UNIVERSITÀ DEGLI STUDI DI PARMA
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Automatizzazione e controllo delle soluzioni per i test dello strumento TCK di JSR331

Relatore: **Chiar.mo Prof. Gianfranco Rossi**
Candidato: **Riccardo Zangrandi**

Anno Accademico 2012/2013

A mia mamma Maria, mio papà Domenico e mia nonna Carla.

Ringraziamenti

Prima di addentrarsi negli argomenti che sono stati sviluppati nel lavoro di tesi è doveroso fare un ringraziamento ad alcune persone.

Un grazie particolare va al prof. Gianfranco Rossi che oltre ad avere avuto il ruolo di relatore è sempre stato disponibile e i suoi consigli sono stati preziosi per portare a termine questo lavoro.

Un ulteriore ringraziamento deve andare a Fabio Biselli che si è sempre prestato a aiutarmi con il materiale necessario a terminare il lavoro di tesi con disponibilità e serietà.

Inoltre intendo ringraziare tutti i professori che in questi anni mi hanno aiutato nel percorso di studi intrapreso facendo nascere in me un interesse sempre più crescente per le materie del corso di laurea.

Un altro grazie intendo rivolgerlo a tutti i miei compagni di corso che mi sono sempre stati vicino negli anni di studi.

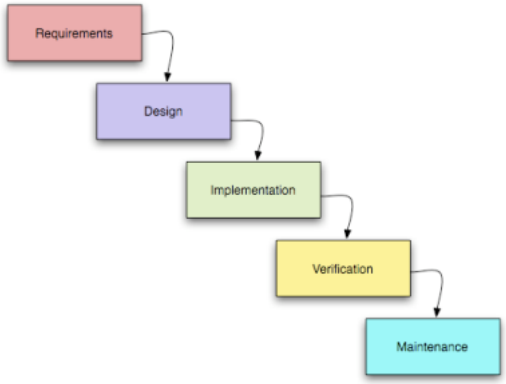
Indice

Capitolo 1

Introduzione

La fase di testing nel ciclo di vita di una qualunque libreria o applicazione software ricopre un ruolo molto importante. Infatti, qualunque sia il modello di sviluppo che si vuole adottare, a cascata, incrementale o evolutivo, la fase di validazione (o testing) prima della messa in uso è inevitabile. Il risultato

Figura 1.1: Modello di sviluppo a cascata



più desiderabile è la garanzia assoluta che tutti gli utenti del programma siano sempre soddisfatti dal comportamento del programma in validazione. Come si può immaginare questo risultato non è ottenibile. L'obiettivo migliore che si può sperare di raggiungere è che un programma soddisfi le specifiche per le quali è stato concepito e creato.

Solitamente esistono due strade per la validazione che possono essere seguite a seconda della complessità dell'applicazione da testare.

Un primo modo di procedere è quello di cercare di argomentare che il programma funziona per tutti i possibili input. Questa attività richiede un ragionamento sul suo codice ed è chiamata verifica. La verifica formale è

troppo pesante senza l'aiuto di procedure automatizzate e gli strumenti a disposizione oggi sono poco efficienti. Perciò la maggior parte della verifica è ancora fatta in modo piuttosto informale ed è un processo difficile.

L'alternativa a questo procedimento è il testing. Possiamo facilmente convincerci che un programma funziona su un insieme di input eseguendolo su ciascun membro dell'insieme e controllando i risultati. Se tale insieme di input possibili è piccolo un testing esaustivo è possibile. Tuttavia per la maggior parte dei programmi la totalità dei casi possibili è così grande che un testing completo è impossibile. Nonostante ciò un insieme ben scelto di casi di test può accrescere la nostra fiducia che il programma funzioni come richiesto. Un testing ben fatto può rilevare e scoprire la maggior parte degli errori di un programma.

Tale procedimento è adottato per la maggior parte dei progetti di grosse dimensioni e la sua praticità e flessibilità gli ha permesso di essere adottato ovunque.

Il processo di testing, inoltre, avviene a diversi livelli in base al procedimento di creazione del software e al modello di sviluppo adottato. In un primo momento si fa la validazione delle diverse unità, ovvero dei semplici algoritmi che compongono i diversi moduli. Successivamente si procede con il testing d'integrazione che ha lo scopo di verificare la correttezza del programma complessivo e l'assenza di anomalie sulle interfacce tra i moduli. Nel fare questo procedimento si seguono due approcci: *non incrementale*, dove tutte le unità precedentemente testate sono unite insieme e testate nel loro complesso analizzando così il sistema globalmente. Nell'approccio *incrementale*, invece, si parte testando i singoli moduli, collegandoli poi con i moduli chiamanti o chiamati, testando il sottosistema ottenuto, così via fino a costituire il sistema complessivo.

Successivamente bisogna testare le funzionalità dell'intero sistema attraverso diverse tipologie di testing complementari.

Test di stress (overload): si vuole verificare non solo che il programma funzioni secondo le specifiche, ma anche che si comporti in modo corretto in condizioni di carico di lavoro eccezionale. Per esempio: un sistema per basi di dati normalmente viene interrogato in modo tale da produrre venti transazioni per unità di tempo.

Test di sicurezza: il sistema può essere usato in condizioni non corrette, ad esempio si sottopone a delle violazioni, anche di tipo accidentale.

Test di robustezza: si forniscono al sistema dei dati sbagliati, e si osserva il comportamento del sistema rispetto a tali dati (esempio tipico:

si digitano sequenze casuali di tasti sulla tastiera per controllare se l'interfaccia utente di un programma si blocca).

Una volta terminate queste prime pratiche di testing, principalmente condotte all'interno dello stesso ambiente in cui si scrive il codice dell'applicazione da validare, si passa al test di accettazione. Nel test di accettazione il software viene confrontato con i requisiti dell'utente finale. Questo test e' normalmente svolto dal cliente. Una caratteristica del test di accettazione è che viene usualmente svolto senza avere a disposizione il codice sorgente. Per prodotti di largo consumo si utilizzano i concetto di alfa-test, in cui il software viene già usato all'interno della casa produttrice per verificarne le funzionalità, però ancora non è stato rilasciato all'esterno e beta-test, in cui il software viene rilasciato a un numero selezionato di utenti che lo usano sapendo che non è una versione stabile, e che possono interagire con chi ha prodotto il software.

Quando si parla di librerie software possono esistere due tipologie di testing che prendono nomi differenti in base allo scopo che si propongono di avere:

- **Test di compatibilità:** questa tipologia di test viene creata da coloro che propongono nuove specifiche e regolamentazioni rispetto ad una tipologia di programmazione. La suite di test viene inserita in uno strumento detto *TCK*[5] (Technology Compatibility Kit) e ogni nuova implementazione della specifica può dichiararsi in linea con le direttive se supera correttamente i test messi a disposizione. Quindi lo scopo di effettuare una validazione di questo tipo è verificare il grado di aderenza ad un determinato standard di programmazione.

- **Test del prodotto:** a differenza dei test precedenti questi ultimi, oltre ad aver come obiettivo la verifica della correttezza delle procedure presenti nell'applicazione, vogliono anche misurare in termini di occupazione di memoria e tempi di esecuzione le prestazioni di un determinato prodotto software. Le suite di test che fanno capo a questa categoria molto spesso vengono create proprio da coloro che creano anche l'applicativo che si intende testare.

L'attività di testing ovviamente non termina mai con la messa in uso sul mercato del software, anzi lo segue nel suo ciclo di vita con lo scopo di evidenziarne i difetti per poter così apportare modifiche essenziali e che ne migliorino il funzionamento.

Per quanto riguarda ciò che viene trattato in questo testo si farà riferimento allo standard *Java JSR-331*[14][1] e al suo strumento di test *TCK*.

Tale standard è una *richiesta per specifiche* Java in fase di sviluppo sotto le regole redatte dal Java Community Process. Queste specifiche definiscono le API per la programmazione a vincoli.

La specifica JSR-331 risponde alla necessità di ridurre i costi associati all'incorporazione di risolutori di vincoli (come JSetL, Choco, etc.) ad applicazioni commerciali e non, operanti nel mondo reale. Esiste già un certo numero di fornitori di queste API, come già accennato possiamo ricordare JSetL, Choco, JaCoP ed altri. Tuttavia le differenze tra questi sono abbastanza significative da causare gravi difficoltà di utilizzo per gli sviluppatori di software.

La standardizzazione della programmazione a vincoli si prefigge come scopo quello di rendere tale tecnologia più accessibile per gli sviluppatori. Avendo un'interfaccia unificata sarà possibile, per i programmatori, modellare il problema in modo tale da poter provare la soluzione con più CP solver. Questo minimizza la dipendenza da fornitori specifici, ma allo stesso tempo non limita la possibilità di quest'ultimi nel procedere con lo sviluppo del solver.

Gli obiettivi delle specifiche sono:

- facilitare l'inserimento della tecnologia basata sui vincoli nelle applicazioni Java;
- aumentare la comunicazione e la standardizzazione tra i vari fornitori di CP solver;
- incoraggiare il mercato delle applicazioni basate sulla programmazione a vincoli e dei suoi strumenti mediante la standardizzazione delle suddette API;
- facilitare l'integrazione di tecniche basate sui vincoli in altri JSR per supportare la programmazione dichiarativa;
- rendere le applicazioni Java più portabili tra vari fornitori di risolutori di vincoli;
- fornire un modello per l'implementazione ed il supporto di librerie di vincoli e strategie di ricerca per diverse applicazioni basate sulla programmazione a vincoli;
- supportare i fornitori di solver offrendo API che vadano incontro alle loro necessità e che siano di facile implementazione.

La specifica è rivolta principalmente a tre soggetti:

- aziende che utilizzano le CP API per sviluppare applicazioni di supporto a decisioni in ambito industriale;

- fornitori di risolutori di vincoli che intendono sviluppare e mantenere la propria implementazione delle CP API;

- ricercatori in ambito di programmazione a vincoli che vogliono fornire o arricchire librerie di vincoli standard, algoritmi di ricerca e problemi concreti che vengano mantenuti dalla CP community.

Lo scopo della specifica JSR-331 è di definire un'interfaccia semplice da utilizzare, leggera e che costituisca uno standard per acquisire ed utilizzare risolutori di vincoli.

La specifica è mirata a piattaforme basate su Java ed è compatibile con JDK 1.5 o successivi.

L'ambito della specifica segue un approccio minimalista, con particolare cura alla facilità d'uso. Ricopre i più comuni concetti dei problemi con vincoli e della loro rappresentazione che è ormai già diventata una standardizzazione di fatto, adottata dai solver e negli articoli scientifici. Tale ambito è allo stesso tempo sufficientemente ampio da permettere agli sviluppatori di applicazioni l'utilizzo delle interfacce standard per modellare e risolvere tipici problemi di soddisfacimento di vincoli, all'interno dei domini più comuni in ambito aziendale, come la pianificazione, l'allocazione delle risorse e la configurazione.

All'interno dello strumento TCK di JSR331 sono già presenti tutti i test che devono essere superati al fine di poter dichiarare una determinata implementazione corretta secondo le specifiche dello standard. Il lavoro di tesi svolto, quindi, si è sviluppato a partire da questi casi di test già presenti, individuandone i problemi e creando nuovi strumenti di esecuzione e controllo delle soluzioni che non sono presenti all'interno della suite di test nella sua forma nativa.

Il tipo di testing preso in considerazione nel seguente elaborato di tesi è detto di tipo black-box, ovvero i casi di test sono scritti in base alle specifiche che sono fornite nel documento che definisce lo standard *JSR331*. In questo modo non è necessario conoscere nei minimi dettagli ciò che le diverse implementazioni intendono testare dal punto di vista del codice. I vantaggi principali di questa tipologia di testing rispetto ad una tipologia white-box che, invece, basa i test case sul codice e la struttura del programma da testare, sono, principalmente, i seguenti:

- Il testing non è influenzato dalla componente sotto test. Se l'autore del programma ha fatto un assunto implicito invalido che il programma

non sarebbe stato mai chiamato con una certa classe di input, può aver omesso di includere il codice per trattare tale input. Se il test fosse fatto esaminando il programma si potrebbero generare dati di test basati sull'assunto invalido.

