetto
  - L'attributo `basedir`: specifica che ogni percorso presente nel file deve essere relativo rispetto alla locazione specificata da questo attributo (in questo caso la *directory* del progetto BowlingScore)

5.3.2 Definizione degli elementi

Per eseguire il processo di *build* devono essere specificati gli elementi che permettono di realizzare il prodotto. Deve essere specificata la struttura delle *directory* di *input* e di *output*.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="bowlingscore" default="compile" basedir=".">

  <property name="build.dir" location="build"/>
  <property name="build.prod.dir" location="${build.dir}/prod"/>
  <property name="build.test.dir" location="${build.dir}/test"/>
  <property name="src.dir" location="src"/>
  <property name="src.main.java.dir" location="${src.dir}/main/java"/>
  <property name="src.test.java.dir" location="${src.dir}/test/java"/>
  <property name="vendor.lib.dir" location="vendor/lib"/>
  <property name="vendor.lib.prod.dir" location="${vendor.lib.dir}/prod"/>
  <property name="vendor.lib.test.dir" location="${vendor.lib.dir}/test"/>

</project>
```

Ogni elemento `property` permette di definire delle variabili che si riferiscono ad una locazione. Il nome della variabile viene definito con l'attributo `name`, la locazione della variabile viene definita con l'attributo `location`. Ogni locazione è relativa alla *directory* definita dall'attributo `basedir` dell'elemento `project`. Definire le locazioni degli elementi, che compongono il processo di *build*, attraverso delle variabili ha due benefici:

1. Rende il processo di *build* indipendente. Lo sviluppatore può scaricare tutti gli elementi dal sistema di versionamento ed eseguire il processo di *build* da qualsiasi *directory*
2. Rende più semplice la manutenzione del processo di *build*. Se viene modificata la struttura delle *directory* del progetto basta modificare le locazioni delle variabili espresse dagli attributi degli elementi `property`

5.3.3 Definizione del `classpath`

Per compilare i sorgenti è necessario definire il `classpath`.

```
  <path id="prod.classpath">
    <pathelement location="${build.prod.dir}"/>
    <fileset dir="${vendor.lib.prod.dir}">
      <include name="*.jar"/>
    </fileset>
  </path>
```

L'elemento `path` viene definito all'interno del elemento `project`. In questo esempio viene specificato il `classpath` del codice di produzione.

Nella prima riga viene definito un elemento `path`. L'attributo `id` permette di specificare un Identificativo univoco che potrà essere utilizzato per riferirsi al `classpath` in vari punti di questo file.

L'elemento `pathelement` permette di specificare la locazione dove sono contenuti i file .class che verranno utilizzati durante la compilazione dei file *Java*.

L'elemento `fileset` permette di specificare un insieme di file contenuti all'interno di una determinata *directory*. In questo esempio vengono incluse nel `classpath` tutte le librerie contenute nella *directory* specificata dalla variabile `vendor.lib.prod.dir`. In questo modo, se il codice di produzione dovrà utilizzare in futuro una libreria, basterà inserirla nella *directory* `vendor/lib/prod/` e questa verrà inserita automaticamente nel `classpath`.

Grazie a questo elemento il processo di *build* risulta essere completo, visto che dipenderà solo dagli elementi specificati in questo file.

5.3.4 Preparazione delle *directory* di *output*

Prima di poter compilare i sorgenti del progetto è necessario definire nello *script*, un'attività che permette di creare le *directory* che conterranno i risultati del processo.

```
<target name="prepare">
  <mkdir dir="${build.prod.dir}"/>
  <mkdir dir="${build.test.dir}"/>
</target>
```

L'elemento `target`, deve essere specificato all'interno del elemento `project` e permette di specificare un attività che potrà essere svolta da *Ant*. In questo elemento viene specificato un nome (tramite l'attributo `name`) che verrà utilizzato come riferimento per invocare l'attività.

L'elemento `mkdir` permette di creare una cartella. In questo caso, tramite l'attributo `dir` viene specificato di creare le cartelle che conterranno i compilati dei file di produzione e di test.

A questo punto è possibile far eseguire questa attività ad *Ant*. Per far ciò, è necessario posizionarsi nella cartella del progetto.

```
cd home/nicola/work/bowlingscore
```

ed eseguire l'attività `prepare`:

```
ant prepare
```

Dopo aver eseguito il comando, verranno create le cartelle `build/prod` e `build/test`.

5.3.5 Compilazione dei file di produzione

Per compilare i file sorgenti è necessario creare una nuova attività.

La nuova attività deve permettere di compilare tutti i sorgenti di produzione (contenuti nella cartella specificata dalla variabile `src.main.java.dir`), utilizzando il `classpath` specificato precedentemente, e salvare i file .class nella *directory* di *output* (specificata dalla variabile `build.prod.dir`).

```
<target name="compile" depends="prepare">
  <javac srcdir="${src.main.java.dir}" destdir="${build.prod.dir}">
        <classpath refid="prod.classpath" />
  </javac>
</target >
```

L'ordine con cui viene eseguita questa attività è molto importante. Infatti può essere eseguita solo dopo che sono state creare le *directory* di *output*. L'ordine di esecuzione viene specificato dall'attributo `depends` dell'elemento `target`.

- Nella prima riga, tramite l'attributo `depends`, viene specificato che prima di eseguire l'attività `compile`, deve essere eseguita l'attività `prepare`. Nell'attributo `depends` possono essere specificate più attività separate da una virgola.

- Nella seconda riga, grazie all'elemento `javac` vengono specificati i sorgenti che devono essere compilati (tramite l'attributo `srcdir`) e la destinazione dove verranno salvati i file creati durante la compilazione (tramite l'attributo `destdir`). Questo elemento permette di compilare solo i file che sono stati modificati dall'ultima compilazione.

- Nella terza riga, l'elemento `classpath` permette di specificare il riferimento al path definito precedentemente (tramite l'attributo `refid`).

A questo punto è stato riprodotto, tramite un file che può essere eseguito da *Ant*, il processo di *build* descritto nella sezione precedente. Il file risultante è il seguente

Listing 5.1: /home/nicola/work/bowlingscore/build.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
   <project name="bowlingscore" default="compile" basedir=".">
3
     <property name="build.dir" location="build"/>
5    <property name="build.prod.dir" location="${build.dir}/prod"/>
     <property name="build.test.dir" location="${build.dir}/test"/>
7    <property name="src.dir" location="src"/>
     <property name="src.main.java.dir" location="${src.dir}/main/java"/>
9    <property name="src.test.java.dir" location="${src.dir}/test/java"/>
     <property name="vendor.lib.dir" location="vendor/lib"/>
11   <property name="vendor.lib.prod.dir" location="${vendor.lib.dir}/prod"/>
     <property name="vendor.lib.test.dir" location="${vendor.lib.dir}/test"/>
13
     <path id="prod.classpath">
15     <pathelement location="${build.prod.dir}"/>
       <fileset dir="${vendor.lib.prod.dir}">
17       <include name="*.jar"/>
       </fileset>
19   </path>

21   <target name="prepare">
       <mkdir dir="${build.prod.dir}"/>
23     <mkdir dir="${build.test.dir}"/>
     </target>
25
     <target name="compile" depends="prepare">
27     <javac srcdir="${src.main.java.dir}" destdir="${build.prod.dir}">
             <classpath refid="prod.classpath" />
29     </javac>
     </target >
31
   </project>
```

Prima di eseguire e verificare il processo di *build* è consigliato inserire questo file nel sistema di versionamento. Per eseguire il processo di *build* si devono eseguire i seguenti comandi.

```
cd home/nicola/work/bowlingscore
ant
```

A questo punto, se tutto è andato a buon fine, verranno prodotti i seguenti *output*.

```
Buildfile: build.xml

prepare:
    [mkdir] Created dir: /home/nicola/work/bowlingscore/build/prod
    [mkdir] Created dir: /home/nicola/work/bowlingscore/build/test

compile:
    [javac] Compiling 7 source files to /home/nicola/work/bowlingscore/build/prod

BUILD SUCCESSFUL
Total time: 1 second
```

In questo modo *Ant* notifica che ha letto il file `build.xml` ed ha eseguito le attività `prepare` e `compile` (attività principale del progetto che dipende da prepare). Notifica lo stato del processo di *build* e il tempo impiegato per l'esecuzione del processo.

È possibile anche eseguire solo una specifica attività contenuta in uno specifico file

```
ant -f build.xml prepare
```

In questo modo viene eseguita solo l'attività `prepare` del file `build.xml`.

In questa sezione è stata creata un'automazione a comando che esegue un processo di *build* in modo completo, informativo, ripetibile, schedulabile e portatile.

Come si può notare l'aspetto informativo non è stato approfondito, infatti viene eseguita solo la compilazione dei sorgenti e non vengono eseguiti i test. Di seguito viene spiegato come migliorare questo aspetto aumentando le informazioni prodotte dal processo di *build*, rendendo automatica l'esecuzione e la pubblicazione dei risultati dei test di unità.

5.3.6 Compilazione ed esecuzione dei test

Prima di poter eseguire i test di unità in modo automatico, ad ogni processo di *build*, è necessario specificare il `classpath` dei test e compilare i test.

Definizione del `classpath` dei test

Per specificare il `classpath` dei test è necessario creare un altro elemento `path` all'interno dell'elemento `project` del file `build.xml`.

```
<path id="test.classpath">
  <path refid="prod.classpath"/>
  <pathelement location="${build.test.dir}"/>
  <fileset dir="${vendor.lib.test.dir}">
    <include name="*.jar"/>
  </fileset>
</path>
```

- Nella prima riga viene specificato un nuovo elemento `path` con attributo `id` settato a `test.classpath`. In questo modo, nella compilazione dei test è possibile riferirsi a questo elemento specificando solo il suo riferimento. È stato necessario dividere il `classpath` dei test da quello di produzione, perché i test utilizzano delle librerie (p.es. *JUnit*) che non vengono utilizzate dal codice di produzione
- Nella seconda riga viene specificato, tramite un altro elemento `path` che il `classpath` dei test deve contenere il `classpath` del codice di produzione. Questo perché il codice di test si riferisce alle classi di produzione. È stato utilizzato il riferimento, e non è stato duplicato il codice per non andare contro al principio *DRY*, in questo modo se vengono aggiunti o tolti elementi dal `classpath` di produzione, non è necessario modificare il `classpath` di test
- Nella terza riga vengono specificati le classi di test
- Nella quarta riga vengono inserite le librerie che vengono utilizzate dal codice di test

Compilazione dei test

Per compilare i file di test è necessario aggiungere al file una nuova attività.

```
<target name="compile-tests" depends="compile">
  <javac srcdir="${src.test.java.dir}" destdir="${build.test.dir}">
    <classpath refid="test.classpath" />
  </javac>
</target>
```

Questa attività è molto simile all'attività `compile` precedentemente descritta.

- Nella prima riga viene specificato un elemento `target` che definisce l'attività di compilazione di test. Come si può notare essa dipende dall'attività `compile` (infatti i test hanno bisogno di utilizzare le classi generate dai sorgenti di produzione).
- Nella seconda riga viene specificato un elemento `javac` che permette di compilare i file di test e salvarli nella *directory* specificata dalla variabile `build.test.dir`.

- Nella terza riga viene definito il `classpath` per compilare i test. Come si può vedere esso si riferisce al path precedentemente creato.

Se viene eseguito l'attività `compile-tests` è possibile verificare che siano prodotti i file .class dei test nella *directory* `build/test`.

```
ant compile-tests
```

Dopo l'esecuzione dell'attività, *Ant* produce il seguente risultato.

```
Buildfile: build.xml

prepare:

compile:

compile-tests:
    [javac] Compiling 11 source files to /home/nicola/work/bowlingscore/build/test

BUILD SUCCESSFUL
Total time: 1 second
```

Come si può notare, poiché le attività `prepare` e `compile` erano state eseguite precedentemente, *Ant* effettua solo l'attività `compie-tests` che esegue con successo. Se viene visitata la *directory* `build/test` verranno trovati i file .class generati.

Esecuzione dei test

Per eseguire e notificare il risultato dei test in modo automatico è necessario creare una nuova attività. Poiché i test utilizzando delle risorse, contenute nella *directory* `src/test/resources`, è necessario copiare tutti i file contenuti in questa *directory* nella *directory* `build/test`. In questo modo verranno automaticamente aggiunti questi file al `classpath`, e i test potranno accedere alle risorse. Per far questo è necessario aggiungere un'attività e una proprietà al file `build.xml`.

```
1 <property name="test.resources.dir" location="${src.dir}/test/resources"/>

3 <target name="copy-test-resources">
   <copy todir="${build.test.dir}">
5    <fileset dir="${test.resources.dir}">
       <include name="**/*.*"/>
7    </fileset>
   </copy>
9 </target>
```

Nella prima riga viene aggiunta una proprietà che permette di specificare la locazione delle risorse utilizzate dai test.

Successivamente viene definita l'attività `copy-test-resources` che permette di copiare tutte le risorse nella *directory* di *build* dei test.

Di seguito viene definita l'attività che permette di eseguire i test di unità del progetto.

```
1 <target name="test" depends="compile-tests,␣copy-test-resources">
    <junit haltonfailure="true">
3     <classpath refid="test.classpath" />
      <formatter type="brief" usefile="false" />
5     <batchtest>
        <fileset dir="${build.test.dir}" includes="**/*Test.class" />
7     </batchtest>
    </junit>
9 </target>
```

- Nella prima riga viene specificato che l'attività di esecuzione di test dipende dalle attività di compilazione dei test e di ricopiamento delle risorse.

- Nella seconda riga viene utilizzato l'elemento `junit`. Questo elemento permette di eseguire i test di unità specificati. L'attributo `haltonfailture`, che è settato a vero, permette di specificare di far fallire il processo di *build* se un test fallisce.

- Nella terza riga viene specificato il `classpath` per l'esecuzione dei test.
- Nella quarta riga viene specificato il formato con cui devono essere pubblicati i risultati dei test. In questo esempio si è scelto di utilizza la modalità `brief`, che permette di visualizzare informazioni dettagliate solo se il test fallisce. In oltre viene settato l'attributo `usefile` a falso. In questo modo i risultati vengono solo mostrati a video.
- Nelle righe 5-7 viene definita la *suite* di test. In questo esempio vengono eseguiti tutti i file contenuti nella *directory* definita dalla variabile `build.test.dir` (`build/test`) che hanno il nome che termina con la parola "Test".

Se viene eseguita l'attività `test` vengono eseguiti tutti i test. Le dipendenze delle attività sono espresse dalla figura 5.2 realizzata da *Grand*[3].

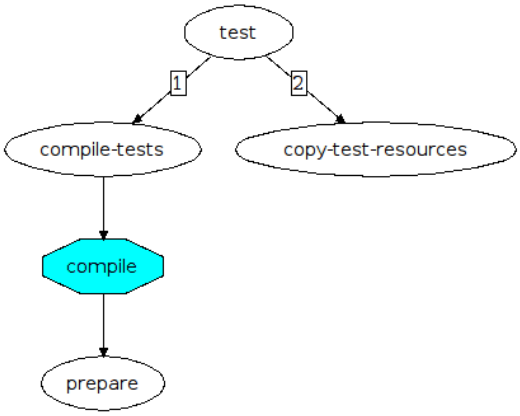

Figura 5.2: Dipendenze del processo di *build*

Prima di provare ad eseguire il processo di *build* è consigliato salvare il file nel sistema di versionamento. Se viene eseguita l'attività di test tramite il comando

```
cd home/nicola/work/bowlingscore
ant test
```

verrà riportata la seguente notifica

```
Buildfile: build.xml

prepare:

compile:

compile-tests:

test:
    [junit] Testsuite: it.unipd.app.AppTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,002 sec
    [junit]
    [junit] Testsuite: it.unipd.app.BowlingGameTest
    [junit] Tests run: 9, Failures: 0, Errors: 0, Time elapsed: 0,005 sec
    [junit]
    ...
    [junit]

BUILD SUCCESSFUL
Total time: 0 seconds
```

La notifica riporta l'*output* prodotto dalle attività eseguite. In questo caso *Ant* ha effettuato la compilazione di tutti i sorgenti e di tutti i test ed ha eseguito tutti i test specificati pubblicando i risultati. In questo modo è stata creata un'automazione a comando che riporta maggiori informazioni rispetto alla precedente.

[3] `http://www.ggtools.net/grand`

5.3.7 *Checkstyle*: Verifica del rispetto di convenzioni di stile

Checkstyle è uno strumento molto simile a *PMD*. Inizialmente è stato creato per verificare il rispetto di convenzioni di stili durante la fase di codifica. Recentemente sono state aggiunte ulteriori regole che permettono di trovare delle imperfezioni o degli errori ricorrenti presenti nel codice sorgente. Uno sviluppatore può usare *Checkstyle* nel seguente modo:

1. Utilizzare *Checkstyle* per verificare il rispetto di convenzioni di stile, ed utilizzare altri strumenti per trovare altri problemi nel codice
2. Utilizzare *Checkstyle* per verificare il rispetto di convenzioni di stile e per trovare altri problemi nel codice, e utilizzare anche altri strumenti di verifica statica
3. Utilizzare esclusivamente *Checkstyle* per verificare il rispetto di convenzioni di stile e i problemi nel codice sorgente

In questa sezione viene utilizzato *Checkstyle* nel primo modo. È consigliato consultare il seguente indirizzo `http://checkstyle.sourceforge.net/availablechecks.html` per capire tutti i possibili controlli che si possono fare utilizzando questo strumento.

Per controllare che il progetto implementato rispetti le convenzioni di stile di *Sun Java* è necessario scaricare e installare *Checkstyle* e inserire una nuova attività nel processo di *build* del progetto.

Installazione di *Checkstyle*

Per utilizzare *Checkstyle* in un progetto gestito da *Ant* è necessario scaricare l'archivio contenente questo strumento da `http://checkstyle.sourceforge.net/` (in questa guida viene utilizzato *Checkstyle* versione 4.4).