- Robustezza rispetto a cambiamenti dell'implementazione. I dati per il black-box testing non devono essere cambiati anche se sono stati fatti grossi cambiamenti al programma sotto test.

- I risultati di un test possono essere interpretati da persone che non conoscono i programmi al loro interno.

Nello specifico la tipologia di testing adotta nello strumento TCK per JSR331 è compatibile con lo *Specification-based testing* utilizzato per il procedimento di testing d'accettazione.

Per questo tipo di validazione esistono tre principali scuole di pensiero che spiegano quali sono gli obiettivi e gli strumenti che tale tipologia utilizza. Essa, infatti, è concepita come:

- Uno stile di testing (collezione di strumenti di test e tecniche di validazione) specializzato nello scoprire quali argomenti e asserzioni sono stati messi nel documento di specifica e utilizzati per verificare la correttezza del prodotto in relazione a questi ultimi.

- Uno stile di testing focalizzato sulla prova che le specifiche contenute all'interno di un documento di specifica (e il codice che le implementano) sono logicamente corrette.

- Un insieme di tecniche di testing incentrate sulle verifica delle relazioni logiche tra variabili che sono documentate spesso all'interno della specifica.

Il seguente testo, viste le condizioni poste in precedenza, è strutturato nel seguente modo.

Il Capitolo 2 descrive che cos'è in generale un TCK (Technology Compatibility Kit), quali caratteristiche deve avere, come deve essere strutturato e quali sono i compiti che deve assolvere.

Il Capitolo 3 descrive brevemente che cos'è in generale un ambiente d'esecuzione, quali di questi tipi sono utilizzati attualmente per la gestione dei test e come questi si legano ad un TCK. In questo capitolo sarà fatta particolare attenzione all'ambiente *JTHarness* utilizzato per l'esecuzione dei test.

Il Capitolo 4 descrive più in dettaglio come viene utilizzato l'ambiente d'esecuzione *JTHarness* all'interno di un qualunque strumento TCK, identificando quali sono le principali modifiche da apportare ai test per poterli eseguire in questo ambiente.

Nel Capitolo 5 verranno analizzati i test in base alla complessità delle soluzioni che vengono fornite e saranno presentati gli strumenti che sono stati creati per permettere di controllare se una determinata soluzione è corretta oppure errata in base ai parametri stabiliti a priori.

Nel Capitolo 6 si analizza il caso concreto di applicazione delle strumento TCK allo standard JSR-331, con l'utilizzo di tutti gli strumenti costruiti nel lavoro di tesi e analizzati nel capitolo 5 per verificare se le implementazioni sono effettivamente corrette. Si vedrà in questo capitolo sia come viene strutturata la suite di test, sia come *JTHarness* riesce ad eseguirla.

Nel Capitolo 7 si vedranno, invece, i risultati che sono stati ottenuti eseguendo il solver JSetl[9][8], adattato allo standard JSR331, durante l'esecuzione dei test. Inoltre, i dati ottenuti saranno interpretati in termini di compatibilità con le specifiche in un'ottica di ottimizzazione delle risorse utilizzate e dei tempi d'esecuzione.

Capitolo 2

Lo strumento TCK: struttura e funzioni

2.1 Introduzione

Quando si parla di TCK[5][2](Technology Compatibility Kit) si deve per forza parlare di test di compatibilità. Infatti questo strumento mette a disposizione collezioni di programmi che come scopo hanno la dichiarazione di conformità di una specifica implementazione ad uno standard *Java*. Nel dettaglio si può definire cosa si intende per testing di compatibilità:

Definizione 2.1. Il testing di compatibilità consiste in un insieme di metodi usati per validare una specifica implementazione di uno standard *Java* (ad esempio JSR-331) allo scopo di garantire coerenza e un corretto funzionamento attraverso differenti piattaforme hardware, sistemi operativi e altre implementazioni della stessa specifica.

Dopo questo tipo di validazione, una determinata applicazione può essere considerata compatibile con lo standard che intende implementare e funzionante su qualsiasi tipo di piattaforma la utilizza. Questa metodologia di testing differisce dalla tradizionale validazione di un prodotto software sotto vari aspetti.

* Questa metodologia di testing non è creata con lo scopo principale di verificare robustezza, efficienza e facilità d'uso del software.

* Lo scopo principale è verificare che una data implementazione sia in linea e rispetti pienamente le specifiche dettate da uno standard tecnologico *Java*.

* Per garantire una forte compatibilità questi test pongono particolare attenzione sulle caratteristiche che differenziano le diverse implementazioni cercando di evidenziarne i problemi al fine di avere più uniformità possibile.

* Ogni test solitamente contiene una specifica asserzione che verifica la correttezza di una singola caratteristica dell'implementazione rispetto allo standard.

2.2 Caratteristiche principali

Per capire bene quindi il legame che c'è tra TCK e test di compatibilità possiamo dare la seguente definizione:

Definizione 2.2. Un TCK è una collezione di test di compatibilità, strumenti, e documentazione che permette ad uno sviluppatore *Java* di determinare se una implementazione è in linea con le direttive che vengono date in una precisa specifica.

Solitamente i test vengono scritti da coloro che prendono parte alla creazione e al mantenimento di un nuovo standard così da creare un TCK che sarà successivamente incorporato insieme alle specifiche da rispettare. Nasce spontaneo quindi chiedersi che cosa differisca un TCK creato come parte integrante di una specifica e l'insieme di test che uno sviluppatore software crea per verificare il corretto funzionamento di un'applicazione prima del suo rilascio. Le principali differenze riguardano aspetti diversi, tra i quali, quelli con maggiore rilevanza pratica sono i seguenti.

Prodotto: Il TCK è un prodotto che viene direttamente consegnato al cliente per testare la sua implementazione, mentre la suite di test creata per l'applicazione è uno strumento interno usato dallo sviluppatore e che di norma non viene rilasciato all'esterno. Questa differenza ha diverse implicazioni:

1. Il TCK deve essere scritto e progettato per un uso esterno all'ambiente di creazione e quindi avrà bisogno di una interfaccia semplice composta da messaggi facilmente comprensibili.
2. E' necessario avere un buon manuale per l'utente che ne specifichi l'utilizzo.
3. Dovrà prevedere una serie di servizi di supporto per l'utente.

Assenza di limiti: Il TCK deve essere progettato per funzionare su una qualunque implentazione che si riferisce alla specifica che si vuole testare, senza alcun limite sull'insieme di piattaforme hardware e software.

Principio della scatola nera: Chi progetta questa tipologia di strumenti non necessita di sapere come sia strutturata dettagliatamente l'implementazione che intende testare.

Configurabilità: Un TCK deve essere configurabile per tutti i possibili scenari di testing.

Imparzialità: Un TCK è concepito per garantire in modo equo e imparziale un ambiente entro il quale tutte le implementazioni possono essere validate senza favorirne qualcuna a discapito di un'altra.

Verificabilità: Un TCK deve poter fornire risultati verificabili e facilmente comprensibili dagli sviluppatori.

2.3 Componenti di un TCK

In questa sezione si analizzano i principali componenti che dovrebbero essere presenti all'interno di un TCK e che, eccezion fatta per il documento di pianificazione, sono inviati al cliente finale.

- Documento di pianificazione del TCK

- Norme di conformità

- Processo d'appello

- Exclude List

- Collezione di test (test suite)

- Strumenti per l'esecuzione e il mantenimento dei test

- Documentazione per l'utente

- Risultati dei test

2.3.1 Documento di pianificazione del TCK

Una volta stabilita definitivamente la specifica *Java* dall'organo preposto, con tutti i relativi dettagli e regole, il primo componente da sviluppare nel processo di creazione di un TCK è il documento di pianificazione. Questo è un documento interno, utile solo a coloro che devono progettare tale strumento e che non verrà poi consegnato al cliente finale. Tutto ciò che viene inserito nel piano sarà la base futura per tutti i componenti che dovranno essere creati successivamente.

Il documento è composto da più elementi tra i quali quelli di maggiore importanza sono:

- **Piano del progetto:** viene utilizzato per descrivere in modo più semplice possibile il TCK come un prodotto ovvero in termini di servizi che dovrà offrire.

- **Piano dei test:** descrive ciò che deve essere testato e come questo deve essere fatto per raggiungere lo scopo di dichiarare un'implementazione aderente allo standard in oggetto.

- **Piano di integrazione dei test:** descrive come i singoli test, a livello organizzativo, devono essere incorporati all'interno della collezione.

- **Specifiche dei test:** descrive minuziosamente l'obiettivo di ciascun test, come fa a raggiungerlo e i risultati che devono essere ottenuti.

- **Piano della documentazione:** chiarisce quale documentazione dovrà essere scritta per rendere il più facilmente comprensibile e utilizzabile l'intero strumento.

- **Piano di esecuzione dei test:** spiega come i test devono essere eseguiti, ovvero quali devono essere utilizzati dalle diverse implementazioni dello standard e con quali parametri d'esecuzione i singoli test devono essere impostati per ottenere i risultati desiderati.

2.3.2 Norme di conformità

Questa tipologia di norme definisce i criteri che una determinata implementazione *Java* deve rispettare per essere certificata come conforme alla specifica tecnologica. Detto molto più semplicemente queste regole definiscono ciò che deve essere validato correttamente dai test. Tali regole, oltre a definire il modo in cui dovrà operare il TCK, lo completano, in quanto definiscono ulteriori

criteri che magari sono impossibili da testare ma che comunque devono essere obbligatoriamente rispettati.

Le regole sono scritte rispettando un principio fondamentale, ovvero l'imparzialità. Infatti tali norme non devono mai essere a favore di un'implementazione piuttosto che un'altra e devono essere valide e praticabili su una qualunque piattaforma. Dopo la messa in uso del TCK i criteri possono sempre essere modificati e le eventuali modifiche poi ricadranno sui test. La pratica comune prevede di inserire sempre le norme di conformità all'interno del manuale di utilizzo, in quanto sono parte integrante del TCK e ne aiutano la comprensione e l'eventuale sviluppo di test da parte di terzi che intendono collaborare alla specifica.

2.3.3 Collezione di test (test suite)

Una test suite è una collezione di test di conformità inclusi nello strumento TCK progettati per verificare che una data implementazione è in linea con le specifiche alle quali intende adeguarsi. Ogni TCK dovrebbe avere più collezioni di test, ognuna delle quali è stata progettata per validare un aspetto importante dell'implementazione.

I test che si trovano all'interno di tale raccolta dovrebbero essere totalmente automatici riducendo al minimo le iterazioni con l'utente durante l'esecuzione e idealmente dovrebbero coprire tutte le asserzioni presenti all'interno del documento di specifica.

Il singolo test dovrebbe, inoltre, testare una sola asserzione e non gruppi di asserzioni ove è possibile. Questo comportamento evita che i possibili problemi all'interno di un test compromettano la validazione di gruppi di asserzioni totalmente diverse tra di loro.

Ogni test deve ritornare due sole tipologie di risultato: **Passed** o **Failed** e nel caso non sia possibile assegnare uno dei due stati è necessario documentare bene il tipo di eccezione sollevata.

Per sviluppare singoli casi di test per un TCK è utile tenere in considerazione alcune linee guida principali:

* I test devono essere progettati in modo tale che lo stato di uscita non sia mai ambiguo e quindi di uno solo dei due tipi visti in precedenza.
* Tutti i test dovrebbero implementare la medesima interfaccia.
* Ogni test dovrebbe ripristinare lo stato del sistema una volta che la sua esecuzione è stata terminata.
* Nessun risultato di qualunque test deve dipendere dall'esecuzione di uno precedente.

* Ogni test deve riportare in modo significativo e chiaro qualunque errore venga riportato durante la sua esecuzione senza assumere comportamenti imprevedibili.

* Ogni test dovrebbe validare al più una sola asserzione della specifica.

Guardando ancora più in generale, la suite di test presente all'interno del TCK devono essere garantiti i seguenti tre punti:

1. Totale indipendenza delle piattaforme sia hardware che software.

2. Indipendenza dalle specifiche implementazioni, ovvero non devono esistere test che funzionino solamente su di un certo tipo di applicazione.

3. Chiarezza del risultato così da determinare subito se il test è passato oppure ha riscontrato degli errori in esecuzione.

I test presenti a loro volta possono essere suddivisi in obbligatori e non obbligatori in base a ciò che vanno a validare. Solitamente i test obbligatori validano le asserzioni di maggiore importanza e sono quelli che dichiarano un'implementazione in linea con le direttive della specifica. Quelli non obbligatori, invece, hanno il compito di testare caratteristiche opzionali dell'implementazione che sono definite facoltative all'interno del documento di specifica. Il superamento di questi ultimi è da ritenere importante solo se sono passati correttamente i test obbligatori. Molto spesso in questa categoria entrano a far parte tutti quei test scritti da terzi che collaborano nella creazione della specifica e che hanno come scopo, oltre quello di dichiarare corretta l'implementazione, anche la valutazione dell'efficienza in termini di occupazione di memoria e tempi di calcolo.

2.3.4 Processo d'appello

Come si può immaginare, un TCK messo in uso per testare le diverse implementazione di una specifica necessita sempre di essere mantenuto e, all'occorrenza, ogni componente che ne fa parte può essere modificato, eliminato oppure gli potranno essere aggiunti dei nuovi componenti. Il procedimento che permette a coloro che utilizzano il TCK di apportarne modifiche è detto processo d'appello. Questo procedimento deve essere specificato in ogni dettaglio al momento della creazione del Technology Compatibility Kit in un documento che accompagna la vita di questo strumento.

Ogni modifica che si vuole apportare, che può riguardare qualunque componente all'interno del TCK, deve rispettare le fasi che sono incluse nel documento e alla fine se questa viene approvata allora diventa definitiva per tutti.

2.3.5 Exclude List

Spesso, specialmente nelle prime versioni del TCK, capita che alcuni test presentino dei problemi che vengono sollevati durante la loro esecuzione. Gli eventuali errori, che non dipendono dall'implementazione, ma unicamente dalla struttura del test, vengono portati nel processo d'appello, valutati dagli organi preposti al mantenimento del TCK ed eventualmente si decide di toglierli dalla suite di test.

Questo processo di rimozione è poco utilizzato in quanto richiederebbe ogni volta che c'è la modifica anche solo di un test di rilasciare una nuova versione dell'intero strumento, che per certe specifiche può assumere dimensioni davvero consistenti.

E' stato ideato, a tal proposito, il sistema della Exclude List, ovvero un metodo di gestione dei test che prevede di inserire, all'interno di questa lista, tutti quei test che durante un processo d'appello sono stati dichiarati inappropriati oppure errati. In questo modo si evita di avere troppe versioni del TCK e quindi un maggiore controllo su tutti i suoi elementi. Inoltre, i test, anche se errati, non vanno mai persi e sarà in futuro possibile apportare le modifiche necessarie affinché possano funzionare.

Ogni TCK deve sempre essere dotato di questa lista che ha bisogno di continui aggiornamenti e manutenzioni. I test che si trovano al suo interno non necessitano di essere eseguiti e tanto meno di essere superati con successo, quindi devono essere tralasciati quando si vuole validare la propria implementazione. Le ragioni che incidono maggiormente sulla scelta di fare entrare un test nella Exclude List di un TCK sono:

- La scoperta di un errore o di una ambiguità all'interno del documento di specifica che quindi rende invalido il test che si preponeva lo scopo di validare l'asserzione errata.

- Un banale errore di logica o di programmazione all'interno del test.

- Il test riporta errori dovuti a bug presenti negli strumenti di esecuzione automatica.

- C'è una dipendenza del test ad una specifica implementazione o ad una piattaforma hardware specifica.

2.3.6 Strumenti per l'esecuzione dei test (Test Framework)

All'interno del TCK devono sempre essere presenti tutti quegli strumenti che permettono di supportare la fase di esecuzione dei test. Questi devono essere

sempre personalizzabili e configurabili così da poter andare incontro alle esigenze dello sviluppatore di implementazioni. Gli strumenti che solitamente si trovano all'interno del TCK sono:

Test Harness: questo è l'ambiente di esecuzione all'interno del quale i singoli test vengono lanciati automaticamente al fine di minimizzare gli errori che possono verificarsi quando l'utente interviene manualmente nel flusso. Tale strumento garantisce: una configurabilità totale dell'ambiente in base alle esigenze, la possibilità di eseguire solo alcuni test invece di altri e la capacità di produrre report sullo stato dell'esecuzione in maniera automatica.

Librerie Esterne: insiemi di file e librerie che necessitano dell'ambiente di esecuzione per funzionare e per poter eseguire correttamente i test senza incorrere in errori di compilazione.

Agente di test: è la vera e propria applicazione *Java*, solitamente incorporata nell'ambiente di esecuzione, che si occupa di eseguire i test che gli vengono somministrati e di ritornarne i risultati così da essere interpretati in base alle configurazioni impostate.

Eventuali script per l'esecuzione di test in determinate modalità.

2.3.7 Documentazione per l'utente

La documentazione che viene rilasciata all'utente contiene principalmente i seguenti documenti:

Guida per l'utente: solitamente viene affiancata alla guida del Test Harness per l'esecuzione dei test ed è composta da:

- Panoramica sul TCK e sulla relativa specifica per la quale si vuole verificare la compatibilità
- Descrizione di come installare e configurare i test e l'ambiente d'esecuzione
- Descrizione di come avviare il TCK per verificare la configurazione.
- Descrizione di come avviare i test all'interno del TCK per testare l'implementazione
- Regole di conformità che devono essere rispettate dall'implementazione sotto validazione.

- Descrizione del processo d'appello e modalità per fare ricorso.

Documentazione dei test: ogni test deve essere opportunamente documentato tramite i seguenti elaborati:

- Codice sorgente
- Metadati interpretati dal Test Harness per localizzare e eseguire i test individuati.
- Specifica del test ovvero quello che il test deve fare per verificare un'asserzione dello standard e i risultati che si devono ottenere.

2.3.8 Risultati dei test

Come ultimo elemento ci sono tutti quegli strumenti che permettono di elaborare i risultati prodotti dai test e che determinano se un'implementazione è in linea con le specifiche imposte da un determinato standard.

La possibilità di ottenere risultati leggibili in base alle soluzioni ottenute dall'esecuzione dei test è possibile grazie allo specifico ambiente di elaborazione dei test (Test Harness). Solitamente la riproduzione dei risultati in report è fatta in modo automatico e si basa sul confronto tra soluzioni attese e soluzioni trovate dalla specifica implementazione.

Questi strumenti tuttavia non sono obbligatori e la loro presenza è dettata dal fatto che esistono come elementi di base all'interno dell'ambiente di esecuzione e che possono essere configurati dall'utente in base alla quantità di dati che si vuole generare.

Si vedrà nel capitolo successivo l'analisi e la spiegazione dettagliata di che cos'è un ambiente di testing messo a disposizione in un TCK, essendo questo l'elemento di maggior peso al suo interno e di importanza fondamentale. Nel dettaglio vedremo un ambiente di testing specifico che è *JavaTest Harness* e nei capitoli successivi vedremo come questo si lega al pacchetto di strumenti TCK che vengono messi a disposizione insieme ad una specifica *Java*.

Capitolo 3

Ambienti di esecuzione automatica per suite di test

3.1 Introduzione

Da tutto ciò che è stato detto nel capitolo 2 si capisce chiaramente che la parte più importante e di maggiore impatto all'interno di un TCK è la collezione di strumenti che permettono l'esecuzione dei test.

L'ambiente di esecuzione e gestione dei programmi di test è detto Test Harness e al suo fianco troviamo sempre l'agente ovvero l'applicazione *Java* che esegue effettivamente i test in una determinata configurazione. Dunque con il termine *agente d'esecuzione* si intende la *Java Virtual Machine* che, posta su di un dispositivo qualunque che la supporti, esegue i test e ritorna i risultati da inviare all' Harness.

Nella maggior parte dei casi la locazione dell'ambiente di esecuzione dei test e dell'agente coincidono ed è quindi lo stesso dispositivo che svolge due funzioni:

- Collezionare i test e configurare il Test Harness a seconda delle proprie caratteristiche prestazionali degli strumenti installati, indicando la locazione dell'agente di calcolo presente sulla macchina.

- Gestire la comunicazione tra agente e ambiente d'esecuzione interpretando nel modo corretto i risultati forniti per presentarli all'utente nella maniera più chiara e corretta possibile.

Può capitare, invece, che l'agente di calcolo sia su di un altro dispositivo e che quindi entrino in gioco tutti quei meccanismi di comunicazione attraverso la rete in moto tale da permettere, dall'ambiente d'esecuzione, di specificare quali test, presenti sulla macchina ospite, debbano essere eseguiti

e come i risultati debbano essere interpretati per presentarli all'utente che sta effettuando il testing. In questo caso è di fondamentale importanza che i test siano presenti sulla macchina che ospita la *Java Virtual Machine* in modo tale da rendere minimo il traffico che circola sulla rete per ottenere delle prestazioni ottimali.

Nel seguito di questo capitolo si mostrerà, nei due casi, come avviene la comunicazione tra Test Harness e *Java Virtual Machine*. Inoltre si presenterà l'ambiente più utilizzato nel mondo *Java* per l'esecuzione e la gestione dei test, ovvero *JavaTest Harness* utilizzato nel lavoro di tesi.

3.2 Caratteristiche di un Test Harness

Da quello che abbiamo visto fino ad ora possiamo dedurre che le principali componenti che accomunano tutti gli ambienti di esecuzione dei test sono due, ovvero:

- Agente (o motore) d'esecuzione dei test, identificato nella *Java Virtual Machine*, che compie tutte le operazioni per portare a termine il programma da eseguire.
- L'insieme di configurazioni e programmi che permettono di comunicare con l'agente di calcolo, portando all'utente finale i dati elaborati nel modo più chiaro e completo possibile.

I compiti principali che devono assere assolti da un Test Harness prevedono:

1. L'esecuzione automatica di tutti i test, riducendo al minimo l'intervento dell'utente per evitare errori di natura umana che si possono verificare durante l'esecuzione.
2. Eseguire correttamente le suite di test che identificano i vari test case.
3. Possibilità di generare nel modo più automatico possibile report che analizzano i risultati così da presentarli all'utente in modo chiaro e leggibile.

La presenza di automatismi di esecuzione e di comunicazione con agenti di calcolo presenti sulla rete, fa di un Test Harness lo strumento più utile per il testing di determinate implementazioni. Inoltre, la possibilità di configurare l'ambiente entro il quale eseguire i test rende possibile testare un'applicazione in diverse modalità senza per forza dover cambiare la macchina fisica sulla quale il testing è eseguito.

Per una specifica implementazione è possibile mandare in esecuzione più test in parallelo grazie alla possibilità di associare ad ogni test uno specifico thread. Tutte queste caratteristiche viste determinano alcuni vantaggi che l'utilizzo di un Test Harness può avere:

1. Aumenta la velocità d'esecuzione dei test in quanto l'iterazione con gli utenti è ridotta al minimo oppure è inesistente.

2. Aumenta la possibilità e la velocità di scoprire errori e veri e propri bugs presenti all'interno di un' applicazione.

3. Aumenta la qualità del software testato e delle sue componenti.

4. Permette il testing anche quando nessun utente è presente nella fase di esecuzione, riportando eventuali errori e eccezioni soltanto a procedura conclusa.

3.3 Comunicazione tra ambiente e agente d'esecuzione

In questa sezione si analizza brevemente come avviene lo scambio dei dati tra il motore d'esecuzione e l'ambiente quando si trovano sulla medesima macchina oppure quando non si trovano nella medesima locazione.