Dopo aver estratto l'archivio è necessario creare la cartella `bowlingscore/vendor/lib/report/checkstyle` e copiare al suo interno i seguenti file contenuti nell'archivio di *Checkstyle*:

- `checkstyle-4.4/checkstyle-all-4.4.jar` : Archivio contenente le classi di utilità e le librerie necessarie per utilizzare *Checkstyle* attraverso *Ant*;
- `checkstyle-4.4/sun_checks.xml` : File di configurazione dove vengono specificate le regole per controllare il rispetto delle convenzioni di stile di *Sun Java* attraverso *Checkstyle*, reperibili al indirizzo `http://java.sun.com/docs/codeconv/`;
- `checkstyle-4.4/contrib/checkstyle-author.xsl` : File che permette di trasformare i risultati prodotti da *Checkstyle* in pagine *HTML*;

La cartella quindi dovrà contenere i seguenti file:

```
vendor/lib/report/checkstyle/
|-- checkstyle-all-4.4.jar
|-- checkstyle-author.xsl
`-- sun_checks.xml
```

Questa cartella dovrà essere gestita dal sistema di versionamento, in questo modo sarà possibile effettuare la verifica delle convenzioni di stile sia dall'ambiente di lavoro che dall'ambiente di *build*.

Utilizzo di *Checkstyle* nel processo di *build*

Per verificare e identificare le infrazioni delle convenzioni di stile in un progetto gestito da *Ant* è necessario inserire nel file `build.xml` del progetto le seguenti proprietà e attività:

```
1  <property name="checkstyle.report.dir" location="${reports}/checkstyle-results"/>
   <property name="vendor.lib.checkstyle.dir" location="vendor/lib/report/checkstyle"/>
3
   <taskdef resource="checkstyletask.properties"
5                classpath="${vendor.lib.checkstyle.dir}/checkstyle-all-4.4.jar"/>

7  <target name="checkstyle"
```

```
    description="Generates a report of code convention violations.">
9       <delete dir="${checkstyle.report.dir}" />
        <mkdir dir="${checkstyle.report.dir}"/>
11    <checkstyle config="${vendor.lib.checkstyle.dir}/sun_checks.xml"
          failOnViolation="false">
13      <formatter type="xml" tofile="${checkstyle.report.dir}/checkstyle_report.xml"/>
        <fileset dir= "${src.main.java.dir}" includes="**/*.java"/>
15    </checkstyle>
        <xslt style="${vendor.lib.checkstyle.dir}/checkstyle-author.xsl"
17      in="${checkstyle.report.dir}/checkstyle_report.xml"
        out="${checkstyle.report.dir}/checkstyle_report.html" />
19  </target>
```

- Riga 1: viene definita la variabile `checkstyle.report.dir` che identifica la *directory* che conterrà le reportistiche prodotte da questo strumento;
- Riga 2: viene definita la variabile `vendor.lib.checkstyle.dir` che identifica la *directory* che contiene i file di configurazione e le librerie necessarie per utilizzare *Checkstyle*;
- Riga 4: vengono definite le attività necessarie ad utilizzare *Checkstyle* attraverso *Ant*;
- Riga 7-19: viene definita l'attività `checkstyle`. Ogni volta che viene eseguita questa attività viene eliminata e ricreata la *directory* specificata dalla variabile `checkstyle.report.dir` (righe 9-10). Viene eseguito *Checkstyle* configurato in modo da controllare se il codice del progetto rispetti le convenzioni di stile di *Sun Java* (riga 11);
- Riga 12: viene specificato che, se vengono trovate delle infrazioni delle convenzioni di stile, questo strumento non deve fare fallire il processo di *build*;
- Riga 13: viene specificato che le infrazioni trovate da questo strumento devono essere riportate in un file *XML*
- Riga 14: vengono specificati tramite l'elemento `fileset` i sorgenti dei file da analizzare;
- Righe 16-18: viene specificato che le reportistiche prodotte da questo strumento vengano trasformate attraverso un trasformatore *XSL* e salvate in un file *HTML*;

Per maggiori informazioni riguardanti la configurazione di questo strumento è consigliato consultare `http://checkstyle.sourceforge.net/anttask.html`. Grazie a questa configurazione se viene eseguita l'attività:

```
ant checkstyle
```

viene analizzato il codice sorgente di produzione e prodotte delle reportistiche (salvate nella *directory* specificata dall'elemento `checkstyle.report.dir`) dove vengono riportate le infrazioni trovate da questo strumento.

5.3.8 *Cobertura*: Misurazione della percentuale di codice verificato

Cobertura è uno strumento che permette di misurare la carenza di test di unità all'interno di un progetto. Per ogni file del codice di produzione del progetto, *Cobertura* permette di calcolare le seguenti informazioni:

- La percentuale di linee di codice che vengono controllate dai test di unità
- La percentuale di diramazioni del codice che vengono controllate dai test di unità
- La complessità cicolmatica di ogni classe
- Il numero di volte che una linea di codice è stata eseugita dai test di unità

Le metriche prodotte da questo strumento sono essenziali per capire le parti di un progetto carenti di test di unità ed individuare le classi più complesse da mantenere.

Per utilizzare questo strumento attraverso *Ant* è necessario scaricare e installare le librerie necessarie per utilizzare *Cobertura* e inserire delle nuove attività nel processo di *build* del progetto.

Installazione di *Cobertura*

Per utilizzare *Cobertura* in un progetto gestito da *Ant* è necessario scaricare l'archivio contenente i file necessari ad utilizzare questo strumento da `http://cobertura.sourceforge.net/download.html` (in questa guida viene utilizzata la versione 1.9-bin).

Dopo aver estratto l'archivio è necessario creare la cartella `bowlingscore/vendor/lib/report/cobertura/` e copiare al suo interno i seguenti file contenuti nell'archivio di *Cobertura*:

- `cobertura-1.9/lib/*`: Insieme di librerie necessarie per utilizzare *Cobertura*;
- `cobertura-1.9/cobertura.jar`: Archivio contenente le classi di utilità per utilizzare *Cobertura*;

La cartella creata dovrà contenere i seguenti file:

```
vendor/lib/report/cobertura/
|-- asm-2.2.1.jar
|-- asm-tree-2.2.1.jar
|-- cobertura.jar
|-- jakarta-oro-2.0.8.jar
|-- jakarta-oro-license.txt
|-- log4j-1.2.9.jar
`-- log4j-license.txt
```

Questa cartella dovrà essere gestita dal sistema di versionamento, in questo modo sarà possibile controllare se i test di unità sono esaustivi sia dall'ambiente di lavoro che dall'ambiente di *build*.

Utilizzo di *Cobertura* nel processo di *build*

Per controllare se i test di unità, presenti in un progetto gestito da *Ant*, sono esaustivi è necessario inserire nel file `build.xml`del progetto le seguenti proprietà e attività:

```
1  <property name="vendor.lib.cobertura.dir" value="vendor/lib/report/cobertura/" />
   <property name="cobertura.report.dir" value="${reports}/cobertura-results" />
3  <property name="instrumented.dir" value="${build.dir}/cobertura-instrument" />

5  <path id="cobertura.classpath">
     <path refid="test.classpath"/>
7    <fileset dir="${vendor.lib.cobertura.dir}">
       <include name="**/*.jar" />
9    </fileset>
   </path>
11
   <taskdef classpathref="cobertura.classpath" resource="tasks.properties" />
```

In questo modo vengono definite le variabili e gli elementi che permettono di definire:

- Dove devono essere prodotte le reportistiche di questo strumento (riga 1);
- Dove sono contenute le librerie necessarie ad utilizzare questo strumento (riga 2);
- Dove devono essere salvate le classi che devono essere utilizzate per verificare se i test di unità sono esaustivi;
- Un nuovo elemento `path` dove viene aggiunto a `test.classpath` le librerie necessarie ad utilizzare *Cobertura* (righe 5-10);
- Le attività necessarie ad utilizzare *Cobertura* attraverso *Ant* (riga 12);

Per utilizzare questo strumento è necessario aggiungere tre nuove attività al processo di *build*:

```
   <target name="instrument" depends="compile">
2    <delete file="cobertura.ser"/>
     <delete dir="${instrumented.dir}" />
4    <cobertura-instrument todir="${instrumented.dir}">
       <fileset dir="${build.prod.dir}">
6        <include name="**/*.class" />
         <exclude name="**/*Test.class" />
8      </fileset>
```

```
    </cobertura-instrument>
  </target>

  <target name="coverage-check">
    <cobertura-check branchrate="34" totallinerate="80" />
  </target>

  <target name="coverage-report">
    <delete dir="${cobertura.report.dir}" />
    <mkdir dir="${cobertura.report.dir}" />
    <cobertura-report destdir="${cobertura.report.dir}">
      <fileset dir="${src.main.java.dir}">
        <include name="**/*.java"/>
      </fileset>
      <fileset dir="${src.test.java.dir}">
        <include name="**/*.java"/>
      </fileset>
    </cobertura-report>
  </target>
```

Le attività sono :

- Righe 1-10 `instrument`: permette di selezionare le classi dove dovranno essere eseguiti i test di unità per capire se questi sono esaustivi. Questa attività dipende dall'attività di compilazione;
- Righe 12-14 `coverage-check`: permette di controllare se, dopo aver eseguito l'attività di `instrument` e l'esecuzione dei test nelle classi selezionate, il codice del progetto è testato come definito in questa attività;
- Righe 16-26 `coverage-report`: permette di pubblicare le reportistiche prodotte da questo strumento;

Per permettere a questo strumento di analizzare se i test di unità sono esaustivi è necessario modificare l'attività `test` nel seguente modo:

```
1  <target name="test" depends="compile-tests, copy-test-resources">
     <delete dir="${test.xml.dir}"/>
3    <mkdir dir="${test.xml.dir}"/>
     <junit haltonfailure="true"
5        fork="yes"
         errorproperty="test.failed"
7        failureproperty="test.failed">

9      <classpath location="${instrumented.dir}" />
       <classpath refid="cobertura.classpath" />
11
       <formatter type="brief" usefile="false" />
13     <formatter type="xml"/>
       <batchtest todir="${test.xml.dir}">
15       <fileset dir="${build.test.dir}" includes="**/*Test.class" />
       </batchtest>
17   </junit>

19   <fail message="Tests failed! Check test reports."
     if="test.failed" />
21 </target>
```

Le modifiche effettuate sono presenti nella riga 5 dove viene specificato, tramite l'attributo `fork` dell'elemento `junit`, che i test devono eseguire in separate *virtual machine*. Nelle righe 9-10 viene modificato il `classpath`. Per prime vengono specificate le classi selezionate dall'attività `instrument` e successivamente viene specificato il `classpath`, precedentemente specificato, che contiene le classi di test e le librerie necessarie ad utilizzare *Cobertura*.

Dato che per eseguire questo strumento è necessario eseguire sempre la stessa procedura è consigliato inserire una nuova attività che verrà invocata per controllare se i test di unità sono esaustivi.

```
1  <target name="coverage"
     depends="instrument,test,coverage-report"
3    description=" instrument ourself, run the tests and generate JUnit and coverage reports."/>
```

Per eseguire questa attività è necessario eseguire il seguente comando:

```
ant coverage
```

In questo modo verrà controllato se i test di unità sono esaustivi e prodotte le reportistiche nella *directory* specificata dalla variabile `cobertura.report.dir`. Per maggiori informazioni su questo strumento è consigliato consultare `http://cobertura.sourceforge.net/anttaskreference.html`.

5.4 Automazione con *Maven*

Di seguito viene descritto come implementare il processo di *build* utilizzando *Maven*. Per installare *Maven* seguire le istruzioni reperibili all'indirizzo: `http://maven.apache.org/download.html` Per verificare se *Maven* è installato nel sistema è necessario eseguire il seguente comando:

```
mvn -version
```

Se la versione di *Maven* viene visualizzata correttamente significa che è installato correttamente nel sistema. Di seguito viene descritto come creare e configurare il *Project Object Model* per il progetto BowlingScore.

5.4.1 Creazione del progetto

Maven mette a disposizione uno strumento che permette di creare un progetto conforme alle convenzioni standard. Il progetto BowlingScore aderisce già allo standard delle *directory* di *Maven*.

Per creare il progetto BowlingScore è stato utilizzato lo strumento `archetype`, che permette di creare, tramite un'automazione a comando, un progetto che aderisce agli standard *Maven*.

Per creare la struttura delle *directory* e il *POM* del progetto BowlingScore è stato eseguito il seguente comando in una direcotroy temporanea:

```
mkdir /home/nicola/temp
cd /home/nicola/temp
mvn archetype:create -DgroupId=it.unipd.app -DartifactId=bowlingscore
```

Dopo che è stato eseguito il comando, *Maven* notificherà lo stato dell'esecuzione e creerà una *directory* chiamata `bowlingscore`.

Se si esplora questa *directory* si potrà notare che la struttura di questa è quasi uguale a quella descritta nella sezione 5.3.2 (manca la *directory* `src/test/resources`).

È stato creato anche il file `pom.xml` che contiene la descrizione del progetto appena creato.

Listing 5.2: /home/nicola/tmp/bowlingscore/pom.xml

```
1  <project>
     <modelVersion>4.0.0</modelVersion>
3    <groupId>it.unipd.app</groupId>
     <artifactId>bowlingscore</artifactId>
5    <packaging>jar</packaging>
     <version>1.0-SNAPSHOT</version>
7    <name>bowlingscore</name>
     <url>http://maven.apache.org</url>
9    <dependencies>
       <dependency>
11       <groupId>junit</groupId>
         <artifactId>junit</artifactId>
13       <version>3.8.1</version>
         <scope>test</scope>
15     </dependency>
     </dependencies>
17 </project>
```

Il file `pom.xml` creato, contiene i seguenti elementi:

- `porject`: Identifica un progetto gestito con *Maven*.
- `modelVersion`: Identifica la versione dell'*POM* utilizzato dal progetto. La versione cambia raramente, questo elemento è necessario per garantire stabilità quando verranno introdotto nuove versioni di *POM*.
- `groupId`: Contiene il nome univoco dell'organizzazione che crea il progetto. È uno degli elementi chiave che permettono di identificare univocamente un progetto gestito con *Maven*.

- `artifacId`: Contiene il nome univoco del prodotto principale creato con questo progetto. I prodotti creati tramite *Maven* hanno il nome composto dai seguenti elementi:
  `<artifactId>-<version>.<extension>`.

- `packaging`: Indica il tipo di prodotto che verrà prodotto da questo progetto. In questo esempio viene creato un prodotto di tipo *JAR*.

- `version`: Indica la versione del prodotto generato dal progetto. Se il progetto è in sviluppo (e non ancora in produzione) la versione sarà *SNAPSHOT*.

- `name`: Indica il nome usato per identificare il progetto. Se viene creata la documentazione attraverso *Maven*, verrà visualizzato il valore di questo elemento per indicare il progetto.

- `url`: Indica l'indirizzo del sito del progetto che si sta sviluppando

- `dependencies`: Indica l'elenco delle dipendenze del progetto. In questo esempio il progetto ha solo una dipendenza con *Junit*. Come si può vedere vengono specificati i campi di `groupId`, `artifactId`, `version` per riferirsi in modo univoco ad un prodotto. L'elemento `scope` permette di definire che *Junit* deve essere utilizzato solo per lo scopo di test.

Per una completa lista di tutti gli elementi disponibili per il *POM* è possibile consultare questo indirizzo: `http://maven.apache.org/maven-model/maven.html`

Copiare il file `pom.xml` nella cartella del progetto BowlingScore precedentemente sviluppato ed eliminare la cartella `tmp`:

```
cp home/nicola/temp/bowlingscore/pom.xml home/nicola/work/bowlingscore/
rm -r home/nicola/temp
```

Nelle prossime sezioni verrà modificato il file `pom.xml` in modo da poter effettuare il processo di *build* del progetto BowlingScore attraverso *Maven*.

5.4.2 Compilazione dei sorgenti

Come precedentemente descritto, per realizzare il processo di *build* attraverso *Maven*, basta configurare il file `pom.xml` in modo dichiarativo. Per permettere la compilazione dei sorgenti attraverso *Maven* verranno sfruttati i principi:

- Convenzioni sulla configurazione

- Riuso di logiche di *build*

- Esecuzione dichiarativa

Per eseguire la compilazione del progetto è necessario eseguire i seguenti comandi:

```
cd home/nicola/work/bowlingscore
mvn compile
```

Dopo l'esecuzione del comando verrà segnalato il seguente errore:

```
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure
/home/nicola/work/bowlingscore/src/main/java/it/unipd/app/MultiplayerBowlingGame.java:
[7,21] generics are not supported in -source 1.3
(use -source 5 or higher to enable generics)
    private ArrayList<BowlingGameInterface> playersGame;
...
```

Come si può intuire dal messaggio d'errore, *Maven* ha eseguito con la versione del compilatore *javac* specificato nel *Super POM* (che permette la compilazione di sorgenti *Java* 1.3).

Per permettere a *Maven* di compilare i sorgenti *Java* versione 1.5 è necessario modificare il file `pom.xml` nel seguente modo:

Listing 5.3: work/bowlingscore/pom.xml

```
1  <project>
     <modelVersion>4.0.0</modelVersion>
3    <groupId>it.unipd.app</groupId>
     <artifactId>bowlingscore</artifactId>
5    <packaging>jar</packaging>
     <version>1.0-SNAPSHOT</version>
7    <name>bowlingscore</name>
     <url>http://maven.apache.org</url>
9    <dependencies>
       <dependency>
11       <groupId>junit</groupId>
         <artifactId>junit</artifactId>
13       <version>3.8.2</version>
         <scope>test</scope>
15     </dependency>
     </dependencies>
17
     <build>
19     <plugins>
         <plugin>
21         <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
23         <version>2.0</version>
           <configuration>
25           <source>1.5</source>
             <target>1.5</target>
27         </configuration>
         </plugin>
29     </plugins>
     </build>
31
   </project>
```

Nelle righe 18-30 è stato configurato `maven-compiler-plugin` in modo da permettere la compilazione di sorgenti *Java* versione 1.5.

Se si eseguono i seguenti comandi verranno compilati i sorgenti del codice di produzione del progetto:

```
cd home/nicola/work/bowlingscore
mvn compile
```

L'esecuzione del comando produrrà il seguente *output*:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building bowlingscore
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
Downloading: http://repo1.maven.org/maven2/org/apache/maven/plugins/\
maven-compiler-plugin/2.0/maven-compiler-plugin-2.0.pom
1K downloaded
Downloading: http://repo1.maven.org/maven2/org/apache/maven/plugins/\
maven-compiler-plugin/2.0/maven-compiler-plugin-2.0.jar
13K downloaded
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 7 source files to /home/nicola/work/bowlingscore/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Mon May 05 13:14:20 GMT 2008
[INFO] Final Memory: 4M/54M
[INFO] ------------------------------------------------------------------------
```

Nella notifica prodotta da *Maven*, viene indicato che viene eseguita l'attività compile, viene scaricato `maven-compiler-plugin`, vengono compilati i sorgenti di produzione e vengono salvati i .class nella *directory* `bowlingscore/target/classes`.

Come si può notare nel file `pom.xml` non vengono specificate ne la *directory* che contiene il codice sorgente, ne la *directory* dove devono essere salvati i compilati. Questo perché *Maven* sfrutta il principio "convenzioni sulle configurazioni". Di *default* i sorgenti di produzione di un progetto gestito con *Maven* sono contenuti

nella cartella `src/main/java`. Questo valore di configurazione è ereditato dal *Super POM*, e visto che il progetto BowlingScore utilizza la struttura delle *directory* standard di *Maven* non è necessario specificare questo valore.

Se per caso un progetto ha la necessità di utilizzare una struttura delle *directory* diversa da quella specificata dallo standard, è necessario specificarla nel *POM* tramite l'elemento `sourceDirectory` all'interno dell'elemento `build`.

I file sorgenti sono stati compilati da un *plugin*. *Maven* ha sfruttato il principio "riuso di logiche di *build*", infatti è stato scaricato `maven-compiler-plugin` in modo che tutte le prossime compilazioni, che verranno effettuate (anche di altri progetti), utilizzeranno questo elemento che è stato configurato nel file `pom.xml`. Come si è potuto notare è stato automaticamente associato il comando compile a questo *plugin*. *Maven* definisce un processo di *build* standard (*build life cycle*[4]) che può essere eseguito attraverso comandi standard.

5.4.3 Compilazione ed esecuzione dei test

Durante la compilazione dei test del progetto BowlingScore, verrà sfruttato il principio:

- Organizzazione coerente delle dipendenze

Per compilare ed eseguire i test è necessario eseguire il seguente comando:

```
cd /home/nicola/work/bowlingscore
mvn test
```

Dopo l'esecuzione del comando verrà visualizzato il seguente *output*:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building bowlingscore
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
Compiling 11 source files to /home/nicola/work/bowlingscore/target/test-classes
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure

/home/nicola/work/bowlingscore/src/test/java/it/unipd/app/MultiplayerBowlingGameTest.java:
[4,26] package org.easymock does not exist
```

Come si può osservare, *Maven* notifica che sta tentando di eseguire l'attività test. Prima di eseguire l'attività test vengono eseguite le attività:

- `resources:resource`: permette di copiare le risorse necessarie per l'esecuzione del codice di produzione (dalla *directory* `src/main/resources`)
- `compiler:compile`: permette la compilazione del codice sorgente di produzione
- `resources:testResources`: permette di copiare le risorse necessarie per l'esecuzione del codice di test (dalla *directory* `src/test/resources`)
- `compiler:testCompile`: permette la compilazione del codice di test

Come si può notare la compilazione dei test è stata terminata con un errore. Questo perché, nel file `pom.xml`, non è stato specificato che i test utilizzano la dipendenza *easymock 2.2*.

Per specificare questa dipendenza è necessario effettuare le seguenti operazioni:

- Comprendere come reperire e specificare le dipendenze: Per reperire le dipendenze e capire come specificarle nel *POM* è consigliato consultare il seguente indirizzo:`http://mvnrepository.com`. A questo indirizzo è possibile ricercare e reperire il codice che dovrà essere specificato nel file `pom.xml`.

[4] `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`

- Specificare la libreria nel file `pom.xml`: Dopo aver reperito la definizione della dipendenza è necessario specificarla nel file `pom.xml`.

In questo modo, sfruttando il principio "organizzazione coerente delle dipendenze" è possibile modificare il file `pom.xml` nel seguente modo:

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.easymock</groupId>
      <artifactId>easymock</artifactId>
      <version>2.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Come si può notare è stata aggiunta (nelle righe 8-13) la dipendenza alla libreria *easymock 2.2*. È stato anche specificato, attraverso l'elemento `scope`, che la dipendenza deve essere utilizzata soltanto per la compilazione e l'esecuzione del codice di test. Dopo aver effettuato questa modifica, è necessario eseguire nuovamente il comando:

```
cd /home/nicola/work/bowlingscore
mvn test
```

Dopo che le dipendenze sono state scaricate, verranno terminate con successo le attività di compilazione ed esecuzione dei test.

Dopo l'esecuzione di questo comando vengono create due *directory*:

- `target/test-classes`: Contiene le risorse e i compilati dei test
- `target/surefire-reports`: Contiene i file di reportistica dei risultati dei test

5.4.4 Creazione del pacchetto e installazione locale del prodotto

Per creare il file *JAR* del progetto utilizzando *Maven* è necessario eseguire i seguenti comandi:

```
cd /home/nicola/work/bowlingscore
mvn package
```

Se si osserva il file `pom.xml` del progetto si potrà notare che l'elemento `packaging` è impostato a `jar`. In questo modo *Maven* riesce a capire che deve produrre questo tipo di file se viene eseguito il comando `mvn package`.

Dopo l'esecuzione del comando, viene creato il file `bowlingscore-1.0-SNAPSHOT.jar` nella *directory* `target`. Per installare il prodotto nella *repository* locale (`/home/nicola/.m2/repository`) è necessario eseguire il seguente comando:

```
cd /home/nicola/work/bowlingscore
mvn install
```

Come si può notare l'esecuzione del comando `install` ha eseguito tutte le attività finora descritte e in fine ha copiato il file nella *repository* locale. In questo modo se la *repository* locale è condivisa, altri sviluppatori possono accedere ed utilizzare il prodotto installato.

5.4.5 Importazione del progetto nell'*IDE*

Poiché *Maven* adotta il principio "*Coherent Organization of Dependencies*", per permettere agli sviluppatori di sviluppare progetti *software* gestiti da *Maven* nel *IDE* desiderata sono disponibili alcuni comandi.

Per sviluppare un progetto gestito con *Maven* con l'*IDE eclipse* è necessario eseguire questo comando:

```
mvn eclipse:eclipse
```

Questo comando permette di creare i file di configurazione per importare il progetto gestito con *Maven* in *eclipse*. Oltre a questa operazione è necessario impostare in *eclipse* la variabile `M2_REPO` , nel `classpath` del progetto, con il valore del path assoluto della *repository* locale di *Maven* (`/home/nicola/.m2/repository`).

Per sviluppare un progetto gestito con *Maven* con l'*IDE IntelliJ IDEA* è necessario eseguire questo comando:

```
mvn idea:idea
```

Questo comando permette di creare i file di configurazione per importare il progetto gestito con *Maven* in *IntelliJ IDEA*.

5.4.6 Creazione della documentazione

Grazie alle informazioni contenute nel file `pom.xml`è possibile creare un sito che descrive le informazioni principali del progetto. Per creare il sito è necessario eseguire il seguente comando:

```
mvn site
```

Dopo l'esecuzione del comando, se si osserva la *directory* `target`,si noterà che è stata creata una *directory* `site`. Questa *directory* conterrà il sito di documentazione del progetto. Se si osserva il sito di documentazione (consultabile da un *Web browser* aprendo il file `bowlingscore/target/site/intex.html`) si potranno consultare tutte le informazioni che sono state espresse nel *POM* del progetto. Come si potrà notare non sono presenti le reportistica dei test del progetto.

Per visualizzare le reportistiche di test è necessario modificare il *POM* del progetto e aggiungere un *plugin* di reportistica. È necessario modificare il file `bowlingscore/pom.xml` nel seguente modo:

Listing 5.4: bowlingscore/pom.xml

```
<project>
...

  <reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
      </plugin>
      ...
    </plugins>
  </reporting>

</project>
```

In questo modo è stato aggiunto un'*plugin* che permette di visualizzare le reportistiche dei test nel sito del progetto creato da *Maven*. Dopo aver modificato il *POM* del progetto, se viene eseguito il comando:

```
mvn site
```

Verrà creato un nuovo sito con una sezione *"Project Reports"* dove saranno presenti le reportistiche dei test.

Nella prossima sezione viene descritto come produrre ulteriori reprotistiche che segnalano alcuni errori ricorrenti presenti nel progetto e producono alcune misure di qualità.

Nella sezione 3.12 del libro [16](distribuito gratuitamente) viene spiegato come creare e configurare un sito di documentazione per un progetto attraverso *Maven*.

Nella sezione 6.2 di questa guida viene descritto come pubblicare il sito di documentazione, tramite un'automazione ad evento, in modo che sia accessibile da tutti gli sviluppatori del progetto.

5.4.7 *PMD*: Ricerca di errori ricorrenti nel progetto

PMD è uno strumento che, in base ad un'insieme di regole predefinite o create dallo sviluppatore, esamina del codice sorgente *Java* al fine di trovare dei possibili problemi come:

- Possibili errori
- Codice inutilizzato

- Codice non ottimizzato
- Espressioni inutilmente complicate
- Duplicazioni del codice

Per utilizzare questo strumento attraverso *Maven* è necessario aggiungere al *POM* del progetto i seguenti elementi:

Listing 5.5: bowlingscore/pom.xml

```
1  <reporting>
     <plugins>
3      ...
       <plugin>
5        <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-jxr-plugin</artifactId>
7        <version>2.1</version>
       </plugin>
9      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
11       <artifactId>maven-pmd-plugin</artifactId>
         <version>2.3</version>
13       <configuration>
           <targetJdk>1.5</targetJdk>
15         <rulsets>
             <rulset>/rulesets/basic.xml</rulset>
17           <rulset>/rulesets/imports.xml</rulset>
             <rulset>/rulesets/unusedcode.xml</rulset>
19           <rulset>/rulesets/finalizers.xml</rulset>
           </rulsets>
21       </configuration>
       </plugin>
23     ...
     </plugins>
25 </reporting>
```

In questo modo sono stati aggiunti due *plugin*:

- Righe 4-8: È stato aggiunto `maven-jxr-plugin` che permette di inserire nella documentazione il codice sorgente del progetto. In questo modo la documentazione prodotta dagli altri *plugin* di reportistica farà riferimento alle reportistiche prodotte da questo *plugin*
- Righe 9-22: È stato aggiunto e configurato `maven-pmd-plugin` in modo da poter analizzare codice sorgenti *Java* versione 1.5.
- Righe 15-20: Sono state specificate le regole:
  - `/rulesets/basic.xml`: Permette di ricercare degli errori ricorrenti o delle imperfezioni che possono verificarsi durante la fase di sviluppo di un progetto;
  - `/rulesets/imports.xml`: Permette di trovare importazioni di classi o pacchetti di classi ridondanti, inutilizzate o duplicate;
  - `/rulesets/unusedcode.xml`: Permette di trovare parti di codice inutilizzate. (p.es. attributi o metodi privati non utilizzati all'interno della classe);
  - `/rulsets/finalizers.xml`: Permette di trovare metodi e parametri che possono essere dichiarati `final`;

Se viene eseguito il comando:

```
cd /home/nicola/work/bowlingscore
mvn pmd:pmd
```

Viene creata una pagina *Web* (`bowlingscore/target/site/pmd.html`) dove vengono riportati tutti gli errori trovati da *PMD*. Se viene eseguito il comando:

```
cd /home/nicola/work/bowlingscore
mvn site
```

Viene aggiunta la pagina di reportistica di *PMD* nella sezione *"Project Reports"* nel sito del progetto.

Specificare le regole in un file esterno

A volte capita di dover riutilizzare le stesse regole per più progetti o specificare delle regole per uno specifico modulo del progetto. In questi casi è consigliato non specificare le regole direttamente nel *POM*, ma specificarle in un file del progetto.

Per ottenere lo stesso risultato dell'esempio precedente è necessario creare il seguente file:

Listing 5.6: bowlingscore/src/main/pmd/custom.xml

```
<?xml version="1.0"?>
<ruleset name="custom">
 <description>
  Default rules
 </description>
 <rule ref="/rulesets/basic.xml" />
 <rule ref="/rulesets/imports.xml" />
 <rule ref="/rulesets/unusedcode.xml" />
 <rule ref="/rulesets/finalizers.xml" />
</ruleset>
```

In questo file vengono specificate le regole da utilizzare nel progetto.

Per permettere a *PMD* di leggere le regole da un file è necessario modificare la configurazione del plugin nel seguente modo:

Listing 5.7: bowlingscore/pom.xml

```
<reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <targetJdk>1.5</targetJdk>
          <rulsets>
            <ruleset>${basedir}/src/main/pmd/custom.xml</ruleset>
          </rulsets>
        </configuration>
      </plugin>
      ...
    </plugins>
  </reporting>
```

Per ulteriori esempi o per creare regole personalizzate è consigliato visitare il sito `http://pmd.sourceforge.net/`. In questo sito è presente molta documentazione su *PMD*, in particolare vengono descritte le regole fornite e come creare delle regole personalizzate con *PMD*.

Per selezionare le regole appropriate per un progetto è consigliato seguire le linee guida specificate nel sito `http://pmd.sourceforge.net/bestpractices.html`:

- Selezionare solo le regole che sembrano appropriate per un progetto: Non c'è nessun vantaggio ad utilizzare molte regole e avere centinaia o migliaia di segnalazioni di violazioni.
- Iniziare con poche regole e successivamente aumentare il numero selezionando solo le regole che interessano

Verifica delle regole nel processo di *build*

Per verificare che le regole siano rispettate durante l'esecuzione del processo di *build*, e non solo nella fase di documentazione, è necessario modificare il *POM* del progetto nel seguente modo:

Listing 5.8: bowlingscore/pom.xml

```
 <build>
    <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <executions>
```

```
7         <execution>
            <goals>
9             <goal>check</goal>
            </goals>
11        </execution>
        </executions>
13    </plugin>
      ...
15  </plugins>
 </build>
```

Di *default* la l'attività `pmd:check` viene eseguita nella fase *verify* del processo di *build*. La fase *verify* esegue tra la fase *package* e la fase *install* del processo di *build*. In questo modo quando viene eseguito il comando:

```
mvn verify
```

viene verificato che nel progetto vengano rispettate le regole imposte.

Ricerca di duplicazioni del codice

Il plugin *PMD* contiene uno strumento che permette di ricercare e segnalare duplicazioni del codice. Questo strumento è *CPD* (*copy and paste detection*) e permette di creare una reportistica dove vengono segnalate tutte le duplicazioni presenti nel codice di un progetto.

Per utilizzare questo strumento attraverso *Maven* è necessario modificare il *POM* nel seguente modo:

Listing 5.9: bowlingscore/pom.xml

```
<reporting>
2     <plugins>
        ...
4       <plugin>
          <groupId>org.apache.maven.plugins</groupId>
6         <artifactId>maven-pmd-plugin</artifactId>
          <configuration>
8             <rulesets>
               <ruleset>${basedir}/src/main/pmd/custom.xml</ruleset>
10            </rulesets>
              <minimumTokens>100</minimumTokens>
12            <targetJdk>1.5</targetJdk>
          </configuration>
14      </plugin>
        ...
16    </plugins>
  <reporting>
```

Come si può notare è stato aggiunto l'elemento di configurazione `minimumTokens`. Questo elemento specifica il numero minimo di elementi presenti nel codice, da considerare come copia e incolla (in questo esempio vengono considerati almeno 100 *tokens* che corrispondono a 5-10 righe di codice).

Per verificare se nel codice sorgente sono presenti duplicazioni di codice è necessario eseguire il comando:

```
mvn pmd:cpd
```

Tramite questo comando viene creata la pagina `/target/site/cpd.html` dove vengono riportate le duplicazioni trovate.

Se viene eseguito il comando

```
mvn site
```

Viene aggiunta la pagina di reportistica di *CPD* nella sezione *"Project Reports"* nel sito del progetto.

5.4.8 *Checkstyle*: Verifica del rispetto di convenzioni di stile

La descrizione di questo strumento è presente nella sezione 5.3.7 a pagina 84.

Per controllare che il progetto implementato rispetti le convenzioni di stile di *Sun Java* è necessario configurare il plugin `maven-checkstyle-plugin` nel *POM* del progetto.

Listing 5.10: bowlingscore/pom.xml

```
<reporting>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <configuration>
          <configLocation>config/sun_checks.xml</configLocation>
        </configuration>
      </plugin>
      ...
    </plugins>
</reporting>
```

Come si può vedere, è stato aggiunto nella sezione `reporting` il *plugin* `maven-checkstyle-plugin` ed è stato configurato in modo da controllare che nel codice sorgente siano rispettate le convenzioni di stile *Sun Java*. L'elemento `configLocation` può essere configurato in modo che si riferisca ad uno dei file forniti da *Checkstyle* o un file, all'interno del progetto, creato appositamente dagli sviluppatori.