Analizziamo il discorso principalmente per quanto riguarda test automatici, intesi come programmi per i quali l'interazione con l'utente è inesistente.

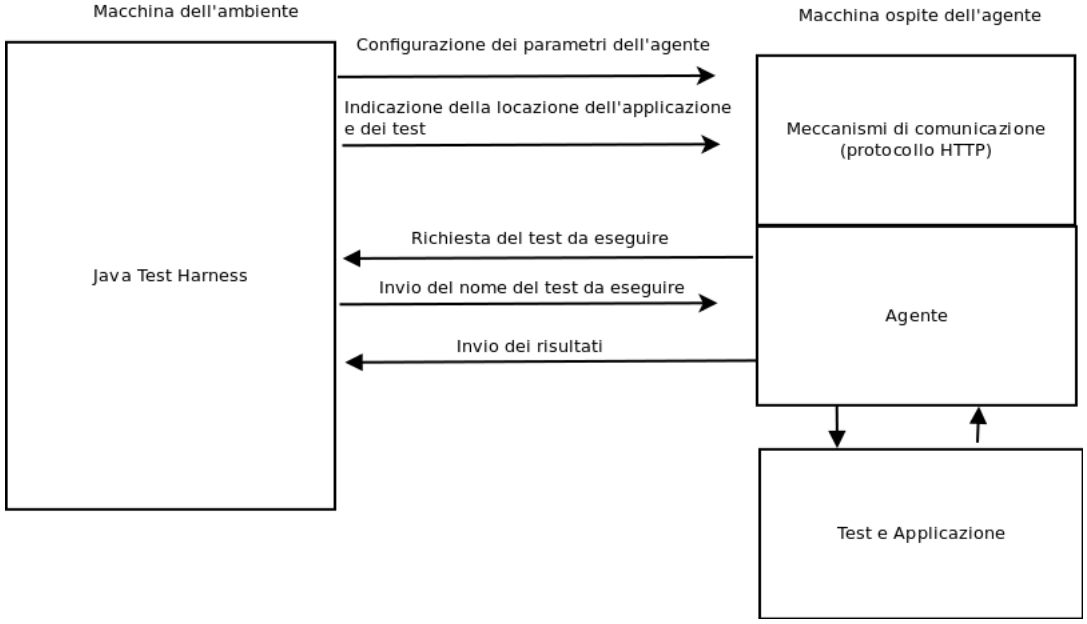
In linea di principio la comunicazione avviene seguendo i seguenti passi:

1. L'ambiente deve essere configurato, indicando dove si trovano le classi compilate dell'applicazione da testare e indicando la locazione dei test da utilizzare per validarla.

2. L'agente, per ogni test individuato, richiede che gli venga passato uno alla volta il codice sorgente compilato, seguendo l'ordine d'esecuzione predisposto, nel caso in cui le due componenti siano sulla medesima macchina. Nel caso in cui l'agente sia su un altro dispositivo, i test compilati sono già presenti e quindi deve solamente essere inviato al motore d'esecuzione il nome del test che si intende eseguire.

3. L'agente, dopo aver eseguito il test con tutti gli elementi a sua disposizione, ritorna all'ambiente il risultato e si predispone a riceverne uno nuovo.

Figura 3.1: Comunicazione tra ambiente e motore d'esecuzione



Qualora la macchina che deve eseguire il test si trovi in rete allora è necessario prevedere dei meccanismi di comunicazione. Inoltre sia i test che l'applicazione da testare devono trovarsi sul dispositivo dell'agente e limitare il compito dell'ambiente alla selezione dei test da eseguire e all'analisi dei risultati.

Quando c'è la presenza di un'iterazione con l'utente questa avviene sempre dalla parte dell'ambiente. Infatti l'eventuale inserimento di dati come input per il test avviene solo a fronte di richieste da parte dell'agente che li elabora e fornisce i risultati. A grandi linee possiamo vedere questo modello, sia in presenza d'utente che no, specialmente se avviene tramite la rete, come un fac-simile dello schema *client-server*; intendendo come lato cliente l'ambiente che comunica con il motore d'esecuzione che, invece, funziona da server.

Si possono riassumere le principali operazioni legate puramente all'esecuzione dei test da parte di un Test Harness nel seguente elenco:

- Operazioni di configurazione dell'ambiente di esecuzione con scelte quali la locazione dell'applicazione da testare e dei relativi test.
- Scelta dell'agente di calcolo tra quelli disponibili e relativa personalizzazione, selezionando ad esempio i parametri d'esecuzione come il numero di thread da associare alla suite di test per l'esecuzione simultanea di più test.

- Scelta della locazione di dove mettere i risultati per poterli elaborare e presentare all'utente.
- Personalizzazione delle configurazioni, scegliendo cosa e come configurare le varie parti dell'agente e dell'ambiente.
- Generazione automatica dei report dove vengono indicati i risultati ottenuti in base ai parametri preventivamente impostati.

A fianco di queste operazioni più pratiche si trovano anche tutte quelle che riguardano la gestione e la manutenzione delle suite di test.

- Possibilità di visualizzare i codici sorgenti dei test
- Possibilità di organizzare i test in directory per averne una collezione ben organizzata concettualmente e facilmente gestibile.
- Gestione delle Exclude List in base agli errori che sono presenti in determinati test tramite la creazione di queste liste e l'inserimento al loro interno di eventuali test errati.

Ovviamente i test che sono scritti per funzionare su di un Test Harness devono subire delle modifiche dal punto di vista del codice che saranno poi viste nel capitolo 4. Adesso ci si occuperà di vedere l'ambiente d'esecuzione più utilizzato delineandone brevemente le caratteristiche principali.

3.4 JavaTest Harness

JavaTest Harness (JT Harness)[3] è un software general-purpose, configurabile e flessibile creato per molti tipi di testing. Inizialmente è stato creato solo come strumento complementare ad un TCK per gestire ed effettuare il testing di compatibilità. Oggi è concepito come un software multipiattaforma in grado di gestire test suite di tutti i tipi. L'interfaccia grafica per JTHarness fornisce una comoda via per configurare ed eseguire semplici o complesse collezioni di test. Essa è continuamente migliorata per rendere la configurazione e l'esecuzione sempre più intuitiva e *user-friendly*. Questo strumento, tuttavia, non è da intendere come un banale tool per sviluppare e scrivere test suite in maniera elegante e semplificata, anche se il suo utilizzo mette a disposizione librerie che rendono questa attività più facile.

I suoi componenti principali sono:

JT harness UserInterface-engine: Il programma principali che si occupa dell'attività di scheduling dei processi, di reporting e di gestione dei risultati mettendoli a disposizione dell'utente.

Test Script: classe Java utilizzata da JT Harness per eseguire i singoli test.

Test Finder: classe Java che individua i singoli test in base alla descrizione fornita all'interno della suite.

Configuration Data: insieme di tutte le informazioni di configurazione dell'ambiente dentro la quale la suite di test viene eseguita

Observer APIs: componente che si occupa di riportare gli eventi, riguardanti i test, all'interfaccia grafica dell'utente.

Nei Configuration Data viene ovviamente specificato dove si trova l'agente d'esecuzione dei test che, come abbiamo visto, è l'elemento che sta sempre a fianco di un qualunque Test Harness.

I primi quattro componenti sono forniti direttamente da JavaTest Harness attraverso le opportune librerie messe a disposizione, mentre i test da eseguire e la configurazione sono dati da coloro che intendono eseguire le operazioni di testing per validare un'implementazione. La comunicazione tra la *Java Virtual Machine* e l'ambiente d'esecuzione avviene tramite la classe Test Script la quale fa eseguire i singoli test e ne ritorna i risultati. Per trovare i singoli test che gli vengono sottoposti la classe Test Script si appoggia sulla classe Test Finder che grazie alla locazione indicata dall'utente trova i test e li manda in esecuzione

Una parte molto importante di JavaTest Harness è anche la sua interfaccia grafica predisposta per l'utente. Questa permette di effettuare le seguenti operazioni:

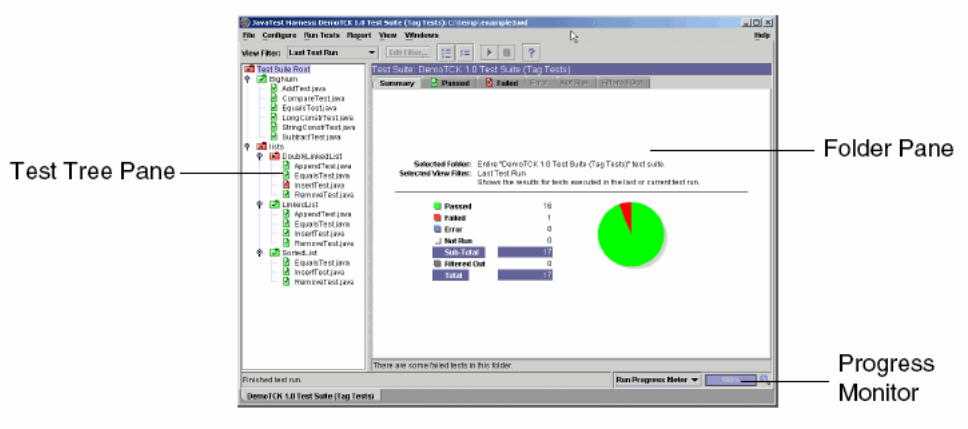
* Monitorare lo stato dei test.
* Valutare e analizzare i risultati dei test.
* Configurare facilmente l'ambiente di testing.
* Includere o escludere facilmente test dalla collezione.
* Generare report che riportano dettagliatamente le informazioni sui risultati ottenuti.

JavaTest Harness prevede per ogni test la possibilità di avere un risultato tra i seguenti valori:

Error: JT Harness non può eseguire il test in quanto ci sono errori dovuti a una scorretta configurazione dell'ambiente o dell'agente di calcolo.

Passed: Il test è stato completato correttamente senza sollevare eccezioni.

Figura 3.2: Esempio di interfaccia grafica JavaTest Harness



Failed: Il test è fallito.

Not Run: Il test non è stato eseguito perché ad esempio è inserito all'interno di una exclude list.

Filtered Out Il test non è stato scelto per essere eseguito, ma non fa parte di una exclude list.

L'utilizzo di questo strumento permette di catturare qualunque eccezione venga sollevata dall'implementazione quando viene posta sotto test e riportarla all'utente decidendo il risultato finale. Infatti, la vera forza di questo meccanismo è la possibilità, da parte di colui che scrive il test, di determinare con quali parametri quest'ultimo è da considerarsi superato con successo oppure fallito. Potrebbe capitare, infatti, di voler testare solamente il corretto funzionamento di alcuni metodi tralasciando la correttezza del risultato fornito. Inoltre è sempre possibile applicare a questo strumento librerie esterne che supportano la scrittura di test, quali *JUnit*, per la verifica della correttezza dei risultati ottenuti dall'implementazione.

Si vedrà ora nel capitolo successivo come si devono organizzare e scrivere i test presenti in un TCK affinché siano correttamente eseguiti. Mostreremo come JT Harness esegue i programmi e quale tipo di comunicazione c'è tra l'interfaccia grafica e il test a livello di codice *Java*.

Capitolo 4

Utilizzo di JTHarness all'interno di uno strumento TCK

Nel seguente capitolo è mostrato come un test, già scritto o comunque da scrivere, si deve adattare a partire dal codice sorgente per poter essere eseguito con JavaTest Harness. Verranno indicati quali porzioni di codice sorgente devono essere presenti obbligatoriamente e quali interfacce *Java* debbano essere implementate correttamente.

Successivamente sarà presentata la struttura di directory che deve essere rispettata affinché sia interpretabile dall'ambiente e direttamente presentabile all'utente nel Tree Panel.

Bisogna ricordare, tuttavia, che qualunque test, non solo quelli di compatibilità sono adattabili per funzionare su JavaTest Harness, l'importante è che rispettino le condizioni minimali che verranno presentati di seguito.