Configurazione	**Convenzione di stile**	**Riferimento**
`config/sun_checks.xml`	*Sun Java*	`http://java.sun.com/docs/codeconv/`
`config/maven_checks.xml`	*Maven*	`http://maven.apache.org/guides/development/guide-m2-development.html`
`config/turbine_checks.xml`	Progetto *Jakarta Turbine*	`http://turbine.apache.org/common/code-standards.html`

Tabella 5.1: Convenzioni di stili presenti in *Checkstyle*

È consigliato utilizzare le convenzioni fornite da *Checkstyle* (vedi tabella 5.1). Se le convenzioni sono molto diffuse è più probabile che siano facilmente comprese e rispettate dagli sviluppatori e che assicurino la realizzazione di codice professionale.

Se è necessario creare una convenzione personalizzata è consigliato consultare l'indirizzo `http://checkstyle.sourceforge.net/config.html` dove vengono forniti alcuni esempi.

Se viene eseguito il comando:

```
mvn checkstyle:checkstyle
```

verrà creata la pagina `target/site/checkstyle.html` dove vengono riportate tutte le infrazioni segnalate da *Checkstyle*.

Se viene eseguito il comando:

```
mvn site
```

Viene aggiunta la pagina di reportistica di *Checkstyle* nella sezione *"Project Reports"* nel sito del progetto.

Per eseguire le verifiche di *Checkstyle* nella fase di *build* del processo effettuare le stesse operazioni specificate a pagina 95.

5.4.9 *Cobertura*: Misurazione della percentuale di codice verificato

La descrizione di questo strumento è presente nella sezione 5.3.8 a pagina 85. Per utilizzare *Cobertura* attraverso *Maven* è necessario modificare il *POM* del progetto nel seguente modo:

```
<reporting>
      ...
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
        <version>2.0</version>
      </plugin>
      ...
    </plugins>
</reporting>
```

Visto che le versioni superiori alla 2.0 hanno dei problemi durante la fase di `instrument`, nell'esempio è stata utilizzata la versione 2.0 del *plugin* che è stato inserito nella sezione di reportistica del *POM*.

Se viene eseguito il comando:

```
mvn site
```

viene generata una reportistica di *Cobertura* sotto la sezione *"Project Reports"* del sito del progetto. Per permettere la cancellazione dei file generati da *Cobertura* e per imporre il rispetto di alcune metriche di qualità nel progetto è necessario modificare il *POM* nel seguente modo:

```
<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.0</version>
      <configuration>
        <check>
          <totalLineRate>80</totalLineRate>
          <totalBranchRate>100</totalBranchRate>
        </check>
      </configuration>
      <executions>
        <execution>
        <id>clean</id>
          <goals>
            <goal>clean</goal>
          </goals>
        </execution>
        <execution>
        <id>check</id>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
<build>
```

È stato necessario specificare il *plugin* anche nell'elemento `build` del progetto. Il *plugin* è stato configurato nel seguente modo:

- Righe 4-7: È stato specificato il *plugin* da configurare
- Righe 8-13: Vengono specificate le percentuali di test che deve avere il progetto
- Righe 15-20: È stato specificato che quando viene invocato il comando `mvn clean` vengono eliminati anche i file creati da *Cobertura*
- Righe 21-26: È stato specificato che quando viene invocato il comando `mvn verify` viene verificato che il codice del progetto sia verificato come configurato nelle righe 8-13

In questo modo se viene eseguito il comando `mvn cobertura:check` o il comando `mvn verify` viene controllato che il progetto sia verificato nelle misure specificate nel *POM*.

La configurazione specificata nell'esempio è abbastanza generica (viene stabilita una percentuale di test per tutto il progetto). *Cobertura* permette di rendere il controllo più specifico in modo da verificare che la percentuale di test sia specificata per pacchetto o per classe. Per maggiori informazioni riguardanti la configurazione di questo *plugin* è consigliato leggere la documentazioni che si trova al seguente indirizzo:
`http://mojo.codehaus.org/cobertura-maven-plugin/`

Capitolo 6

Pianificare il processo di *build*

In questo capilo viene descritto come realizzare un sistema di integrazione continua per gestire un progetto sviluppato in *Java*. Nelle successive sezioni vengono forniti degli esempi per realizzare un sistema di integrazione continua utilizzando *Cruise Control* e *Continuum*.

6.1 Realizzare il sistema di integrazione continua con *CruiseControl*

CruiseControl è un *framework* che permette la realizzazione di un sistema di integrazione continua. Nelle prossime sezioni verrà descritto come realizzare un'automazione ad evento, che permette di creare un processo di *Continuous Integration* per il progetto BowlingScore.

Installazione di *CruiseControl*

Prima di installare *CruiseControl* è necessario trovare un computer *server* che lo ospiti. Questo computer avrà il compito di compilare i codici sorgenti dei progetti ed eseguire il codice di test e dovrà possedere le seguenti caratteristiche:

- Riprodurre le caratteristiche dell'ambiente di produzione
- Essere accessibile dagli sviluppatori
- Poter accedere al sistema di versionamento
- Avere a disposizione *Ant*
- Avere a disposizione un *client* del sistema di versionamento

Dopo aver identificato la macchina, sarà necessario installare *CruiseControl*. Per installare *CruiseControl* è necessario scaricare una copia dell'applicativo all'indirizzo:
`http://cruisecontrol.sourceforge.net/download.html`
In questo volume si fa riferimento alla versione cruisecontrol-src-2.7.2.

La distribuzione è contenuta in un archivio di tipo *ZIP*. Estrarre i file in una *directory* (in questo volume si è scelto di utilizzare la *directory* `/home/nicola/CC_HOME`).

Per installare *CruiseControl* è necessario eseguire il processo di *build* tramite i seguenti comandi:

```
cd /home/nicola/CC_HOME/main
sh build.sh
```

L'automazione a comando eseguita, compilerà i sorgenti di *CruiseControl* ed effettuerà i test. Se tutto è andato a buon fine, alla fine dell'esecuzione, nella *directory* `/home/nicola/CC_HOME/main/dist`, sarà presente il file `cruisecontrol.jar`.

Preparazione dell'ambiente di *build*

È necessario creare un ambiente nel *server* di *build* dove verrà eseguito *CruiseControl*. In questo ambiente, per ogni progetto contenuto nel sistema di versionamento, verranno fatte le seguenti azioni:

- Scaricato il progetto dal sistema di versionamento
- Eseguito il processo di *build* del progetto scaricato
- Mantenuto uno storico dello stato delle *build*

Queste azioni verranno svolte ogni volta che avviene un cambiamento ad un file del progetto contenuto nel sistema di versionamento.

Creazione della *directory* di *build*

L'ambiente di *build* corrisponderà ad una *directory*, presente nella macchina di *build*, dove verranno salvati tutti i file di configurazione necessari a *CruiseControl* per effettuare il processo di *build* del progetto. È quindi necessario creare una cartella che corrisponderà con l'ambiente di *build*.

```
mkdir /home/nicola/builds
```

Nell'ambiente di *build* è necessario creare due *directory*:

1. `checkout` : conterrà una copia dell'ultima versione dei sorgenti del progetto
2. `logs` : conterrà i file di notifica creati da *CruiseControl*

Scaricare il progetto dal sistema di versionamento

Creare la *directory* `checkout` e scaricare una copia del progetto BowlingScore utilizzando i seguenti comandi:

```
cd /home/nicola/builds
mkdir checkout
cd checkout
svn co file:///home/nicola/svn-repo/bowlingscore/trunk bowlingscore
```

In questi esempi si è scelto di utilizzare la stessa macchina sia per il sistema di versionamento che per il sistema di *Continuous Integration*. Se si utilizzano due macchine distinte è necessario modificare l'indirizzo del *repository* del sistema di versionamento.

Creazione della *directory* di notifica

Creare la *directory* `logs` che conterrà i file di notifica del progetto.

```
cd /home/nicola/builds
mkdir logs
```

Creazione di un *wrapper* del file di *build*

L'automazione ad evento, ogni volta che viene effettuato un cambiamento ai file del progetto, contenuti nel sistema di versionamento, deve effettuare le seguenti operazioni:

1. Eliminare i file contenuti nella cartella `checkout` (eliminazione dell'*output* dell'ultimo processo di *build*)
2. Scaricare l'ultima revisione contenuta nel sistema di versionamento
3. Eseguire il processo di *build* del progetto

In questo modo verrà effettuato un processo di *build* per ogni revisione creata. Il processo di *build* inizierà sempre da capo (verranno sempre compilati tutti i file ed eseguiti tutti i test). Sarebbe possibile implementare questi tre passi nel file `build.xml` del progetto. È consigliato però, tenere l'automazione a comando separata dall'automazione che verrà schedulata da *CruiseControl*. È quindi necessario creare nella *directory* `builds` un nuovo file, eseguibile da *Ant*, chiamato `cc-build.xml`.

Poiché *Ant* non ha a disposizione un elemento che permette di scaricare file da *subversion*, è necessario scaricare un applicativo che permetta di effettuare questa attività. L'applicativo utilizzato in questo volume è *svnant* reperibile a questo indirizzo:
`http://subclipse.tigris.org/svnant.html`. In questo volume è stata utilizzata la versione 1.0.0 di questo applicativo.

Scaricare quindi l'applicativo ed estrarlo in una *directory* a piacere (p.es `/tmp/svnant`). Dopo di che effettuare le seguenti operazioni:

```
cd home/nicola/builds
mkdir lib
cd lib
cp /tmp/svnant/lib/*.* .
```

È stato quindi necessario creare una *directory* `lib` nell'ambiente di *build* dove sono state copiate tutte le librerie necessarie per utilizzare *svnant*.

Il file `cc-build.xml`, permette di effettuare le operazioni sopra indicate:

Listing 6.1: builds/cc-home.xml

```
1  <project name="cc-build" default="build" basedir="checkout">
2
3    <property name="svn.url"
4              value="file:///home/nicola/svn-repo/bowlingscore/trunk" />
5
6    <property name="lib.dir" location="../lib" />
7    <property name="svnjavahl.jar" location="${lib.dir}/svnjavahl.jar" />
8    <property name="svnant.jar" location="${lib.dir}/svnant.jar" />
9    <property name="svnClientAdapter.jar"
10             location="${lib.dir}/svnClientAdapter.jar" />
11
12   <path id="project.classpath">
13     <pathelement location="${svnjavahl.jar}" />
14     <pathelement location="${svnant.jar}" />
15     <pathelement location="${svnClientAdapter.jar}" />
16   </path>
17
18   <taskdef resource="svntask.properties" classpathref="project.classpath"/>
19
20   <target name="build">
21     <delete dir="bowlingscore" />
22     <svn>
23       <checkout url="${svn.url}" destPath="bowlingscore"/>
24     </svn>
25     <ant antfile="build.xml" dir="bowlingscore" target="test" />
26   </target>
27 </project>
```

Le righe 6-17 permettono di definire l'elemento `svn` che viene utilizzato nelle righe 20-22. L'attività *build*, specificata nella riga 20, permette di:

- Eliminare la cartella `checkout/bowlingscore` (riga 21)
- Scaricare l'ultima revisione dal sistema di versionamento (righe 22-24)
- Eseguire l'attività `test` del file `checkout/bowlingscore/build.xml` tramite *Ant* (riga 25)

Verificare il funzionamento dell'automazione

A questo punto è possibile verificare se l'automazione creata funziona come desiderato. Per far ciò è necessario eseguire il seguente comando:

```
cd /home/nicola/builds
ant -f cc-build.xml
```

Se *Ant* esegue con successo l'automazione, nella cartella `builds/checkout/bowlingscore/build` saranno presenti i compilati del progetto. A questo punto è consigliato salvare il file `cc-build.xml` e la cartella `lib` nel sistema di versionamento.

Configurare *CruiseControl*

Dato che il file `cc-build.xml` effettua tutte le attività richieste, è necessario far eseguire queste attività da *CruiseControl* ogni volta che avviene un cambiamento ai file del progetto.

Per far ciò è necessario creare un file di configurazione per *CruiseControl*. Di *default* *CruiseControl* legge i dati di configurazione da un file chiamato `config.xml`. È quindi necessario creare questo file in modo che *CruiseControl* possa operare come desiderato.

Di seguito viene descritta ogni parte del file di configurazione.

Definire un progetto

Creare, con un *editor* di testo, il file config.xml nella *directory* `/home/nicola/builds` e inserire i seguenti elementi:

```
1 <cruisecontrol>
    <project name="bowlingscore" buildafterfailed="false">
```

L'elemento `cruisecontrol` specifica che questo è un file di configurazione di *CruiseControl*. L'elemento `project` identifica la configurazione di un progetto all'interno del file di configurazione. All'interno dell'elemento `cruisecontrol` possono essere definiti più progetti, che dovranno avere nomi differenti.

L'attributo `buildafterfailed` permette di specificare se *CruiseControl* deve eseguire il processo di *build* anche se non è stato effettuato nessun cambiamento nel sistema di versionamento. In questo caso è stato settato a falso. In questo modo il processo di *build* viene eseguito solo quando vengono fatte delle modifiche al progetto.

Definire i *listeners*

```
    <listeners>
2     <currentbuildstatuslistener
        file="logs/bowlingscore/currentbuildstatus.txt"/>
4   </listeners>
```

Questo elemento deve essere contenuto all'interno dell'elemento `project`. Permette di specificare dove *CruiseControl* dovrà leggere e scrivere lo stato del processo di *build*.

Controllare i cambiamenti

CruiseControl deve eseguire il processo di *build* solo quando sono avvenuti cambiamenti nel *repository* del progetto. Il processo di *build* è ripetibile, quindi non avrebbe senso eseguire lo stesso processo più volte nella stessa revisione perché i risultati non cambierebbero. Per questo si deve definire come e quando *CruiseControl* deve controllare e decidere se è necessario eseguire il processo di *build*.

```
    <modificationset quietperiod="60">
2     <svn localworkingcopy="checkout/bowlingscore"/>
    </modificationset>
```

L'elemento `modficationset` (contenuto all'interno dell'elemento `project`) permette di definire cosa *CruiseControl* deve controllare per capire se è necessario eseguire il processo di *build*. Poiché il progetto è contenuto in *subversion*, viene utilizzato l'elemento `svn` (fornito dalle librerie di *CruiseControl*) per indicare a *CruiseControl* di verificare se sono avvenute delle modifiche al progetto(indicato nell'attributo `localworkingcopy`).

Supponiamo che uno sviluppatore invii tre modifiche di tre file distinti al *repository* del progetto(tramite tre attività di *committing*). Lo sviluppatore vorrebbe che la *build* fosse eseguita nella revisione che contiene tutti i tre i cambiamenti.

Per permettere ciò è stato settato l'attributo `quietperiod`, che permette di specificare il tempo di assenza di cambiamenti prima che *CruiseControl* inizi il processo di *build*. In questo caso *CruiseControl* effettua il processo di *build* solo se non sono avvenuti cambiamenti dopo sessanta secondi dall'ultima modifica del progetto.

Programmare l'intervallo di esecuzione

È necessario definire quando e come il processo di *build* deve eseguire.

```
<schedule interval="60">
  <ant buildfile="cc-build.xml"
  target="build"/>
</schedule>
```

L'elemento `schedule` permette di definire quando *CruiseControl* cerca di effettuare il processo di *build*. In questo esempio è stato impostato l'attributo `interval` a sessanta secondi. Nel progetto BowlingScore *CruiseControl* opererà nel seguente modo:

- Ogni sessanta secondi verrà controllato se sono state effettuate delle modifiche (specificate nell'elemento `modificationset`)
- Se sono state fatte delle modifiche, verrà atteso un periodo di inattività di sessanta secondi prima di effettuare il processo di *build*

L'elemento ant permette di specificare come il processo di *build* deve eseguire. Nel progetto BowlingScore *CruiseControl* farà eseguire ad *Ant* l'attività *build* del file `cc-build.xml`.

Salvare i risultati dei test

Ogni volta che *CruiseControl* esegue il processo di *build* vengono generati dei file di reportistica. È buona prassi salvare questi file in modo da poterli consultare in futuro.

```
<log logdir="logs/bowlingscore">
  <merge dir="checkout/bowlingscore/build/reports/test-results" />
</log>
```

Nel progetto BowlingScore vengono salvati i file, contenuti nella *directory* `reports/test-results`, creati dal processo di *build*, nella *directory* `logs/bowlingscore`. Questa operazione è necessaria perché la *directory* `checkout/bowlingscore` (quindi anche le reportistiche prodotte) viene eliminata ogni volta che viene eseguito il processo di *build*. La *directory* `logs`, che non viene eliminata dal processo di *build*, conterrà lo storico delle reportistiche del progetto BowlingScore, che potranno essere consultate in futuro.

Generare i risultati dei test in *XML*

Il processo di *build* creato, che viene eseguito da *Ant*, non genera nessuna reportistica. I risultati dei test vengono visualizzati nella *console* ma non vengono salvati in nessun file. È quindi necessario modificare il file `build.xml` del progetto in modo da poter produrre i file di reportistica.

Di seguito viene modificata l'attività di test specificata nel file `build.xml`:

Listing 6.2: /home/nicola/work/bowlingscore/build.xml

```
<property name="reports" location="${build.dir}/reports"/>
<property name="test.xml.dir" location="${reports}/test-results"/>

<target name="test" depends="compile-tests, copy-test-resources">
  <delete dir="${test.xml.dir}"/>
  <mkdir dir="${test.xml.dir}"/>
  <junit haltonfailure="true"
                errorproperty="test.failed"
                failureproperty="test.failed">

    <classpath refid="test.classpath" />
    <formatter type="brief" usefile="false" />
    <formatter type="xml"/>
    <batchtest todir="${test.xml.dir}">
      <fileset dir="${build.test.dir}" includes="**/*Test.class" />
    </batchtest>
  </junit>
  <fail message="Tests failed! Check test reports."
  if="test.failed" />
</target>
```

Sono state aggiunte le seguenti modifiche:

- Righe 5-6: Sono state aggiunte due attività che permettono di eliminare e ricreare la *directory* che conterrà le reportistiche
- Righe 8-9: È stata creata una proprietà (`test.failed`) che diventa vera se il test ha errori o fallisce.
- Riga 13: Viene specificato che i risultati dei test vengono salvati in file nel formato *XML*.
- Riga 14: È stata aggiunto l'attributo `todir` all'elemento `batchtest` che permette di specificare la *directory* dove verranno salvate le reportistiche dei test.
- Righe 18-19: Viene specificato che se i test falliscono (quindi la proprietà `test.failed` è uguale a vero), nel processo di *build* viene visualizzato un messaggio.