Oltre agli adattamenti che vengono visti nel seguente capitolo, si rende noto che esistono altri modi per rendere il test eseguibile su JT Harness, ad esempio implementando altre interfacce presenti nel package **javatest**. Qui saranno viste tutte le metodologie di adattamento sperimentate e utilizzate in questo lavoro di tesi.

4.1 Adattamento dei test a JavaTest Harness

Prima di addentrarci nei particolari bisogna premettere che è necessario disporre di due file jar essenziali per il funzionamento corretto dell'esecuzione dei test e della loro compilazione:

* **jh.jar:** che si occupa del sistema di help in linea presente nell'interfaccia grafica di JavaTest Harness

* **javatest.jar:** che è l'insieme di interfacce e classi che permettono ai test di comunicare con l'interfaccia grafica dell'utente e di essere eseguiti.

Vediamo ora nel dettaglio quali sono le parti di codice da inserire in un test affinché sia riconosciuto come eseguibile da JavaTest Harness.

4.1.1 Interfacce e classi necessarie

Per ogni test che si intende scrivere è necessario avere incluse la classe **Status** e l'interfaccia **Test**.

```
import com.sun.javatest.Test;
import com.sun.javatest.Status;
```

La classe Status è responsabile della comunicazione che avviene tra interfaccia grafica e risultato ottenuto dal test. Essa identifica il suo stato finale, che, come abbiamo visto, può assumere valori ben definiti, descritti nel capitolo 3 sezione 3.4. Il risultato d'uscita quindi è definibile dalla dichiarazione di una variabile di questo tipo.

```
static Status s;
```

Successivamente l'attribuzione di un risultato piuttosto che un altro viene decisa e applicata con i metodi presenti all'interno della classe. Quelli più utilizzati e visti nel lavoro di tesi sono:

*
```
static Status error(String reason);
```

  questo metodo permette di attribuire allo stato il valore `error` specificandone la ragione nella stringa `reason` che sarà poi utilizzata come descrizione della causa dell'errore.

*
```
static Status passed(String reason);
```

  il metodo attribuisce allo stato il valore `passed` con la motivazione specificata nel parametro `reason`.

*
```
static Status failed(String reason);
```

  la funzione assegna allo stato il valore `failed` con la ragione del fallimento racchiusa nella stringa `reason`.

L'assegnamento del valore desiderato ad uno stato *s* avviene quindi attraverso il seguente statement:

```
s = Status.passed("...");
```

E' buona norma che la decisione sul valore da assegnare allo stato sia presa da un metodo. Quest'ultimo, a discrezione di chi ha scritto il test, può mandare in esecuzione l'intero test oppure le varie parti in cui si suddivide determinando lo stato complessivo da ritornare all'ambiente d'esecuzione. Per questo motivo si richiede che la classe *Java* che contiene il test implementi l'interfaccia `Test`.

```
public class Nome_del_test implements Test
```

Infatti, l'effetto di questa scelta, obbliga a implementare correttamente il metodo `run` così definito:

```
public Status run(String[] args, PrintWriter out,
  PrintWriter err)
```

All'interno di questa funzione viene determinato lo stato d'uscita che sarà ritornato seguendo due modalità di decisione:

1. si sceglie di eseguire tutto il test all'interno della funzione `run` e lo stato viene determinato in base agli errori riscontrati nei diversi statement che si incontrano nell'esecuzione, all'interno del metodo.

```
public Status run(String[] args, PrintWriter out,
   PrintWriter err) {
  static Status s = Status.passed(...);
  try{
    .
    . // porzione di codice del test
    .
  }catch(Exception e){
    s = Status.failed(...);
  }
  .
  .
  .
  try{
    .
    . // porzione di codice del test.
    .
```

```
  }catch(Exception e){
    s = Status.failed(...);
  }
  return s;
}
```

2. Il test è suddiviso in funzioni ognuna delle quali esegue una porzione di codice e ritorna un risultato alla funzione `run` che lo elabora insieme agli altri per determinare lo stato finale.

```
public Status run(String[] args, PrintWriter out,
   PrintWriter err) {
  static Status s = Status.passed(...);
  if (!prima_parte_Test(...)){
    s = Status.failed(...);
  }
  .
  .
  .
  if (!n_esima_parte_Test(...)){
    s = Status.failed(...);
  }
  return s;
}
```

Lo stato `s` è ritornato solitamente nella funzione `main` attraverso l'istruzione:

```
s.exit();
```

4.1.2 Strutturazione delle informazioni per JavaTest Harness

L'ambiente d'esecuzione appoggia il proprio funzionamento su un insieme di directory e sottodirectory contenenti le diverse informazioni che servono al corretto svolgimento dei test. Si pone il caso di voler quindi testare una determinata applicazione e di non dover ricorrere alla rete per trovare i test e l'agente di calcolo.

La directory principale che contiene tutte le informazioni si chiamerà genericamente *jtNomeImplementazione_ Test*. Al suo interno troviamo quindi il seguente insieme di directory e file:

classes: è la directory che contiene tutti i file compilati necessari per eseguire i test. Al proprio interno troviamo le classi compilate dell'applicazione che vogliamo testare e tutte le altre che appartengono alle librerie esterne che vengono utilizzate come ad esempio la classe `Status` e l'interfaccia `Test`. Inoltre, nella cartella, devono essere presenti i file compilati dei sorgenti dei test. La locazione di questa directory deve essere specificata al momento della configurazione e sarà utilizzata dalla classe Test Finder per individuare i test.

data: è la cartella che contiene tutti i file di input per i test e le soluzioni che attese così che l'ambiente possa funzionare in maniera completamente automatica e autonoma.

lib: è la cartella che contiene i file jar necessari per avviare l'interfaccia grafica di JavaTest Harness.

tests: è la directory che contiene i sorgenti dei test, spesso suddivisi in sottodirectory in base alla loro tipologia.

conf.jti: è il file che contiene la configurazione impostata dall'utente in fase di preparazione dell'ambiente di esecuzione. Al proprio interno è specificata la locazione dei test compilati, dove si trova l'agente di calcolo, la presenza o meno di exclude list e altre specifiche di minore importanza.

NomeImplementazione.jtx: è la exclude list che contiene tutti quei test che vi rientrano.

testsuite.jtt: è il file da includere all'interno della directory principale per identificarla come contenente tutte le informazioni che permettono a JavaTest Harness di funzionare.

4.1.3 Script per il Test Finder

All'interno del test, è necessario incorporare uno script che permette alla classe `Test Finder` di individuare il singolo test all'interno della suite e mandarlo in esecuzione all'agente.

```
/**
* @test
* @sources Nome_test.java
* @executeClass Nome_test
*/
```

All'interno dello script `sources` identifica il nome del file del test, mentre `executeClass` si riferisce al file compilato del test. La classe `Test Finder`, elaborando queste informazioni, riesce ad trovare il test all'interno della locazione che gli è stata specificata in fase di configurazione e a mandarlo alla classe `Test Script` che si occuperà della sua esecuzione.

4.2 Esempio di test

Di seguito viene presentato un semplice test che si pone l'obiettivo di controllare la correttezza di due funzioni appartenenti ad una classe di nome `ArithmeticOperation` e che svolgono il compito di sommare due numeri interi in due modalità diverse. La correttezza delle soluzioni fornite dal test è controllata tramite le asserzioni di *JUnit*, che vedremo in dettaglio nel capitolo 5. Se tali soluzioni sono corrette significa che i metodi della classe sono implementati in modo esatto.

```
import junit.framework.TestCase;

public class Test extends TestCase{

  public static void main(String[] args) {
    /*
     * Test per la classe ArithmeticOperation effettuato
   tramite
     * lo strumento JUnit per verificare la correttezza dei
    risultati.
     * Tale classe effettua la somma tra due numeri interi.
     */
    ArithmeticOperation ao = new ArithmeticOperation(5,10);
    int x = 5;
    /*
     * Si fa il test della funzione operSum() che effettua
   la somma
     * tra gli operandi 5 e 10 verificando che ritorni il
   valore 15
     */
    assertEquals(ao.operSum(),15);
    /*
     * Si fa il test della funzione sum(int x) che effettua
    la somma
     * tra la variabile x e il valore 5 verificando che
   ritoni il valore 10
```

```
      */
     assertEquals(ao.sum(x), 10);
     System.out.println("Test OK");
   }
}
```

Come si può notare l'esempio sopra riportato non ha nessuna delle componenti che gli permettono di funzionare nell'ambiente JT Harness. Vediamo, quindi, come le modifiche vengono apportate al caso specifico riportando di seguito lo stesso esempio con le opportune variazioni di codice. Come prima cosa da fare si importano le classi necessarie per la comunicazione con l'ambiente d'esecuzione e si scrive lo script necessario al `Test Finder`.

```
import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.Test;
import junit.framework.*;
/**
* @test
* @sources TestModificato.java
* @executeClass TestModificato
*/
```

Successivamente il test deve ereditate dalla classe `Test Case` del package *junit.framework* e implementare l'interfaccia `Test`. Inoltre, vengono dichiarate all'interno della classe tutti gli oggetti che si intendono testare (`ArithmeticOperation ao`) insieme all'oggetto di tipo `Status` per comunicare all'ambiente d'esecuzione la buona riuscita o no del test.

```
public class TestModificato extends TestCase implements
   Test {

  ArithmeticOperation ao;
  static Status s = Status.passed("Test passed without
   exception");
```

Per questo tipo di test si è preferito suddividere l'esecuzione in due parti, una che testa la funzione `sum(int x)` e l'altra che testa al funzione `operSum()`. Il metodo `run`, quindi, richiama le due funzioni che testano i diversi metodi decidendo quale sia lo stato corretto da ritornare.

```
public Status run(String[] args, PrintWriter out,
   PrintWriter err) {
```

```
  String stringErr;
  stringErr = testSum();
  if (stringErr != "OK"){
    return s = Status.failed(stringErr);
  }
  stringErr = testOperSum();
  if (stringErr != "OK"){
    return s = Status.failed(stringErr);
  }
  return s;
}
public static void main(String[] args) {
  PrintWriter err = new PrintWriter(System.err,true);
  Test t = new TestModificato();
  s = t.run(args,null,err);
  s.exit();
}
```

Le funzioni che si occupano di testare i due metodi della classe `ArithmeticOperation` comunicano con la funzione `run` attraverso un opportuna stringa che indica se il risultato ottenuto è corretto oppure no.

```
  String testSum(){
    ao = new ArithmeticOperation(5, 10);
    int x = 5;
    try{
      assertEquals(ao.sum(x), 10);
    }catch(Throwable e){
      e.printStackTrace();
      return "Errore, soluzione non corretta";
    }
    return "OK";
  }
  String testOperSum(){
    ao = new ArithmeticOperation(5,10);
    try{
      assertEquals(ao.operSum(),15);
    }catch(Throwable e){
      e.printStackTrace();
      return "Errore, soluzione non corretta";
    }
  return "OK";
  }
```

```
} // FINE TEST
```

Il test scritto in questo modo è pronto per essere eseguito nell'ambiente JavaTest Harness in quanto è dotato di tutte quelle componenti che gli servono per poter comunicare con l'interfaccia grafica dell'ambiente d'esecuzione.

4.3 Risultati dell'esecuzione

Quando JavaTest Harness ha finito di eseguire tutta la suite di test, i risultati sono riportati in una directory preventivamente selezionata in fase di configurazione e che è buona norma chiamare *jt_NomeImplementazioneResult*. All'interno di questa cartella troviamo:

jtData: una directory che contiene tutti i file di log per generare i report e le informazioni relative ad ogni test come risultato ottenuto, ultima data d'esecuzione e configurazione dell'ambiente impostata.