In questo modo, quando verrà effettuato il processo di *build*, *CruiseControl* potrà salvare le reportistiche dei test delle varie versioni del progetto.

Di seguito viene riportato il file `config.xml` che contiene gli elementi descritti fino ad ora.

Listing 6.3: /home/nicola/builds/config.xml

```
<cruisecontrol>
  <project name="bowlingscore" buildafterfailed="false">
    <listeners>
      <currentbuildstatuslistener
       file="logs/bowlingscore/currentbuildstatus.txt"/>
    </listeners>

    <modificationset quietperiod="60">
      <svn localworkingcopy="checkout/bowlingscore"/>
    </modificationset>

    <schedule interval="60">
      <ant buildfile="cc-build.xml"
           target="build"/>
    </schedule>

    <log logdir="logs/bowlingscore">
      <merge dir="checkout/bowlingscore/build/reports/test-results" />
    </log>

  </project>
</cruisecontrol>
```

Eseguire *CruiseControl*

Visto che è stato creato il file di configurazione è possibile eseguire *CruiseControl*.

Per eseguire *CruiseControl*, posizionarsi nella *directory* `builds` ed eseguire i seguenti comandi:

```
cd home/nicola/builds
sh /home/nicola/CC_HOME/main/bin/cruisecontrol.sh
```

CruiseControl a questo punto leggerà il file `config.xml` e inizierà a schedulare il processo di *build* come configurato.

Nella console verranno visualizzati i seguenti messaggi di notifica:

```
projectName = [bowlingscore]
LConfigManager- using settings from config file [/home/nicola/builds/config.xml]
ProjectConfig - No previously serialized project
found [/home/nicola/builds/bowlingscore.ser], forcing a build.
Project       - Project bowlingscore starting
Project       - Project bowlingscore:  idle
```

Nella notifica vengono riportate le seguenti informazioni:

- Nella prima riga viene specificato il nome del progetto che *CruiseControl* sta controllando.
- Nella seconda riga viene specificato il nome del file di configurazione che sta utilizzando.

Ogni sessanta secondi *CruiseControl* ricarica le configurazioni contenute nel file `config.xml`. In questo modo le modifiche apportate a questo file verranno caricate dinamicamente da *CruiseControl*. Nella terza riga viene specificato che il processo di *build* non è mai stato eseguito, quindi viene forzata l'esecuzione di un processo di *build*.

```
BuildQueue    - now adding to the thread queue: bowlingscore
Project       - Project bowlingscore:  bootstrapping
Project       - Project bowlingscore:  checking for modifications
ModificationSet- 3 modifications have been detected.
Project       - Project bowlingscore:  now building
```

A questo punto inizia il processo di *build* e *CruiseControl* ne visualizza gli *output*. Terminato il processo di *build* verrà visualizzata la seguente notifica.

```
Project bowlingscore:  merging accumulated log files
Project bowlingscore:  build successful
Project bowlingscore:  publishing build results
Project bowlingscore:  idle
Project bowlingscore:  next build in 1 minutes
```

In questa notifica viene comunicato che il processo di *build* è andato a buon fine, e che *CruiseControl* controllerà se sono avvenute delle modifiche tra un minuto.

```
Project bowlingscore:  No modifications found, build not necessary.
Project bowlingscore:  idle
Project bowlingscore:  next build in 1 minutes
```

Poiché non sono state fatte modifiche non viene eseguito il processo di *build*. D'ora in poi ogni volta che viene effettuato una modifica al progetto (e inviata al sistema di versionamento), verrà eseguito un nuovo processo di *build*. Si consiglia quindi di modificare un file del progetto, inviare la modifica al sistema di versionamento e controllare l'*output* prodotto da *CruiseControl*.

Pubblicare lo stato del processo di *build*

Le notifiche pubblicate da *CruiseControl* nella console sono troppo prolisse e a volte sono difficili da interpretare. È quindi necessario avere informazioni del processo di *build* in diverse forme.

Di seguito vengono descritte due modi per pubblicare e notificare lo stato del processo di *build* al *team* di sviluppo.

Pubblicare lo stato del processo di *build* in una pagina *Web*

CruiseControl mette a disposizione un'applicazione *Web* che permette di pubblicare lo storico dei processi di *build*. In questo modo tutti gli sviluppatori potranno accedere e consultare gli stati dei processi di *build* di tutti i progetti gestiti tramite *CruiseControl*.

Per compilare e utilizzare l'applicazione *Web* di *CruiseControl* è necessario effettuare le seguenti operazioni:

```
cd /home/nicola/CC_HOME/reporting/jsp
sh build.sh
```

Durante l'esecuzione del processo di *build* vengono richiesti i seguenti parametri:

- `user.log.dir`: inserire il percorso assoluto della *directory* che contiene i file di *log* dei progetti (p.es. `/home/nicola/builds/logs`)
- `user.build.status.file`: inserire il percorso relativo, rispetto a user.log.dir, del file di stato dei processi di *build* (p.es. `currentbuildstatus.txt`)
- `set.artifacts.dir`: inserire il percorso assoluto della *directory* dove verranno pubblicati i prodotti (p.es. `/home/nicola/builds/artifacts`)

Dopo aver eseguito il processo di *build* verrà creato il file `cruisecontrol.war` nella *directory* `/home/nicola/CC_HOME/reporting/jsp/dist`.

Mettere in opera il file creato in un *server* (In *Apache Tomcat* basta copiare il file `cruisecontrol.war` nella *directory* `TOMCAT_HOME/webapps`).

A questo punto si può accedere all'applicazione *Web* dal seguente indirizzo: `http://buildmachine:port/cruisecontrol`

Da questa pagina si può accedere a tutti i risultati dei processi di *build* dei progetti gestiti da *CruiseControl*, come visualizzato in figura 6.1 alla pagina 106.

Figura 6.1: Applicazione *Web* di *CruiseControl*

Inviare lo stato del processo di *build* via *e-mail*

A volte può capitare di non poter accedere alla pagina *Web*, o di voler ricevere informazioni solo se il processo di *build* fallisce.

È quindi possibile configurare *CruiseControl* in modo da comunicare agli sviluppatori se il processo di *build* ha successo o fallisce.

Per far ciò è necessario aggiungere un elemento publisher all'interno dell'elemento `project` del file `builds/config.xml` come segue:

```
<htmlemail mailhost="SMTP_HOST"
        returnaddress="cruise@mydomain.com"
        buildresultsurl="http://localhost/cruisecontrol/buildresults/bowlingscore"
        css="/home/nicola/CC_HOME/reporting/jsp/webcontent/css/cruisecontrol.css"
        xsldir="/home/nicola/CC_HOME/reporting/jsp/webcontent/xsl"
        logdir="logs/bowlinscore">
    <always address="manager@mydomain.com"/>
    <failure address="dev1@mydomain.com"/>
    <failure address="dev2@mydomain.com"/>
</htmlemail>
```

Gli attributi dell'elemento `htmlemail` devono essere settati correttamente (in base alle configurazione del *server* di posta e alla locazione del progetto).

Nell'esempio specificato lo sviluppatore che ha l'indirizzo:
manager@mydomain.com
riceverà un *e-mail* sia che il processo di *build* del progetto è terminato con successo sia che sia fallito.

Gli altri sviluppatori riceveranno un'*e-mail* solo se il processo di *build* fallisce.

Per maggiori informazioni consultare la pagina:
`http://cruisecontrol.sourceforge.net/main/configxml.html#htmlemail`

6.2 Realizzare il sistema di integrazione continua con *Continuum*

Apache Continuum è un il sistema di *continuous integration* principalmente per progetti gestiti tramite *Maven*. Nelle prossime sezione verrà descritto come realizzare un'automazione ad evento, che permette di creare un processo di *continuous integration* per il progetto BowlingScore attraverso l'utilizzo di *Maven* e *Continuum*.

Installare *Continuum*

In questa sezione viene descritto come effettuare un'istallazione temporanea di *Continuum*.

Per maggiori informazioni su come effettuare un'istallazione in un *Web Server* nel modo desiderato è consigliato consultare il sito:
`http://continuum.apache.org/`

Per installare *Continuum* è necessario scaricare la versione desiderata dal sito:
`http://continuum.apache.org/download.html`
In questa sezione viene utilizzata la versione apache-continuum-1.1.

Dopo aver scaricato l'archivio contenente il prodotto, estrarlo nella *directory* desiderata (p.es. `/home/nicola/CONTINUUM`). Per avviare *Continuum* (nel sistema operativo *Linux*) è necessario eseguire i seguenti comandi:

```
cd /home/nicola/CONTINUUM/bin/linux-x86-32/
sh run.sh start
```

A questo punto è stato avviato *Continuum*. Per verificare se l'installazione e l'avvio di *Continuum* è andata a buon fine è necessario visitare il seguente indirizzo: `http://localhost:8080/continuum/`

Se tutto è andato a buon fine viene richiesto di registrare l'utente amministratore. Dopo aver compilato i campi per registrare l'utente amministratore è possibile accedere a *Continuum* con questo utente. A questo punto viene richiesto di compilare i campi per la configurazione generale (*"General Configuration"*). Salvare i campi proposti dall'installazione, lasciando vuoto il campo *"Deployment Repository Directory"*.

Inserire un progetto in *Continuum*

Per permettere a *Continuum* di effettuare il processo di *build* del progetto BowlingScore è necessario inserire delle informazioni aggiuntive al file *POM* del poggetto.

Aprire il file `/home/nicola/work/bowlingscore/pom.xml` del progetto BowlingScore e inserire i seguenti elementi:

```
  <ciManagement>
2 <system>continuum</system>
  <url>http://localhost:8080/continuum</url>
4 </ciManagement>

6 <scm>
      <connection>
8       scm:svn:file:///home/nicola/svn-repo/bowlingscore/trunk
      </connection>
10    <developerConnection>
      scm:svn:file:///home/nicola/svn-repo/bowlingscore/trunk
12    </developerConnection>
  </scm>
```

L'elemento `ciManagment` permette di configurare il sistema di *Continuous Integration*. È quindi necessario inserire i seguenti elementi che permettono di definire la configurazione di questo sistema.

L'elemento `scm` permette di definire e configurare il sistema di *source code managment* (sistema di gestione del codice sorgente). In questo elemento sono definite le modalità di concessione al sistema di versionamento. Dopo aver modificato il *POM* è necessario inviare queste modifiche al sistema di versionamento.

Accedere a Continuuom all'indirizzo:
`http://localhost:8080/continuum/`

Effettuare l'accesso con l'utente amministratore e inserire un nuovo progetto gestito da *Maven 2* (seguendo il collegamento *"Maven 2.0.X Project"*). A questo punto si aprirà una pagina che permetterà di inserire l'indirizzo di un *POM*.

Inserire il percorso assoluto del *POM* del progetto BowlingScore nel campo *"Upload Pom"* e premere il pulsante add, come rappresentato in figura 6.2 alla pagina 108

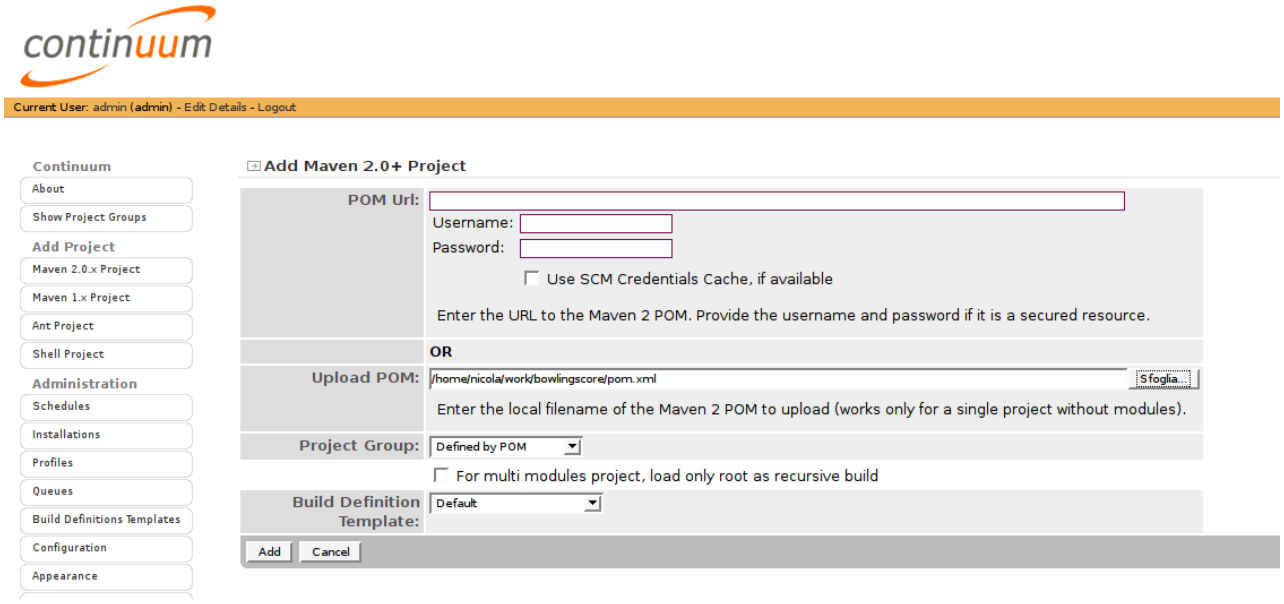

Figura 6.2: Caricamento di un *POM* in *Continuum*

In questo modo è stato inserito un progetto in *Continuum*.

Configurare il progetto

Avviare *Continuum* e collegarsi all'indirizzo:
`http://localhost:8080/continuum/`
Autenticarsi come amministratore e accedere alla sezione *"Show Project Groups"*.

In questa sezione verranno visualizzati tutti i progetti gestiti da *Continuum*. Accedere alla configurazione del progetto BowlingScore seguendo il collegamento che identifica il progetto.

A questo punto dovrebbe comparire la pagina *"Project Group"* dove vengono visualizzate le caratteristiche del progetto BowlingScore.

Accedere alla configurazione del progetto BowlingScore seguendo il collegamento che identifica il progetto nella sezione *"Member Projects"*.

A questo punto dovrebbe comparire la pagina "*Project Information*" dove vengono visualizzate le informazioni riguardanti il processo di *build* (*"Build Definition"*).

Programmare l'intervallo di esecuzione

Di norma *Continuum* è configurato in modo che il processo di *build* venga eseguito ogni ora, solo se sono state effettuate delle modifiche al progetto contenuto nel sistema di versionamento.

Per modificare la programmazione del processo di *build* è necessario creare un nuovo elemento di schedulazione. Per fare questo è necessario seguire il collegamento *"Scheudles"*.

In questo modo verrà visualizzata la pagina *"Shedules"* che permette di inserire e configurare un nuovo intervallo di schedulazione. Premere il pulsante *"Add"* e configurare un nuovo intervallo di configurazione come illustrato nella seguente figura 6.3 alla pagina 109.

- Name: MINUTE_ SCHEDULE
- Description: Run every minute
- Cron Expression:
  - Second: 0
  - Minute: 0/1

  - Hour: 0-23
  - Day of Month: *
  - Month: *
  - Day of Week: ?

- Maximum Job execute time: 3600
- Quiet period: 0

In questo modo è stato inserito un intervallo di schedulazione che permette di attivare un'automazione a comando ogni minuto.

È quindi necessario modificare il progetto bowlingscore gestito da *Continuum* in modo che venga controllato ogni minuto se sono stati effettuati dei cambiamenti

Accedere alla pagina *"Project Information"* del progetto bowlingscore e modificare l'elemento presente nella sezione *"Build Definitions"* seguendo il collegamento del pulsante *"edit"* (rappresentata dall'icona ingranaggio e matita).

A questo punto è possibile modificare il parametro *"Schedule"* del progetto e inserire l'intervallo di schedulazione appena creato, come viene rappresentato in figura 6.4 alla pagina 110.

Premere il pulsante *"Save"*.

D'ora in poi, *Continuum* controllerà ogni minuto se sono state fatte delle modifiche al progetto. Se vengono trovate le modifiche verrà eseguito un nuovo processo di *build* e i risultati verranno pubblicati nella pagina *''Builds"* del progetto BowlingScore.

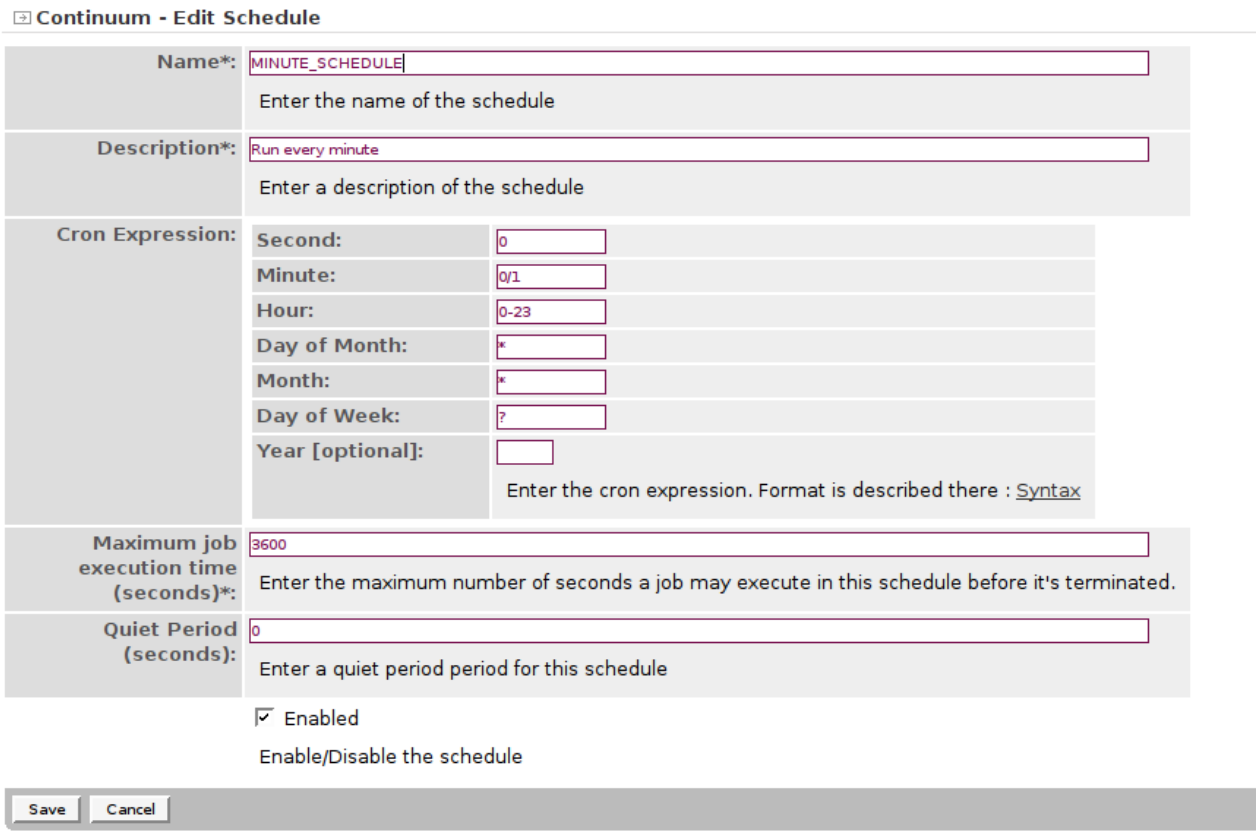

Figura 6.3: Configurazione di un elemento di schedulazione

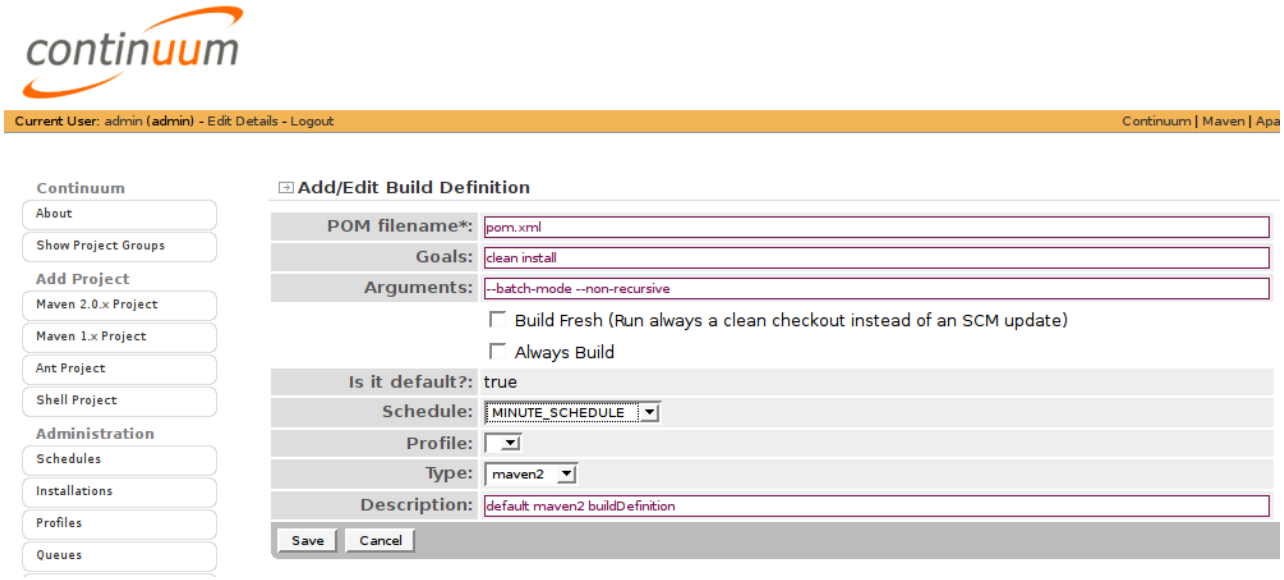

Figura 6.4: Modifica dell'intervallo di schedulazione del progetto

Pubblicare i risultati di *build* tramite *e-mail*

Risulta scomodo consultare i risultati del processo di *build* dall'applicazione *Web* di *Continuum*. Per questo *Continuum* permette di notificare lo stato del processo di *build* di un progetto in diversi modi (p.es. tramite messaggi di *IRC*, *Jabber*, *MSN* e *Google Talk*).

In questa sezione viene descritto come configurare *Continuum* in modo che i risultati di una *build* vengano notificati tramite un'*e-mail*.

Configurare *Continuum*

Per configurare *Continuum*, per inviare *e-mail* che notifichino lo stato del processo di *build*, è necessario effettuare le seguenti operazioni:

Fermare il *server* di *Continuum* tramite il seguente comando:

```
cd ~/CONTINUUM/bin/linux-x86-32/
sh run.sh stop
```

Aprire tramite un *editor* di testo il file `home/nicola/CONTINUUM/conf/plexus.xml` e modificare la risorsa mail/Session nel seguente modo:

Listing 6.4: home/nicola/CONTINUUM/conf/plexus.xml

```
1  ...
     <resource>
3        <name>mail/Session</name>
         <type>javax.mail.Session</type>
5        <properties>
           <property>
7            <name>mail.smtp.host</name>
             <value>smtp.gmail.com</value>
9          </property>
           <property>
11           <name>mail.smtp.port</name>
             <value>465</value>
13         </property>
           <property>
15           <name>mail.smtp.auth</name>
             <value>true</value>
17         </property>
           <property>
19           <name>mail.smtp.user</name>
             <value>username@gmail.com</value>
21         </property>
```

```
        <property>
23        <name>password</name>
          <value>password</value>
25      </property>
        <property>
27        <name>mail.smtp.starttls.enable</name>
          <value>true</value>
29      </property>
        <property>
31        <name>mail.smtp.socketFactory.class</name>
          <value>javax.net.ssl.SSLSocketFactory</value>
33      </property>
      </properties>
35  </resource>
...
```

In questo esempio è stato configurato *Continuum* in modo da utilizzare il servizio *SMTP* di *gmail*.

Modificare i campi in modo da impostare il servizio *SMTP* del *server* di posta desiderato (per utilizzare il servizio di *gmail* è necessario solo modificare i valori delle proprietà `password` e `username`).

Salvare le modifiche e riavviare *Continuum* con i seguenti comandi:

```
cd ~/CONTINUUM/bin/linux-x86-32/
sh run.sh start
```

Configurare i *Notifiers* del progetto

Accedere a *Continuum*, con l'utente amministratore, alla pagina del progetto BowlingScore. Premere il pulsante "*add*" nella sezione *"Notifiers"* e selezionare *notifier* di tipo *"Mail"*. Inserire l'indirizzo dell'utente a cui si vuole notificare lo stato del processo di *build* e selezionare lo stato da notificare (successo, errori, fallimento del processo di *build*). Premere il pulsante *Salva*. D'ora in poi, ogni volta che il processo di *build* terminerà nel modo selezionato, *CruiseControl* invierà un'*e-mail* all'indirizzo inserito.

Per testare se la configurazione è andata a buon fine è consigliato modificare un file del progetto BowlingScore e inviare le modifiche al sistema di versionamento.

Pubblicare la documentazione

Come descritto nella sezione 2.4.2 è molto importante eseguire degli strumenti che permettono di calcolare alcuni aspetti riguardanti la qualità del progetto.

Come si può capire, il sito di documentazione che contiene queste metriche, deve essere aggiornato ogni volta che avviene un cambiamento ai sorgenti del progetto presenti nel sistema di versionamento. È quindi consigliato inserire nel sistema di *continuous integration* anche la creazione della documentazione di un progetto.

Pubblicare la documentazione del progetto

Per pubblicare la documentazione di un progetto in un *server* prestabilito è necessario modificare il *POM* del progetto nel seguente modo:

Listing 6.5: bowlingscore/pom.xml

```
<project>
2   ...
  <distributionManagement>
4     <site>
        <id>website</id>
6       <url>file://localhost/TOMECAT_HOME/webapps/bowlingscore</url>
      </site>
8   </distributionManagement>
    ...
10 </project>
```

È stato inserito l'elemento `distributionManager`. Questo elemento permette di configurare tutti gli aspetti della distribuzione dei prodotti che possono venir creati in un progetto. In questo esempio è stata specificata l' URL dove verrà pubblicato il sito di documentazione del progetto.

Se viene eseguito il comando:

```
mvn site-deploy
```

Verrà creato e pubblicato il sito di documentazione del progetto nell'URL specificata.

Pubblicare la documentazione tramite un automazione ad evento

Per inserire la pubblicazione in *Continuum*, in modo che venga eseguita ogni volta che avviene un cambiamento al progetto, è necessario effettuare le seguenti azioni:

- Avviare *Continuum*
- Autenticasi come amministratore
- Accedere al progetto BowlingScore
- Premere il pulsante *"Add"* nella sezione *"Build Definition"*
- Configurare la sezione *Build Definition* come in figura 6.5
- Salvare le configurazioni

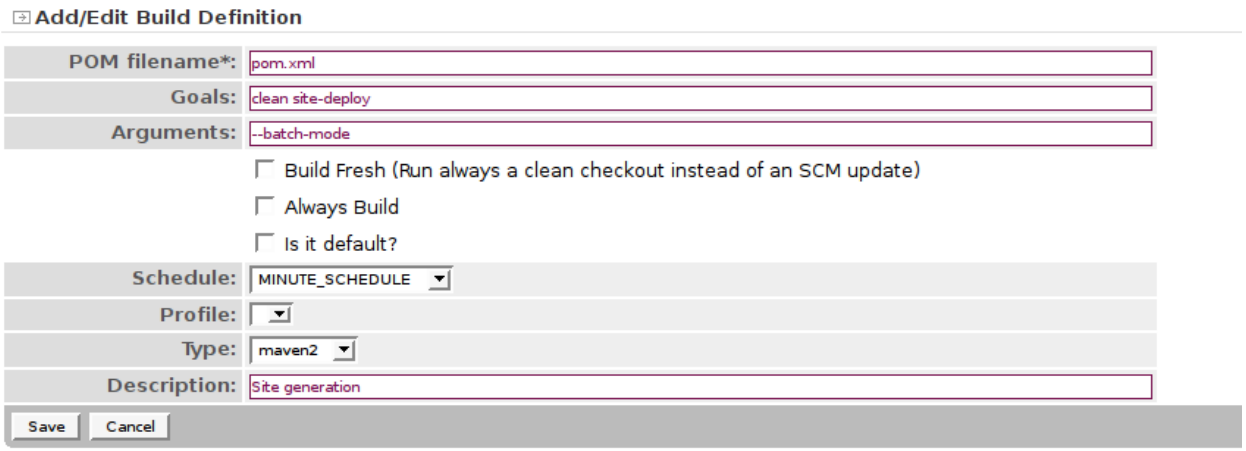

Figura 6.5: Generazione e pubblicazione della documentazione

In questo modo è stato aggiunto il processo di generazione e distribuzione del sito del progetto nel sistema di *Continuous Integration*.

Ogni volta che verrà inviata una modifica del codice del progetto, al sistema di versionamento, verrà aggiornato e pubblicato il sito di documentazione.

Capitolo 7

Test di sistema

In questo capitolo vengono descritte alcune tipologie di test di sistema. Differentemente dai test di unità, che hanno lo scopo di verificare il funzionamento delle componenti che compongono il progetto in un ambiente isolato, i test di sistema ha lo scopo di integrare due o più componenti che implementano le funzionalità del sistema e verificare che il comportamento e le caratteristiche sono come desiderate dallo sviluppatore e richieste dal cliente. In base alla criticità e all'importanza del prodotto da realizzare viene impiegato più o meno tempo in questa attività di test. Dato che è quasi impossibile realizzare un sistema perfetto questa attività ha lo scopo di convincere gli sviluppatori e i clienti che il sistema realizzato è pronto per l'uso operativo.

Esistono varie tipologie di test che permettono di verificare il comportamento del sistema, in questa parte della guida verranno descritti:

- I test di integrazione
- I test funzionali

Per motivi di poca esperienza da parte dell'autore sulle tematiche trattate in questo capitolo della guida, le informazioni che vengono riportate nelle successive sezioni sono meno approfondite dei precedenti capitoli e prendono spunto dal capitolo 23 di [13].

7.1 Test di integrazione

Come viene definito in [3] i test di integrazione sono:

> "*testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them*"

Il processo di integrazione del sistema consiste nella costruzione del prodotto combinando le componenti che lo compongono. Le tipologie delle componenti che compongono un sistema possono essere:

- Prefabbricate: queste componenti possono essere *hardware* o *software* esistenti che dopo una configurazione iniziale possono essere combinate con altre componenti per realizzare parti del prodotto (p.es. un *database*, un *Web container*, etc.);
- Riutilizzabile: queste componenti possono essere funzionalità fornite da una libreria esterna che vengono utilizzate per realizzare il prodotto;
- Sviluppate da zero: queste componenti sono combinazioni delle unità che vengono create per realizzare il prodotto;

Il test di integrazione ha lo scopo di verificare che le varie componenti che realizzano il sistema collaborino nella maniera desiderata. Per effettuare questa attività è necessario identificare dei sottoinsiemi di componenti che forniscono alcune funzionalità del sistema ed effettuare la loro integrazione tramite l'aggiunta di codice che permetta di realizzare la loro collaborazione.

L'integrazione delle componenti può avvenire principalmente in due modi:

Integrazione *top-down*: In questo tipo di integrazione viene realizzato lo scheletro del sistema e poi, uno ad uno, vengono aggiunti i componenti del sistema. In questo modo, durante l'integrazione, se viene trovato un errore questo sarà causato dall'integrazione dell'ultima componente.

Integrazione *Bottom-up*: In questo tipo di integrazione vengono integrate le componenti che forniscono i servizi comuni(p.es. accesso alla rete, accesso al *database*, etc.) successivamente vengono aggiunte le componenti funzionali fino a realizzare l'intero prodotto.

Il test di integrazione è un attività incrementale dove, a partire dallo scheletro del sistema, si aggiungono le componenti fino a verificare il comportamento dell'intero prodotto. Per poter eseguire il prodotto parziale (composto solo da una parte delle componenti) è necessario sviluppare del codice aggiunto per simulare le funzionalità delle componenti che non sono ancora state integrate. Il codice che permette di simulare il comportamento di una componente può essere di due tipi:

***Stub*:** Codice che permette di creare le funzionalità definite dall'interfaccia di una componente nel modo più semplice possibile. È una entità passiva che viene invocata dalle altre componenti del sistema. Gli *Stub* sono molto simile ai *Mock Object* utilizzati nei test di unità per rendere indipendente la verifica di una unità da altre parti del sistema. Come riportato in [12] a pagina 218, gli *Stub* hanno lo scopo di permettere al prodotto di eseguire anche se non sono stati integrati tutti i componenti. I *Mock Object* hanno lo scopo di simulare un aspetto di una componente e controllare in che modo la componente viene invocata dall'unità che si sta verificando (quante volte, quali metodi, etc.).

***Driver*:** Codice, programmi o strumenti che permettono di esercitare in modo controllato le componenti che compongono un sistema. È una entità attiva che permette di invocare e verificare il comportamento delle componenti che compongono il sistema. I metodi di test del *framework JUnit* sono dei *Driver* perché permettono di invocare e verificare il comportamento delle componenti di un sistema.

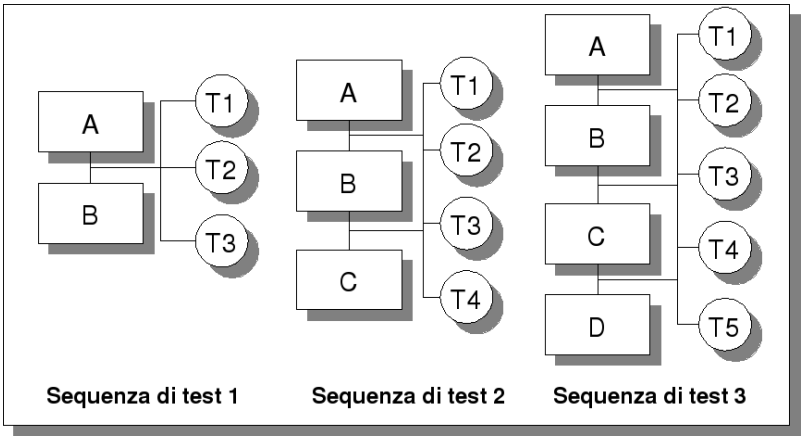

Figura 7.1: Test di integrazione incrementale tratta da [13] pagina 528

Nell'esempio mostrato dalla figura 7.1 tratta da [13], i componenti del sistema sono A, B, C e D mentre T1-T5 sono i relativi insiemi di test. Nella sequenza di test 1 viene testato il sistema minimo, composto dai componenti A e B, tramite l'esecuzione dei test T1-T3. In questa fase si cerca di verificare alcuni aspetti delle funzionalità offerte dalle componenti A e B e si cerca di verificare se l'integrazione tra le due componenti funziona come desiderato. Se per caso è impossibile effettuare il funzionamento del sistema minimo, perché ci sono delle dipendenze con le componenti C e D, verranno creati degli *Stub* che permetteranno di far compilare ed eseguire il sistema. Se vengono rilevati dei difetti nella sequenza di test 1 questi vengono corretti.