Cartelle contenenti i test: in base a come sono organizzati i test, per ogni cartella sono contenuti i file con nome *NomeTest.jtr* che contengono le informazioni relative ai risultati ottenuti per ciascuno di essi. I dati di maggiore importanza che sono riportati all'interno di questi file per ogni test sono:

- stato del test;
- risultato ottenuto;
- eventuali messaggi di log presenti all'interno del test;
- locazione e nome del test;
- caratteristiche salienti dell'ambiente d'esecuzione quali sistema operativo e versione di *Java*.

Capitolo 5

Rappresentazione e controllo delle soluzioni

5.1 Introduzione

Nel seguente capitolo verrà affrontato un tema molto importante e delicato che riguarda la rappresentazione di una soluzione fornita da un test, di un qualunque programma, e il controllo di correttezza di quest'ultima. Molto spesso, infatti, l'esattezza della soluzione fornita da un test è sinonimo della sua corretta esecuzione e quindi del suo superamento. Dunque se si unisce al controllo delle eccezioni che possono essere sollevate, il controllo sulla correttezza dei risultati è possibile determinare lo stato del test da far pervenire all'ambiente d'esecuzione.

Per affrontare l'argomento è necessario formulare alcune ipotesi da tenere in considerazione:

1. Il problema da risolvere, contenuto nel test, è composto da una serie di entità dette variabili che possono assumere solo un valore per volta nei diversi istanti temporali d'esecuzione.

2. Ogni variabile del problema è identificata da un nome.

3. La soluzione al problema è concepita come un insieme di variabili e di valori che queste assumono al termine dell'esecuzione. Ovviamente le variabili sono le stesse presenti nel problema posto.

4. Il numero di soluzioni fornite può essere idealmente infinito.

5. L'ordine delle soluzioni e delle variabili all'interno di esse non deve influire sulla loro valutazione di correttezza.

Le soluzioni vengono solitamente fornite insieme ai test e quindi sono parte integrante dello strumento TCK. Queste sono da considerarsi sempre corrette e, qualora la soluzione data a fianco di un test venga considerata errata, ad esempio durante il ricorso ad un processo d'appello, tutto il test è da ritenersi errato e inseribile in un Exclude List.

Nel seguito del capitolo saranno affrontati tutti gli strumenti, che sono stati creati per questo lavoro di tesi e che controllano le soluzioni che possono essere fornite dall'esecuzione di diversi test. L'argomento sarà sviluppato in base alla complessità delle soluzioni. Si parte da quelle composte da un numero limitato di variabili passando poi a quelle con un numero consistente, superiore alla decina. Dopo si vedrà il trattamento di quei test che forniscono più soluzioni e gli strumenti che permettono di effettuare confronti tra quelle attese e quelle ottenute durante l'esecuzione, tralasciando il numero di variabili che le compongono.

5.2 Soluzioni con un numero ridotto di variabili

In questa tipologia di soluzioni sono presenti quelle formate da un numero di variabili che non supera la decina e per le quali il controllo avviene ricorrendo al modello della asserzioni, ben strutturato attraverso lo strumento *JUnit*.

Un'asserzione per *JUnit* è un metodo che verifica la veridicità o la falsità di un predicato. Qualora tale condizione non sia verificata viene sollevata un'eccezione che dovrà essere gestita per fornire un opportuno messaggio all'utente. I tipi di asserzione presenti possono essere di uno dei seguenti:

AssertEquals: verifica l'uguaglianza tra due parametri che gli vengono passati

AssertNotNull: verifica che il parametro dato in ingresso non sia nullo (valore speciale Null).

AssertNull: verifica che il parametro in ingresso sia nullo.

AssertTrue: verifica che la condizione passata come parametro sia vera.

AssertFalse: verifica che la condizione passata come parametro sia falsa.

Per poter utilizzare questi strumenti è necessario che la classe che contiene il test da svolgere erediti dalla classe `TestCase` contenuta nel package **junit.framework**. In questo modo le asserzioni sono viste come metodi del test e quindi possono essere richiamate in una qualunque porzione di codice senza dover dichiarare nessun oggetto specifico.

```
public class TestExpressions extends TestCase
```

L'assezione JUnit è strutturata nel seguente modo:

```
AssertEquals(int/String actual, int/String expected);
```

dove il parametro `actual` contiene il valore di un variabile della soluzione fornita dal test, mentre `expected` contiene il valore calcolato a priori per essa.

Vista la possibilità che l'asserzione fallisca e che quindi rilanci un'eccezione è sempre meglio inserirla all'interno di un blocco **try-catch** che mandi un opportuno messaggio d'errore.

```
try{
  assertEquals(actual, expected);
}catch(Throwable e){
  e.printStackTrace();
  return "Error Message";
}
```

Sono necessarie, quindi, tante asserzioni quante sono le variabili della soluzione che si vogliono verificare. L'utilizzo di questo strumento porta ad una più chiara leggibilità del codice e ad un controllo diretto delle variabili del problema in questione. Inoltre il meccanismo di JUnit, qualora non fosse presente JavaTest Harness per automatizzare i test, permette di creare una classe che manda in esecuzione tutti i test che ereditano da `Test Case` senza alcun intervento a run-time.

In presenza invece di JavaTest Harness le due componenti collaborano insieme e quando un' asserzione fallisce il test assumerà lo stato finale `failed`, mandando nell'interfaccia grafica dell'utente il messaggio di errore e il risultato del test.

Questa modalità di controllo delle soluzioni tramite delle asserzioni può essere sostituita con degli statement **if-then-else** nel seguente modo:

```
if (actual == expected)
  return "OK";
else
  return exception;
```

Tuttavia questa tipologia di statement risulta meno leggibile rispetto a quella vista in precedenza e per certi tipi di asserzioni non è applicabile; ad esempio quando si vuole verificare la veridicità di un predicato. Inoltre per

certi tipi di variabile non è sempre semplice capire quale operatore utilizzare per verificare la relazione d'uguaglianza. Mentre con le asserzioni il metodo che contiene il giusto operatore viene scelto in base al tipo dei due parametri passati.

Tutti questi controlli che si sono visti fino ad ora sono applicabili se il numero di variabili presenti nella soluzione non supera la decina, altrimenti il numero di asserzioni all'interno del codice renderebbe il test illeggibile.

A tal proposito nelle sezioni successive vengono presentati tutti gli strumenti creati nel lavoro di tesi che permettono di operare su soluzioni composte da un numero consistente di variabili oppure su insiemi di soluzioni a loro volta contenenti un numero considerevole di valori.

5.3 Rappresentazione di una soluzione

Le tipologie di soluzioni considerate sono suddivise in due categorie in base al valore delle variabili. Nel seguito si assumerà che tali valori all'interno di una soluzione possono appartenere all'insieme dei numeri interi oppure possono essere stringhe di caratteri. Si hanno infatti soluzioni di interi e soluzioni di stringhe rappresentate ciascuna da due classi ben distinte:

- IntegerSolution

- StringSolution

5.3.1 IntegerSolution Class

La soluzione è rappresentata attraverso l'utilizzo di tre attributi:

```
private int numberOfElements = 0;

private Vector<String> elementsName = new Vector<String>(
   numberOfElements);

private Vector<Integer> elements = new Vector<Integer>(
   numberOfElements);
```

L'attributo `numberOfElements` indica il numero di elementi presenti all'interno dei vettori e indirettamente il numero di variabili della soluzione. Mentre gli attributi `elementesName` e `elements` modellano il concetto della coppia <variabile, valore>. Infatti nel primo vettore sono contenuti i nomi delle variabili, mentre nell'altro sono presenti i valori che queste assumono

nella soluzione. Ovviamente c'è una corrispondenza biunivoca tra valori e nomi ovvero, l'i-esimo nome che identifica l'i-esima variabile avrà come valore corrispondente l'i-esimo intero contenuto nel vettore `elements` dei valori.

I metodi che permettono di gestire la classe sono i seguenti:

```
public int getNumberOfElements(){
  return numberOfElements;
}
```

funzione che ritorna il numero di elementi all'interno dei due vettori. Questo valore è interpretabile come il numero di variabili presenti all'interno della soluzione.

```
public Vector<Integer> getElements(){
  return elements;
}
```

ritorna i valori interi delle variabili contenuti nella soluzione

```
public Vector<String> getElementsName(){
  return elementsName;
}
```

ritorna i nomi delle variabili della soluzione.

```
public void increaseElement(){
  numberOfElements++;
}
```

incrementa di una unità il numero di elementi presenti nei vettori, ovvero il numero delle variabili.

```
public void setElementName(int index, String name){
  elementsName.add(index,name);
}
```

aggiunge il nome `name` di variabile al vettore che contiene tutti i nomi all'indice specificato in `index`.

```
public void setElement(int index, int element){
  elements.add(index,element);
}
```

aggiunge il valore `element` al vettore dei valori nella posizione `index`.

```
public void addSolution(String[] names, int[] elements)
   {
  for (int i = 0; i < names.length; i++){
    setElementName(i,names[i]);
    setElement(i,elements[i]);
    numberOfElements++;
  }
}
```

dato un array di nomi `names` e uno di valori `elements` messi in corrispondenza biunivoca si aggiungono alla soluzione creata tutti i nomi e i valori corrispondenti.

```
public int getElement(int i){
  return elements.get(i);
}
```

ritorna il valore della variabile alla posizione i-esima del vettore `elements`

```
public String getElementName(int i){
  return elementsName.get(i);
}
```

ritorna il nome della variabile alla posizione i-esima del vettore `elementsName`

```
public int getElementWithName(String name){
  int i = 0;
  for (i = 0; i < elementsName.size(); i++){
    if(elementsName.get(i).equals(name)){
      break;
    }
  }
  return elements.get(i);
}
```

ritorna il valore della variabile che corrisponde al nome del parametro `name`.

Questa classe attraverso i metodi appena visti modella esattamente un tipo di soluzione a valori interi. I principali vantaggi che questa implementazione

offre sono: la possibilità di rappresentare soluzioni con un numero potenzialmente illimitato di variabili e di non interessarsi all'ordine con cui queste variabili sono presenti all'interno della soluzione, infatti ciascuna di questa è identificata da un nome che la rende univoca nei confronti delle altre e non da una posizione specifica nel vettore.

5.3.2 StringSolution Class

La classe *StringSolution* modella, invece, il concetto di soluzione composta da stringhe. I tre attributi, come nel caso precedente, contengono le informazioni riguardo alle variabili e ai valori.

```
  private int numberOfElements = 0;

  private Vector<String> elementsName = new Vector<String>(
   numberOfElements);

  private Vector<Integer> elements = new Vector<Integer>(
   numberOfElements);
```

I metodi sono gli stessi presentati in precedenza e hanno le stesse funzionalità. Ovviamente sono stati cambiati in modo da poter operare su stringhe.

```
public int getNumberOfElements(){
  return numberOfElements;
}

public Vector<Integer> getElements(){
  return elements;
}

public Vector<String> getElementsName(){
  return elementsName;
}
public void increaseElement(){
  numberOfElements++;
}
public void setElementName(int index, String name){
  elementsName.add(index,name);
}

public void setElement(int index, int element){
  elements.add(index,element);
```

```
}

public void addSolution(String[] names, int[] elements){
  for (int i = 0; i < names.length; i++){
    setElementName(i,names[i]);
    setElement(i,elements[i]);
    numberOfElements++;
  }
}

public int getElement(int i){
  return elements.get(i);
}

public String getElementName(int i){
  return elementsName.get(i);
}

public int getElementWithName(String name){
  int i = 0;
  for (i = 0; i < elementsName.size(); i++){
    if(elementsName.get(i).equals(name)){
      break;
    }
  }
  return elements.get(i);
}
```

5.4 Rappresentazione di un insieme di soluzioni

Data la rappresentazione di una soluzione è possibile creare una classe che implementi, invece, il concetto di insieme di soluzioni. Una cosa molto importante da tenere in considerazione è che ogni elemento dell'insieme non deve rispettare un ordine al suo interno. Infatti un test può fornire le soluzioni diversamente da un'implementazione all'altra e non è detto che i due insiemi forniti coincidano per cardinalità.