Nella sequenza di test 2 viene integrato il componente C nel sottosistema minimo, vengono ripetuti i test della sequenza di test 1 e aggiunti dei nuovi test per verificare che l'integrazione non introduca errori. In questo modo vengono sostituiti gli *Stub* presenti nella sequenza di test 1 con la componente reale C. Se vengono trovati dei difetti nella sequenza di test 2 questi devono essere identificati e corretti. Se le correzioni portano a modificare le componenti A e B è necessario eseguire nuovamente i test della sequenza 1. Infine viene aggiunto l'ultimo componente (D) e viene testato il sistema completo.

Per decidere l'ordine di integrazione è consigliato che il sottosistema minimo fornisca le funzionalità più importanti per il cliente. In questo modo queste verranno verificate dai test più frequentemente delle ultime componenti aggiunte. Nell'esempio è consigliato che le componenti abbiano un importanza proporzionale all'ordine alfabetico (A più importante D meno importante).

Come si può intuire uno dei principali problemi di questa attività è la localizzazione degli errori. A differenza dei test di unità, i test di integrazione non sono indipendenti quindi l'aggiunta di una nuova componente può produrre errori nel integrazione con le componenti già testate. È buona norma che rendere i test di integrazione automatici, in questo modo è più semplice e rapido eseguire tutti i test creati facilitando la localizzazione degli errori. È quindi consigliato creare i test di integrazione con gli stessi strumenti utilizzati per i test di unità e con gli strumenti descritti nella sezione dei test funzionali, in modo che questi siano ripetibili e automatici.

Confronto tra test di integrazione e integrazione continua

Nel capitolo 2 viene descritta la pratica di *Continuous integration*. Questa pratica consiste nell'integrare il lavoro degli sviluppatori ogni volta che viene effettuata una modifica al progetto. Come viene rappresentato dalla figura 2.2 b il processo di sviluppo viene guidato dai test di unità che vengono eseguiti automaticamente dal sistema di integrazione continua ogni volta che viene consolidata una modifica.

Poiché i test di unità hanno il compito di verificare in modo isolato le funzionalità atomiche del prodotto, tramite la pratica di integrazione continua viene verificato che il comportamento delle unità rimanga costante nel tempo e che gli errori corretti in passato non si ripresentino in futuro. Per questo motivo la pratica di integrazione continua non può sostituire l'attività di test di integrazione.

È buona norma che i test di integrazione possano essere eseguiti frequentemente in modo automatico. Se i test di integrazione vengono eseguiti, ad ogni modifica del progetto, dal sistema di integrazione continua, rallenterebbero il processo di sviluppo. Ad ogni modifica apportata dallo sviluppatore sarebbe necessario interrompere l'attività di sviluppo fino a che il sistema di integrazione continua non ha terminato l'esecuzione di tutti i test (di unità e di integrazione). Dato che i test di integrazione prevedono l'utilizzo di tutte le componenti che compongono il sistema, la velocità di esecuzione è più lenta rispetto ai test di unità. Per non ritardare eccessivamente l'attività di sviluppo è consigliato eseguire questa tipologia di test con una frequenza di esecuzione inferiore rispetto ai test di unità (p.es. a determinate ore della giornata lavorativa).

È quindi possibile programmare il sistema di integrazione continua in modo che esegua i test di integrazione a determinate ore del giorno. In questo modo, se vengono trovati degli errori, il sistema di integrazione continua avvertirà i programmatori che interromperanno l'attività di sviluppo finché questi non verranno corretti.

7.2 Test funzionali

Il test funzionale ha lo scopo di verificare se una versione del sistema, che si intende consegnare o presentare al cliente soddisfa i requisiti concordati. In questo modo si cerca di dare una prova al cliente che il sistema che si sta implementando funzioni come desiderato al fine di aumentare la fiducia sul prodotto.

Per dimostrare che il sistema soddisfa i requisiti concordati si deve dimostrare che questo fornisce le funzionalità, le prestazioni e l'affidabilità specificate, e che non fallisce la sua esecuzione durante il suo normale uso.

Tipicamente questa attività è un processo a scatola chiusa, dove i test sono derivati dalle specifiche del sistema. Come illustrato in figura 7.2, il comportamento del prodotto viene verificato fornendo solo dei parametri di *input* ed esaminando gli *output* corrispondenti. Se i parametri di *output* non sono quelli previsti, allora il test ha individuato un difetto nel prodotto.

Quando si verificano le funzionalità di una versione del prodotto si dovrebbe cercare di riprodurre sia il normale funzionamento del programma che le situazioni d'errore (rappresentate nella figura 7.2 da Ie). In questo modo viene verificato che il programma gestisca e segnali correttamente le situazioni d'errore (producendo gli *output* rappresentati nella figura 7.2 da Oe). In [13] vengono riportate alcune situazioni d'errore che possono aiutare a riprodurre alcuni problemi durante la fase di test:

- Scegliere gli *input* che forzano il sistema a generare tutti i messaggi di errore;
- Inserire gli *input* che portano all'*overflow* dei *buffer* di ingresso;

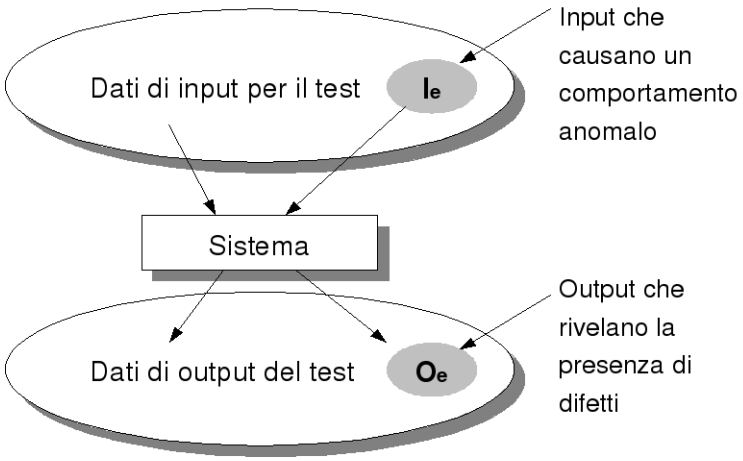

Figura 7.2: Test black-box tratta da [13] pagina 530

- Ripetere gli stessi *input* o serie di *input* diverse volte;
- Forzare la generazione di *output* non validi;
- Forzare i risultati del calcolo a essere troppo grandi o troppo piccoli;

Nel modello di sviluppo evolutivo/incrementale i test funzionali viene effettuata ad ogni iterazione del processo per verificare che la versione del prodotto realizzata sia una parte stabile dell'intero sistema e che fornisca le funzionalità richieste dal cliente. Negli approcci agili questa attività viene effettuata con il cliente dove viene assicurato che gli sviluppatori realizzino il prodotto come richiesto.

Come si può intuire è preferibile che questa attività coinvolga il cliente, in questo modo viene verificato che i requisiti funzionali siano stati interpretati nel modo giusto. Nelle prossime sezioni vengono forniti degli esempi su come realizzare questa tipologia di test utilizzando degli strumenti che permettono di coinvolgere attivamente il cliente in questa attività.

7.2.1 Realizzazione con *Fitnesse*

In questa sezione viene descritto come realizzare dei test funzionali attraverso *Fitnesse*. Per realizzare questa attività attraverso *Fitnesse* è necessario:

1. Installare *Fitnesse*
2. Definire delle tabelle con il cliente dove verranno specificati i dati di *input* e gli *output* da verificare
3. Creare un progetto che permetta al cliente e agli sviluppatori di verificare le funzionalità del programma
4. Creare una *suite* di test che permetta agli sviluppatori e al cliente di verificare le funzionalità del programma

Di seguito vengono descritte queste quattro fasi fornendo degli esempi sulla realizzazione dei test funzionali del progetto BowlingScore. In particolare vengono realizzati i test che permetteranno agli sviluppatori e al cliente di capire se la parte del progetto che simula una sezione di partita fornisca le funzionalità richieste.

Installazione di *Fitnesse*

Per installare *Fitnesse* è necessario accedere al sito `http://fitnesse.org/` e scaricare l'ultima versione disponibile del progetto (in questa guida viene utilizzata la versione 20070619).

Dopo aver scaricato l'archivio è necessario estrarre il progetto in una cartella (in questa guida è stato installato questo programma nella cartella `/home/nicola/fitnesse`).

Per eseguire *Fitnesse* è necessario eseguire i seguenti comandi:

```
cd /home/nicol/fitnesse
sh run.sh -p 9999
```

In questo modo viene eseguita una applicazione *Web* che permette di utilizzare *Fitnesse* (l'opzione -p permette di specificare la porta da dove sarà possibile accedere all'applicazione). Per accedere a *Fitnesse* è necessario aprire tramite un *browser* la pagina `http://127.0.0.1:9999/`. Se compare una pagina, come rappresentato in figura 7.3 significa che l'applicazione è stata installata e avviata con successo.

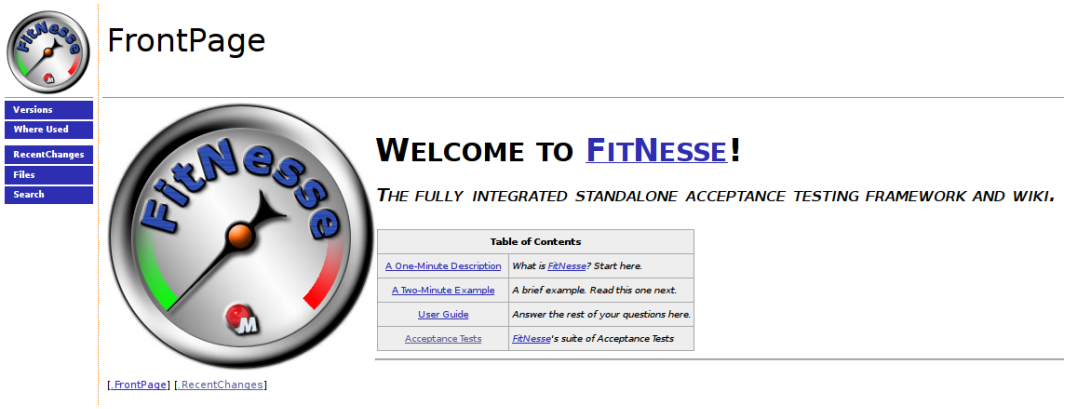

Figura 7.3: Schermata iniziale di *Fitnesse*

Da questa pagina è possibile accedere a molta documentazione e molti esempi utili per realizzare i test funzionali con questo strumento.

Definizione delle tabelle per specificare i dati di test

Per permettere di specificare i dati dei test funzionali di un programma è necessario concordare con il cliente come questi devono essere espressi. Solitamente i dati di *input* e i risultati di *output* dei test vengono specificati attraverso delle tabelle che verranno utilizzate dagli sviluppatori e dal cliente per verificare e analizzare i dati prodotti dalle funzionalità del programma.

Si supponga di concordare con il cliente una tabella per specificare i dati per verificare la parte del progetto BowlingScore che simula un'istanza di una partita. Una possibile tabella che potrebbe essere utilizzata in questo esempio è quella rappresentata nella tabella 7.1. In questa tabella sono specificati i seguenti elementi:

pin	**times**	**roll**	**isFinish**
1	19	true	false
0	1	true	true
score is	19		

Tabella 7.1: Esempio di tabella dove vengono specificati i dati dei test funzionali

- Nella prima colonna (`pin`) vengono indicati il numero di birilli che vengono colpiti in un lancio da un giocatore.
- Nella seconda colonna (`times`) vengono indicate il numero di volte consecutive in cui il giocatore colpisce i birilli specificati dalla colonna `pin`.
- Nella terza colonna (`roll`) viene specificato se la serie di lanci non ha causato problemi.
- Nella quarta colonna (`isFinish`) viene specificato se l'istanza di gioco è conclusa.
- Nell'ultima riga (`score is`) della tabella viene specificato il punteggio della partita.

Nella tabella dell'esempio viene descritta una partita dove il giocatore colpisce per 19 volte consecutive 1 birillo. Al ventesimo lancio colpisce 0 birilli. Nell'ultima riga viene controllato che il punteggio della partita specifica sia 19.

Tramite questa tabella il cliente e gli sviluppatori potranno specificare i valori di *input* e di *output* dei test funzionali che si possono effettuare in un'istanza di gioco del progetto BowlingScore.

Creazione del codice di test

Per permettere di effettuare dei test funzionali tramite *Fit* e *Fitnesse* è necessario specificare del codice (che viene chiamato *fixture*) che permetta di verificare il comportamento di un progetto prendendo come *input* i valori specificati in una tabella.

Visto che questo codice deve accedere al prodotto realizzato, è consigliato creare un nuovo progetto che abbia come dipendenza il progetto BowlingScore. Per far questo è possibile creare un nuovo modulo del progetto BowlingScore (come spiegato nel capitolo 4.13 del libro [16]) oppure creare un progetto separato. In questa guida si è scelto di creare un progetto separato attraverso *Maven* (come spiegato nella sezione 5.4.1). È quindi necessario creare un nuovo progetto (bowlingscore-functional-test) che conterrà solo i test funzionali del progetto BowlingScore.

```
mkdir /home/nicola/temp
cd /home/nicola/temp
mvn archetype:create -DgroupId=it.unipd.app -DartifactId=bowlingscore-functional-test
```

Per permettere a questo progetto di accedere alle classi di BowlingScore e accedere alle librerie di *Fitnesse*, che permettono di creare il codice di test, è necessario modificare il *POM* del progetto nel seguente modo.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>it.unipd.app</groupId>
  <artifactId>bowlingscore-functional-test</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>bowlingscore-functional-test</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>it.unipd.app</groupId>
      <artifactId>bowlingscore</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.fitnesse</groupId>
      <artifactId>fitnesse</artifactId>
      <version>20070619</version>
    </dependency>
    <dependency>
      <groupId>org.fitnesse</groupId>
      <artifactId>fitlibrary</artifactId>
      <version>20070619</version>
    </dependency>
  </dependencies>
</project>
```

Come si può osservare sono state specificate le seguenti dipendenze:

- Righe 11-15: Il progetto BolingScore. In questo modo i test funzionali potranno accedere alle classi di questo progetto;
- Righe 16-25: Le librerie e le funzionalità offerte da *Fitnesse* che permettono di creare il codice necessario per realizzare i test funzionali;

A questo punto è possibile realizzare il codice che permetterà di analizzare i valori come specificato nella tabella 7.1. Le librerie di *Fit* e *Fitnesse* mettono a disposizione delle funzionalità che permettono di analizzare i dati riportati nelle tabelle di test. Le descrizioni di queste funzionalità possono essere trovate nella pagina `http://127.0.0.1:9999/FitNesse.TestTableStyles` della documentazione fornita con l'installazione di *Fitnesse*. Nell'esempio, per analizzare i dati specificati nel formato della tabella è stato scelto di utilizzare le classi di utilità:

- `fit.ColumFixture`: I valori di *input* e di *output* di un test sono specificati per righe. L'intestazione di ogni colonna rappresenta un metodo o un'attributo pubblico della classe che assume i valori specificati.

- `fitlibrary.DoFixture`: Permette di eseguire un metodo e controllare che questo ritorni il valore booleano `true`.

Le classi realizzate sono illustrate nella figura 7.4:

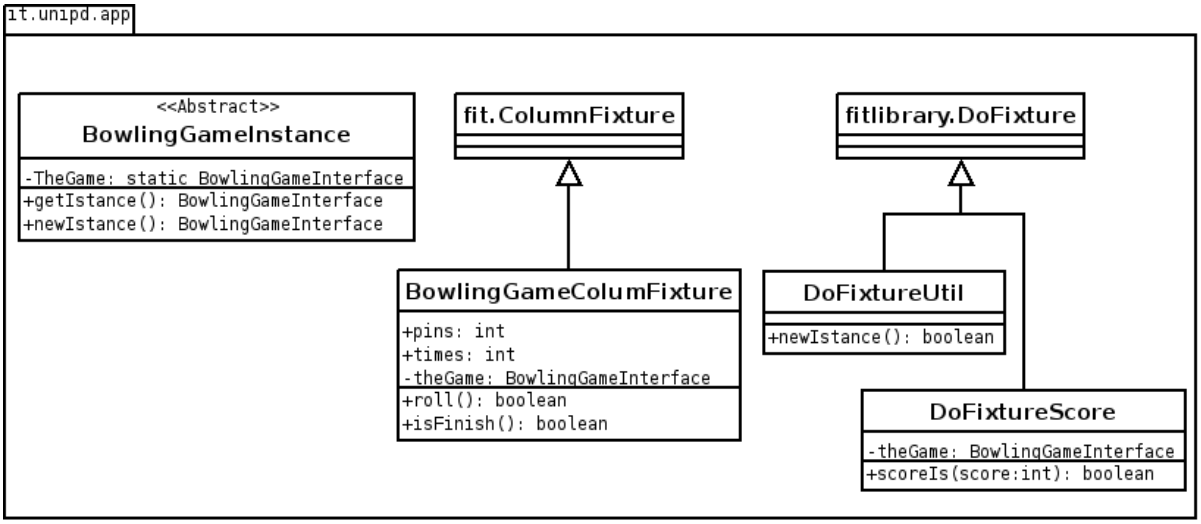

Figura 7.4: Diagramma delle classi del codice dei test funzionali

Il codice che permette di leggere i valori di *input* e analizzare i valori di *output* specificati in tabelle come quella dell'esempio 7.1 consiste nei seguenti file:

- `BowlingGameInstance`: Classe astratta statica che permette di creare un unica istanza della classe `BowlingGame` del progetto BowlingScore. Questa classe implementa il *pattern singleton* che permette di avere un unica istanza di una classe (in questo caso in tutta la sezione di un test delle funzionalità). Questa classe espone due metodi:

  - `getIstance`: ritorna una istanza della classe `BolingGame`.
  - `newIstance`:crea una nuova istanza della classe `BolingGame` per simulare una nuova partita.

- `BowlingGameColumFixture`: Classe che deriva da `ColumFixture` e contiene i metodo che permettono di simulare i lanci di una partita e verificare se la partita è terminata. tramite questa classe, specificando i valori come nella tabella 7.1 sarà possibile verificare il comportamento della classe che simula una partita.

- `DoFixtureScore`: Classe che deriva da `DoFixture` e permette di verificare il punteggio di una partita conclusa.

- `DoFixtureUtil`: Classe che deriva da `DoFixture` e permette di ricreare una nuova istanza di una partita.

Tramite queste classi è possibile verificare le funzionalità della parte del progetto che simula una sessione di una partita di bowling. Se si è scelto di creare il progetto che contiene il codice dei test funzionali attraverso *Maven* è necessario inserire questo progetto nel *repostitory* locale di *Maven* tramite il comando:

```
cd /home/nicola/temp
mvn install
```

In questo modo le classi potranno essere utilizzate da *Fit* e *Fitnesse* per creare ed eseguire i test funzionali.

Realizzazione i test funzionali attraverso *Fitnesse*

Per dimostrare al cliente che il prodotto soddisfi i requisiti stabiliti è necessario creare attraverso *Fitnesse* delle pagine che permettano di eseguire e modificare i dati dei test precedentemente creati.

Per permettere di organizzare in modo corretto i test di un progetto è consigliato creare una *suite* di test. In questo modo è possibile specificare una sola volta le parti in comune di tutti i test di un progetto (come il `classpath` e i metodo i setUp e TearDown). Per far una *suite* di test è necessario:

- avviare *Fitnesse*
- inserire nella barra dell'indirizzo del browser che si sta utilizzando il nome della pagina che si vuole creare (in questo caso `http://127.0.0.1:9999/SuiteBowlingScore`) che deve iniziare con la parola *Suite*
- seguire il collegamento *create this page*

Se il nome della pagina inizia con il nome *Suite Fitnesse* riconosce che questa è una *suite* di test. In questo modo la pagina conterrà tutte le funzionalità necessarie per eseguire tutti i test contenuti nella *Suite*. Inserire nello spazio create il seguente codice:

```
!2 ''Classpaths''
!path classes
Directory contenente le classi necessarie per eseguire i test
!path /home/nicola/.m2/repository/it/unipd/app/bowlingscore/1.0-SNAPSHOT/*.jar
!path /home/nicola/.m2/repository/it/unipd/app/bowlingscore-functional-test/1.0-SNAPSHOT/*.jar

---
!2 ''Utilities''
 * SuiteBowlingScore.PageHeader
 * SuiteBowlingScore.PageFooter
 * SuiteBowlingScore.SetUp
 * SuiteBowlingScore.TearDown
 * SuiteBowlingScore.TestBowlingGame
---
```

E premere il pulsante *Save*. In questo modo è stata creata la *suite* di test per il progetto BowlingScore. Tramite gli elementi `!path` è stato specificato il `classpath` di tutti i test che saranno contenuti in questa *suite*. Come si può notare sono state incluse sia le classi del progetto BowlingScore che del progetto bowlingscore-functional-test (contenti le classi illustrate nella figura 7.4). Successivamente è stato creato un elenco delle pagine che fanno parte della *suite* di test:

- La pagina `Suite.BowlingScore.PageHeader` e `Suite.BowlingScore.Footer` permette di specificare il testo che viene scritto in alto e in basso di ogni pagina contenuta nella *suite*.
- La pagina `Suite.BowlingScore.SetUp` e `Suite.BowlingScore.TearDown` permette di specificare le azioni che devono essere eseguite prima e dopo ogni esecuzione dei test specificati nella *suite*.
- La pagina `SuiteBowlingScore.TestBowlingGame` conterrà il test dove verranno specificati i dati come riportato nella tabella 7.1.

Come si può osservare, ogni elemento dell'elenco appena descritto è caratterizzato da un collegamento ?. Premendo questo collegamento è possibile creare la pagina selezionata. Se nell'elenco delle pagine era presente una pagina che iniziava con la parola *Suite* era possibile specificare una sotto *suite* composta da test che avevano caratteristiche comuni dove era possibile specificare altre risorse comuni (p.es aggiungere altre librerie al `classpath` e altre pagine di *SetUp* e *ThearDown*).

Per realizzare il test con i dati specificati nel formato della tabella 7.4 non sono necessarie operazioni prima e dopo i test. In questo caso è possibile creare subito la pagina `SuiteBowlingScore.TestBowlingGame` senza la necessità di creare le altre pagine.

Premere il collegamento ? di `SuiteBowlingScore.TestBowlingGame` e inserire il seguente codice:

```
!| it.unipd.app.BowlingGameColumFixture|
| pin| times| roll?|isFinish?|
| 1  | 19   | true | true    |
| 0  | 1    | true | true    |
```

```
6 !|it.unipd.app.DoFixtureScore|
  |score is|19|
8
  !|it.unipd.app.DoFixtureUtil|
10 |restart game|

12
  !| it.unipd.app.BowlingGameColumFixture|
14 | pin| times| roll?|isFinish?|
  | 0  | 20   | true | true    |
16
  !|it.unipd.app.DoFixtureScore|
18 |score is|0|
```

Come si può capire l'applicazione *Web* di *Fitnesse* consiste in una *wiki* dove è possibile inserire del testo e delle tabelle di test. Come si può vedere nelle righe 1-4 vengono specificati i valori di una partita come riportato nella tabella 7.1. Nella riga 1 viene specificata la classe che contiene i metodo che verranno utilizzati nella tabella per realizzare il test. Nella seconda riga è presente l'intestazione della tabella dove vengono specificate le variabili e i metodi che vengono utilizzati per effettuare il test.

Per permettere di salvare le tabelle inserite è necessario premere il pulsante *Save*. A questo punto comparirà la pagina illustrata in figura 7.5.

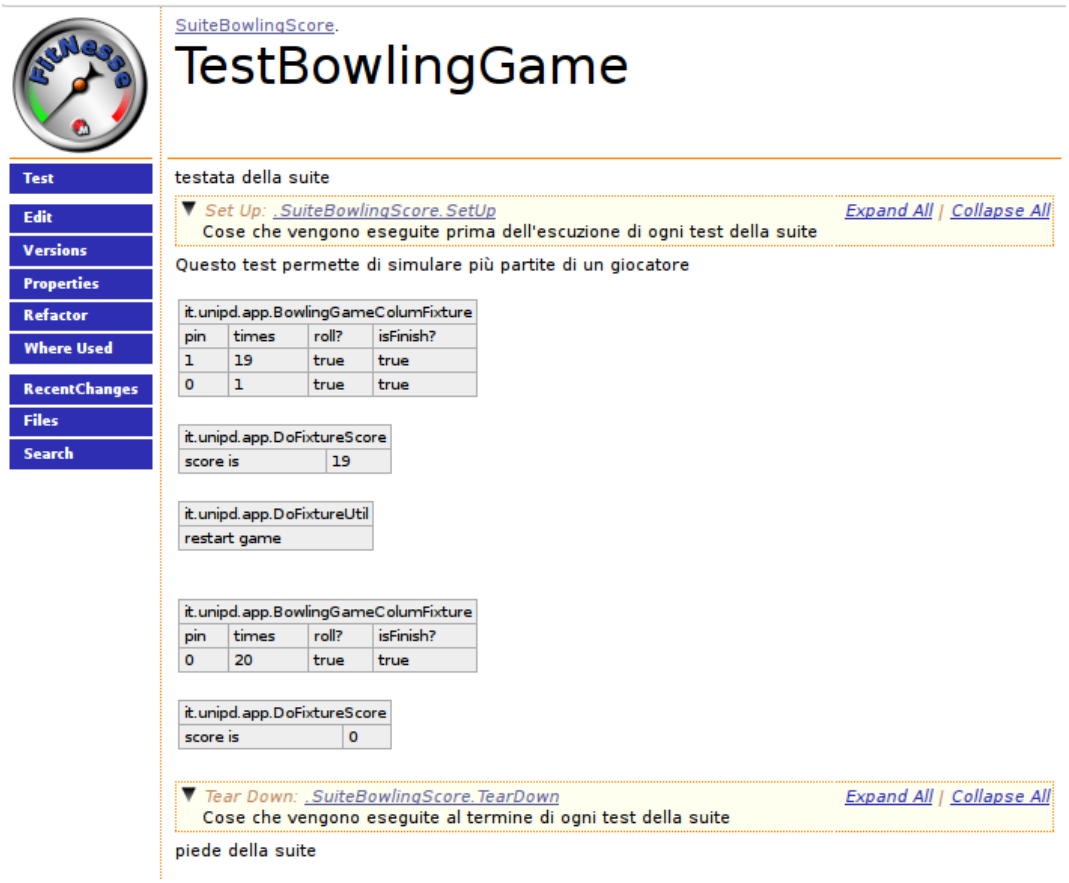

Figura 7.5: Pagina SuiteBowlingScore.TestBowlingGame

Come si può osservare, tramite il codice specificato sono state create due tabelle che permettono di simulare una partita con i dati dell'esempio 7.1 e una partita dove non viene mai colpito un birillo.

A questo punto, premendo il pulsante *test* verrà eseguito il test e verranno segnalate in verde le parti della tabella dove i valori specificati sono come desiderato, in giallo dove viene lanciata un'eccezione e in rosso dove vengono trovati degli errori.

Se vengono eseguiti i test con i valori specificati precedentemente verrà segnalato un errore. Come si può vedere nella figura 7.6 è stato specificato che dopo 19 lanci la partita deve essere terminata. Fortunatamente

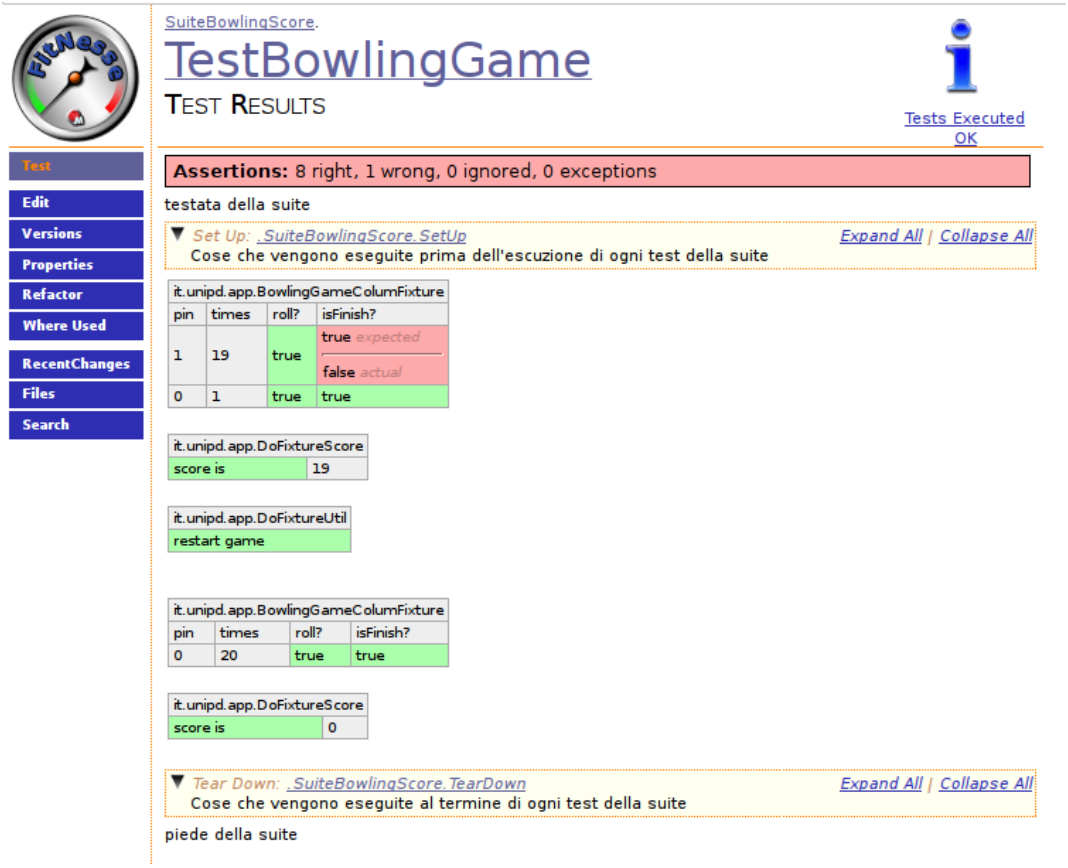

Figura 7.6: Segnalazione di un errore nella pagina SuiteBowlingScore.TestBowlingGame

questo è un errore nei dati di test. Per correggere l'errore basta premere il pulsante *edit* e modificare la cella della tabella che riporta l'errore. Dopo aver corretto l'errore è possibile rieseguire il test premendo nuovamente il pulsante *test*. A questo punto tutti i possibili valori che possono essere controllati attraverso il test saranno colorati di verde. Premendo il pulsante *edit* della pagina `SuiteBowlingScore.TestBowlingGame` sarà possibile modificare o inserire nuovi test.

Come si può intuire *Fitnesse* è uno strumento molto utile per effettuare i test funzionali. Dispone di una applicazione *Web* che consiste in una *wiki* dove è possibile creare e modificare i test funzionali di un programma. In questo modo se la *wiki* viene resa pubblica è possibile coinvolgere il cliente nella fase di test. Aggiungendo delle tabelle ed eseguendo i test il cliente e gli sviluppatori possono controllare che le funzionalità di un'applicazione sono state realizzate correttamente. I test possono essere salvati in modo che altri successivamente possano accedervi e possono essere eseguiti in modo automatico attraverso una *suite* dove viene prodotta una pagina di riepilogo dove vengono riportati i risultati dei test.

Come si è potuto vedere è stato possibile creare dei test funzionali di un prodotto ancora in fase di sviluppo. In questo modo, soprattutto nei modelli di sviluppo evolutivi/iterativi, è possibile coinvolgere e presentare al cliente le funzionalità un pò alla volta fino alla realizzazione completa del prodotto. In questo modo è più probabile che il programma venga verificato in modo esaustivo e sarà più semplice realizzare un prodotto che soddisfi le aspettative del cliente.

Bibliografia

[1] Dijkstra, Edsger W. Structured Programming, 1969. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF

[2] Wilson, James Q. and Kelling, George. The police and neighborhood safety. The Atlantic Monthly, 249(3):29-38, March 1982.

[3] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990.

[4] The Standish Group. The CHAOS report, 1994. http://net.educause.edu/ir/library/pdf/NCP08083B.pdf

[5] The Standish Group. CHAOS: A recipe for success, 1999. http://www.ii.metu.edu.tr/~is526/course_material/papers/ChaosReport1998.pdf

[6] Flower, Martin and Beck, Kent and Brant, John and Opdyke, William and Roberts, Don. Refactoring: Improving the Design of Existing Code. Addison Wesley Lingman, Reading, 1999.

[7] Hunt, Andrew and Thomas, David. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, Reading, MA, 2000.

[8] Beck, Kent. Test Driven Development: By Example. Addison-Wesley Professional, 2002.

[9] Meszaros, G. and Smith, S. M. and Andrea, J. The Test Automation Manifesto. Extreme Programming and Agile Methods – XP/Agile Universe 2003, September 2003.

[10] Hunt, Andrew and Thomas, David. Pragmatic Unit Testing In Java with JUnit. The Pragmatic Programmers, 2003.

[11] Clark, Mike. Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps. The Pragmatic Programmers, 2003.

[12] Astels, David. Test Driven Development: A Practical Guide. Prentice Hall PTR, 2003.

[13] Sommerville, Ian. Ingegneria del software. settima edizione. Pearson Addison wesley.

[14] Mason, Mike. Pragmatic Version Control Using Subversion. Pragmatic Bookshelf, 2005.

[15] Spacco, Jaime and Strecker, Jaymie and Hovemeyer, David and Pugh, William. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In Proceedings of the Mining Software Repositories Workshop (MSR 2005), St. Louis, Missouri, USA, May 2005.

[16] Casey, John and Massol, Vincent and Porter, Brett and Sanchez, Carlos and Van Zyl, Jason. Better Builds with Maven. How-to Guide for Maven 2.0. Mergere Library Press, 2006. http://www.exist.com/

[17] IEEE Software. Test-Driven Development, Volume 24 Number 3, May | June 2007.

[18] Sanchez, Julio Cesar and Williams, Laurie and Maximilien, E. Michael, On the Sustained Use of a Test-Driven Development Practice at IBM, AGILE 2007 (AGILE 2007), 2007.

[19] IEEE Software. Quality Requirements, Volume 25 Number 2, March | April 2008.

Riepilogo strumenti

Ant	`http://ant.apache.org/`
Checkstyle	`http://checkstyle.sourceforge.net/`
Cobertura	`http://cobertura.sourceforge.net/`
Continuum	`http://continuum.apache.org/`
CruiseControl	`http://cruisecontrol.sourceforge.net/`
EasyMock	`http://www.easymock.org/`
Fitnesse	`http://www.fitnesse.org/`
Fit	`http://fit.c2.com/`
Grand	`http://www.ggtools.net/grand/`
JUnitPerf	`http://clarkware.com/software/JUnitPerf.html`
JUnit	`http://www.junit.org/`
Maven	`http://maven.apache.org/`
PMD	`http://pmd.sourceforge.net/`
Subversion	`http://subversion.tigris.org/`
SvnAnt	`http://subclipse.tigris.org/svnant.html`