Per implementare questo nuovo concetto si è ovviamente fatto uso delle due classi `IntegerSolution` e `StringSolution` viste in precedenza per rappresentare la singola soluzione. Per creare un insieme, invece, si è fatto uso dell'interfaccia parametrica `Set`, implementata dalla classe templatica

`HashSet`, messe a disposizione nel package **java.util**. Le due classi create con l'ausilio di questi strumenti sono:

- SetIntegerSolution

- SetStringSolution

5.4.1 SetIntegerSolution Class

Questa classe utilizza due attributi fondamentali per modellare il concetto di insieme di soluzioni.

```
private Set<IntegerSolution> solutionSet = new HashSet<
    IntegerSolution>();

private int numberOfSolution;
```

L'attributo `numberOfSolution` tiene il conto di quante soluzioni sono presenti all'interno dell'insieme, mentre `solutionSet` è il vero è proprio insieme di soluzioni. Come si può vedere esso è un istanza della classe `HashSet` parametrica rispetto al tipo `IntegerSolution`. L'utilizzo di questa particolare implementazione degli insiemi permette di non curarsi dell'ordine con la quale le soluzioni sono pervenute dal test. Nell'insieme, infatti, ciò che conta sono gli elementi all'interno e il loro numero.

I metodi che permettono di utilizzare questo modello sono i seguenti

-
```
public Set<IntegerSolution> getSolutionSet(){
  return solutionSet;
}
```

ritorna l'insieme delle soluzioni di interi.

-
```
public int getNumberOfSolution(){
  return numberOfSolution;
}
```

ritorna il numero di soluzioni che sono presenti all'interno dell'insieme.

-
```
public void addSolution(IntegerSolution solution){
  solutionSet.add(solution);
  numberOfSolution++;
}
```

questo metodo aggiunge una soluzione di tipo `IntegerSolution` all'interno dell'insieme.

Come si può notare non esistono metodi all'interno della classe che ritornino la singola soluzione specificandone un indice o comunque un identificatore. Questo è dato dal fatto che ogni elemento dell'insieme non viene inserito in un punto specifico del contenitore. Per accedervi è necessario usare un tipo di dato `Iterator` che passa in rassegna tutti gli elementi dell'insieme.

5.4.2 SetStringSolution Class

Analogamente alla classe vista in precedenza in questa sezione viene trattato l'insieme delle soluzioni che hanno come valori delle stringhe al posto di interi. Gli attributi sono gli stessi, ma la classe `HashSet` viene parametrizzata rispetto al tipo di dato `StringSolution`.

```
private Set<StringSolution> solutionSet = new HashSet<
   StringSolution>();

private int numberOfSolution;
```

I metodi che la compongono sono del tutto simili a quelli che sono presenti nella classe *SetIntegerSolution* modificati per poter operare su stringhe.

```
public Set<StringSolution> getSolutionSet(){
  return solutionSet;
}

public int getNumberOfSolution(){
  return numberOfSolution;
}

public void addSolution(StringSolution solution){
  solutionSet.add(solution);
  numberOfSolution++;
}
```

5.5 Strumenti per il controllo delle soluzioni

In questa sezione sono presentati tutti gli strumenti che permettono di verificare se una soluzione o le soluzioni ad un determinato test sono corrette rispetto ai risultati ottenuti. Per fare tutto questo ci si basa sulle soluzioni

che sono note a priori ritenendole esatte e la verifica avviene seguendo diverse modalità:

* la soluzione fornita dal test coincide perfettamente con quella calcolata a priori.
* si vuole controllare la correttezza solo di alcune variabili all'interno della soluzione
* si verifica se la soluzione trovata appartiene all'insieme delle soluzioni possibili esatte.
* si controlla che l'insieme di soluzioni trovate sia un sottoinsieme di quelle attese o viceversa.

Si partirà nell'esposizione di queste metodologie di controllo da quelle soluzioni che hanno un numero consistente di variabili, per aumentare di complessità fino a raggiungere insiemi di soluzioni con un numero di variabili considerevole.

5.6 Soluzioni con un numero consistente di variabili

Per ovviare al problema di trovarsi davanti ad una soluzione con un numero considerevole di variabili e dover scrivere per ciascuna di esse una specifica asserzione, sono stati creati, nel lavoro di tesi, degli strumenti più potenti per effettuare i dovuti controlli. L'idea che sta alla base è la possibilità di costruire soluzioni secondo il modello descritto dalle classi `IntegerSolution` e `StringSolution`.

In ogni test è quindi necessario costruire la soluzione calcolata in fase d'esecuzione e la soluzione attesa che può trovarsi tranquillamente su di un file esterno. Il controllo che viene fatto può essere di due tipi:

1. Bisogna controllare che la soluzione fornita da un'implementazione sia esattamente identica a quella determinata a priori.
2. Bisogna controllare che alcune variabili presenti nella soluzione calcolata abbiano lo stesso valore delle stesse variabili nella soluzione attesa.

Per poter sviluppare gli strumenti necessari si è preso spunto dal modo di operare di *JUnit*, o per meglio specificare, della classe `TestCase`. Si vuole,

infatti, che esistano due metodi, uno per le stringhe e l'altro per gli interi, che prese in ingresso due soluzioni del tipo `StringSolution` o `IntegerSolution` sollevino un' eccezione se esiste un valore di una variabile nella soluzione attesa che non è uguale al valore della stessa variabile nella soluzione calcolata dall'applicazione sotto test.

Per prima cosa è necessario definire una nuova eccezione che sarà poi sollevata dai diversi metodi che effettuano i controlli. La classe che si occupa di questa definizione è detta `SolutionException` ed è così definita:

```
public class SolutionException extends Exception {
  private static final long serialVersionUID = 1L;

  public SolutionException(String message){
    super(message);
  }
}
```

La classe che invece si occupa della gestione dei metodi per il controllo delle soluzioni si chiama `AssertionSolution` ed ora, nel dettaglio, vengono mostrati i metodi che operano su oggetti di tipo `IntegerSolution` e `StringSolution`.

-
```
private boolean AssertEqualStringSolutionOpz(
   StringSolution found, StringSolution expected ){
  boolean ok = true;
  for (int i = 0; i < expected.getNumberOfElements(); i
  ++){
    if (found.getElementsName().contains(expected.
  getElementName(i))){
      if (!expected.getElement(i).equals(found.
  getElementWithName(expected.getElementName(i)))){
        ok = false;
      }
    }
  }
  return ok;
}
```

il metodo privato, date due soluzioni composte da stringhe, si occupa di ritornare vero se per ogni valore contenuto nelle variabili della soluzione `found` c'è un valore uguale della stessa variabile che compone la soluzione `expected`. Ritorna falso altrimenti.

```
private boolean AssertEqualSolutionOpz(IntegerSolution
   found, IntegerSolution expected){
  boolean ok = true;
  for (int i = 0; i < expected.getNumberOfElements(); i
  ++){
    if (found.getElementsName().contains(expected.
  getElementName(i))){
      if (expected.getElement(i) != found.
  getElementWithName(expected.getElementName(i))){
        ok = false;
      }
    }
  }
  return ok;
}
```

questa funzione invece è stata creata per poter lavorare su numeri interi e svolge lo stesso compito che si è visto in precedenza.

```
public void AssertEqualSolution(IntegerSolution found,
   IntegerSolution expected) throws SolutionException{
  if(!AssertEqualSolutionOpz(found, expected)){
    throw new SolutionException("Assertion␣
  AssertEqualSolution␣failed");
  }
}
```

```
public void AssertEqualStringSolution(StringSolution
   found, StringSolution expected) throws
   SolutionException{
  if(!AssertEqualStringSolutionOpz(found, expected)){
    throw new SolutionException("Assertion␣
  AssertEqualStringSolution␣failed");
  }
}
```

questi due metodi invece utilizzano le funzione `AssertEqualSolutionOpz` per stringhe o interi allo scopo di rilanciare un'eccezione quando queste due ritornano il valore booleano `false`.

Con questi metodi è possibile applicare il confronto tra due soluzioni composte da un numero arbitrario di variabili qualora non si intendessero usare i metodi messi a disposizione da *JUnit*.

Questi strumenti permettono un confronto variabile per variabile del valore che contengono le soluzioni permettendo di controllarne solo alcune al posto di tutte. Quello che effettivamente fa la differenza è come viene costruita la soluzione `expected`. In quest'ultima infatti verranno inserite tutte le variabili che si vogliono controllare, nell'ordine che si preferisce e con i valori che effettivamente queste devono assumere. Qualora le variabili inserite nella soluzione attesa siano anche presenti nella soluzione calcolata, allora si procede alla loro verifica variabile per variabile. Questo modo di procedere garantisce un controllo personalizzato delle soluzioni e attraverso il modello delle asserzioni rende il codice molto più leggibile e intuitivo da scrivere. Un esempio di utilizzo di queste funzioni ricorda molto quello visto in precedenza per *JUnit*. L'asserzione viene racchiusa in un blocco **try-catch** e l'eventuale messaggio d'errore è ritornato alla funzione che si propone lo scopo di determinare lo stato del test.

```
try{
  AssertEqualSolution(foundSolution, expectedSolution);
}catch(Throwable e){
  e.printStackTrace();
  return "Error Message";
```

5.7 Più soluzioni con un numero arbitrario di variabili

In questa sezione sono analizzati gli strumenti che permettono di operare tra insiemi di soluzioni e tra una singola soluzione e tale insieme. La tipologia di controllo che si intende effettuare si può svolgere in due modalità:

1. si verifica che l'insieme delle soluzioni calcolate a priori coincida con l'insieme di quelle fornite dall'esecuzione del test, oppure che uno sia sottoinsieme dell'altro.

2. se il test fornisce una sola soluzione tra un insieme di possibili soluzioni, si deve verificare che questa gli appartenga effettivamente.

Si vedono ora i metodi che permettono prima di effettuare il primo controllo e successivamente l'altro; come sempre le funzioni che svolgono que-

sto compito sono suddivise in base alla tipologia dell'insieme delle soluzioni: `SetIntegerSolution` oppure `SetStringSolution`.

```
public void isASubSet(SetIntegerSolution set,
   SetIntegerSolution subSet) throws SolutionException{
  boolean ok = false;
  boolean control = true;
  Iterator<IntegerSolution> iter = subSet.
   getSolutionSet().iterator();
  IntegerSolution sol;
  while (iter.hasNext()){
    sol = iter.next();
    Iterator<IntegerSolution> iter2 = set.
   getSolutionSet().iterator();
    ok = false;
    while (iter2.hasNext()){
      if (AssertEqualSolutionOpz(sol, iter2.next())){
        ok = true;
      }
    }
    control = ok & control;
    if (control == false){
      throw new SolutionException("Assertion isASubSet
   failed");
    }
  }
}
```

```
public void isASubSetString(SetStringSolution set,
   SetStringSolution subSet) throws SolutionException{
  boolean ok = false;
  boolean control = true;
  Iterator<StringSolution> iter = subSet.getSolutionSet
   ().iterator();
  StringSolution sol;
  while (iter.hasNext()){
    sol = iter.next();
    Iterator<StringSolution> iter2 = set.getSolutionSet
   ().iterator();
    ok = false;
    while (iter2.hasNext()){
    if (AssertEqualStringSolutionOpz(sol, iter2.next())
   ){
```

```
        ok = true;
      }
    }
    control = ok & control;
    if (control == false){
      throw new SolutionException("Assertion␣
     isASubSetString␣failed");
    }
}
```

questi due metodi operano in modo totalmente identico con l'unica differenza sul tipo di dato che utilizzano. Il primo iteratore `iter` scandisce tutte le soluzioni presenti all'interno del parametro `subSet`, mentre il secondo `iter2` quelle che sono in `set`. A questo punto trattandosi di oggetti del tipo `IntegerSolution` o `StringSolution` è necessario controllare che nel primo iteratore e nel secondo i due oggetti rispettino la funzione `AssertEqualStringSolutionOpz` per il controllo delle variabili. Se per tutte le soluzioni contenute in `subSet` c'è una validazione corretta determinata da quelle in presenti in `set` allora il metodo non solleva nessuna eccezione, la rilancia altrimenti.

Nei casi in cui i due insiemi di soluzioni abbiano una cardinalità non molto elevata si tende a controllare che l'insieme delle soluzioni trovate sia uguale all'insieme di quelle determinate a priori. Quando invece il numero di elementi è grande l'operazione di controllo che si effettua è quella di verificare che nell'insieme di soluzioni fornite dal test, sia presente un sottoinsieme di soluzioni che sono quelle attese e determinate prima dell'esecuzione.

Un esempio di utilizzo dei due metodi molto simile a quello visto nella sezione 5.6 è riportato di seguito.

```
try{
  isASubSet(foundSolution, expectedSolution);
}catch(Throwable e){
  e.printStackTrace();
  return "Error␣Message";
}
```

Qualora invece l'implementazione che effettua il test fornisce una soluzione che non è l'unica al problema è necessario controllare che questa appartenga all'insieme delle possibili soluzioni determinate a priori. I due metodi che permettono di fare questa validazione sono i seguenti, ovviamente suddivisi in base al tipo di dato che trattano:

```
public void AssertIsSolution(IntegerSolution found,
  SetIntegerSolution expected) throws
  SolutionException{
 boolean ok = false;
 boolean control = true;
 Iterator<IntegerSolution> iter = expected.
  getSolutionSet().iterator();
 while(iter.hasNext()){
   IntegerSolution sol = iter.next();
   if(AssertEqualSolutionOpz(found, sol)){
     ok = true;
   }
 }
 control = ok & control;
 if (control == false){
   throw new SolutionException("Assertion␣
  AssertIsSolution␣failed");
 }
}
```

```
public void AssertIsSolution(StringSolution found,
  SetStringSolution expected) throws SolutionException
  {
 boolean ok = true;
 boolean control = true;
 Iterator<StringSolution> iter = expected.
  getSolutionSet().iterator();
 while(iter.hasNext()){
   StringSolution sol = iter.next();
   if(AssertEqualStringSolutionOpz(found, sol)){
     ok = true;
   }
 }
 control = ok & control;
 if (control == false){
   throw new SolutionException("Assertion␣
  AssertIsSolution␣failed");
 }
}
```

con i due metodi si controlla se una data soluzione `found` appartiene all'insieme delle soluzioni attese `expected`. Con un iteratore si passa in

rassegna tutto l'insieme e per ogni soluzione al suo interno si verifica se ogni variabile ha valore uguale a quelle della soluzione trovata con il metodo `AssertEqualStringSolutionOpz`. Se la soluzione appartiene all'insieme di quelle possibili il metodo non solleva eccezioni, altrimenti le rilancia.

Un esempio di utilizzo prevede l'inserimento della funzione in un blocco **try-catch**.

```
try{
  AssertIsSolution(foundSolution, expectedSolution);
}catch(Throwable e){
  e.printStackTrace();
  return "Error Message";
}
```

5.8 Considerazioni sull'utilizzo degli strumenti di controllo

Con tutti questi strumenti visti è possibile effettuare diversi controlli sulle soluzioni fornite da un test. Riassumendo è possibile così verificare le seguenti condizioni:

1. Verificare che una determinata soluzione trovata da un'applicazione, coincida con quella calcolata a priori su un numero arbitrario di variabili.

2. Verificare che una data soluzione appartenga all'insieme di quelle possibili

3. Verificare che l'insieme delle soluzioni trovate coincida o sia un sottoinsieme di quelle attese, e viceversa.

La costruzione delle soluzioni secondo i due modelli di classi previste permette allo sviluppatore di test di poter verificare solo quelle variabili che effettivamente interessano e nell'ordine che più si addice ai diversi casi. Qualora, invece, vi fossero diverse soluzioni il vantaggio principale che è presente in questi metodi, data la loro costruzione, è la possibilità di non doversi curare dell'ordine d'arrivo con cui le diverse soluzioni vengono fornite.

In uno strumento TCK è necessario fornire, oltre alle soluzioni dei diversi problemi, anche tutte queste classi che svolgono un ruolo molto importante. La validazione delle soluzioni, infatti, è una delle componenti principali

che incidono sulla decisione dello stato del test. Quando le soluzioni non sono corrette l'implementazione non può definirsi conforme allo standard che intende implementare, anche se, durante l'esecuzione, tutti i vincoli del problema sono impostati senza sollevare eccezioni. Tuttavia questo controllo non è sempre l'unico che viene effettuato, infatti è necessario sempre affiancarlo ad altri quali la verifica sulla corretta esecuzione di tutti i metodi presenti all'interno del test e la verifica della massima copertura di codice possibile dell'applicazione da testare.

Capitolo 6

Utilizzo dello strumento TCK per JSR-331

In questo capitolo sarà presentato lo strumento TCK per lo standard JSR331, standard che regolamenta la programmazione a vincoli. In un primo momento verrà analizzata la struttura e come, al proprio interno, sono presenti tutti gli strumenti, mettendone in mostra i difetti.

Nella seconda parte del capitolo invece vengono presentate le modifiche che devono essere apportate alla suite di test presente all'interno del TCK. Vedremo come quest'ultimo dovrà essere modificato per poter funzionare correttamente nell'ambiente d'esecuzione JavaTest Harness.

Come già citato nel capitolo 1, la suite di test è già presente all'interno dello strumento; il compito da svolgere è quello di modificarla al fine di renderla semplice da eseguire e totalmente automatica per limitare al minimo gli errori dovuti all'iterazione umana.

6.1 Struttura TCK

Vediamo in questa sezione come lo strumento TCK per JSR331 si presenta nella sua forma originale senza aver apportato alcuna modifica. Esso è contenuto nel pacchetto: *org.jcp.jsr331.tck* ed all'interno di essi sono contenuti i seguenti package con i relativi test.

cloud.balancing: contenente una serie di test ancora in via di sviluppo per implementazioni che prevedono anche l'utilizzo della rete.

org.jcp.jsr331.hakan: test di particolare complessità in cui la soluzione del problema implica la presenza di un numero consistente di variabili. Molto spesso in questi test la soluzione non è unica, ma sono un insieme

di soluzioni. Tuttavia non c'è alcun controllo sulla correttezza della soluzione finale.

org.jcp.jsr331.linear.samples: test sviluppati per testare i solver lineari quali *glpk, gurobi, ecc.* Di questi test non ce ne occupiamo, ma tutti i discorsi di struttura e verifica delle soluzioni fatte per gli altri sono valide ed adattabili anche per questa tipologia.

org.jcp.jsr331.samples: in questo package sono contenuti test di ogni genere sviluppati da tutte quelle persone che hanno contribuito allo standard JSR331. Sono quindi presenti test di divesa complessità e forma strutturale. Molto spesso sono privi di commenti e non vi è alcun controllo sulla soluzione finale per poter verificare se è corretta oppure no.

org.jcp.jsr331.tests: insieme di test obbligatori. Tali test presentano una complessità relativamente bassa in quanto ognuno di questi presenta una sola soluzione con un numero limitato di variabili. Per ognuno è presente un controllo della soluzione ottenuta e dei valori forniti tramite il metodo delle asserzioni di *JUnit* visto nel capitolo 5.

Come si può notare, quindi, c'è una suddivisione tra quelli che sono i test obbligatori e quelli non obbligatori. Il superamento dei primi consente di affermare che una determinata implementazione è aderente alle specifiche dello standard JSR331. Invece gli altri test, che non devono essere obbligatoriamente superati, hanno il principale scopo di misurare, in termini di efficienza, le diverse implementazioni che hanno preventivamente superati i test di sbarramento, controllando tuttavia attraverso le diverse funzioni richiamate che l'implementazione funzioni correttamente. Si può affermare che se non si superano tutti i test obbligatori non si possono sicuramente superare i test facoltativi, in quanto le funzioni richiamate sia negli uni che negli altri sono sempre le stesse. Inoltre sulla correttezza delle soluzioni gli unici test che le verificano sono quelli obbligatori che lo fanno attraverso le asserzioni del modulo aggiuntivo *JUnit*.

6.1.1 Test obbligatori TCK

I test che devono essere superati necessariamente sono i seguenti:

TestBins.java: in questo test si vuole rappresentare il problema dei contenitori. Quindi, dato un numero di contenitori e di sostanze, si pongono

dei limiti sulle capienze dei contenitori e sul tipo di sostanza che possono contenere. La soluzione finale dovrà mettere nel giusto modo le sostanze nei relativi contenitori rispettando i vincoli imposti.

TestCardinality.java: test che si compone di 3 parti ognuna delle quali vuole testare, con vincoli diversi, il metodo:

```
postCardinality(Var[] vars, int cardValue, String oper,
    Var var);
```

TestCompare.java: semplice test che si pone il problema di trovare una soluzione che rispetti dei vincoli posti su di una variabile x (x-1,...x-n).

TestGlobalCardinality : test per verificare il corretto funzionamento del metodo:

```
postCardinality(Var[] vars, int[] value, Var[]
   cardinalityVars);
```

TestGraphColoring.java: test che modella il problema di colorazione dei nodi di un grafo in modo che nodi adiacenti non abbiano gli stessi colori. In questo test vengono utilizzati i metodi che sono stati testati nei test precedenti piú il metodo:

```
postAllDifferent(Var[] vars)
```

TestIfThen.java: test per verificare il funzionamento del metodo:

```
implies(Constraint c)
```

TestMagicSequence.java: utilizzando i metodi testati precedentemente in questo test si compone un puzzle numerico così da ottenere una sequenza di numeri che rispetti determinati vincoli imposti.

TestMagicSquare.java: anche in questo test utilizzando i metodi testati in precedenza si costruisce un puzzle numerico dove la soluzione da trovare è una sequenza di numeri che rispetta determinati vincoli.

TestQueens.java: con questo test si verifica il numero di soluzioni trovate al problema delle 8 regine su di una scacchiera.

TestSolution.java: questo test é suddiviso in sotto-test ognuno dei quali ha il compito specifico di testare i metodi:

- ```
  findSolution()
  ```

  per trovare una soluzione al problema.

- ```
  findAllSolution()
  ```

  per trovare tutte le soluzioni possibili del problema.

- ```
  findSolutionIterator()
  ```

  per trovare tutte le soluzioni espresse sotto forma di iteratore.

- ```
  findOptimalSolution(Objective objective, Var
      objectiveVar)
  ```

  per trovare la soluzione ottimale al problema in base al parametro *objective* impostato.

TestSum.java: utilizzando i metodi testati in precedenza il test calcola somme tra variabili fornendo una soluzione ottimale al problema ed una soluzione qualunque nell'insieme delle possibili soluzioni che rispetti i vincoli impostati.

TestVarMatrix.java: piú che un vero test dove bisogna trovare la soluzione, in questo caso viene creata una matrice di variabili e viene poi stampata a video testando cosí il metodo:

```
variableMatrix(String name, int min, int max, int rows,
    int columns);
```

Con questi semplici test si vogliono quindi testare quei metodi di base che una qualsiasi implementazione dello standard JSR331 deve mettere a disposizione.

6.1.2 Test facoltativi TCK

I test facoltativi sono suddivisi in due sottocategorie:

- Test di Hakan: particolarmente utili per verificare al meglio le prestazioni dei solver, data la complessità dei test.

- Sample: test di complessità ridotta rispetto ai test di Hakan, ma che comunque aiutano a controllare il corretto funzionamento dei vari metodi e ricercano soluzioni composte da un numero di variabili che può essere consistente.

Per quanto riguarda questi test, visto il loro numero consistente, non si fa una lista come per il caso precedente, ma si evidenziano quali sono i problemi che li caratterizzano.

Il vero problema che risulta da queste due tipologie è che non vi è un controllo sulla correttezza delle soluzioni. Ovvero, non sono implementati metodi di controllo che prese le soluzioni ottenute controllano se queste appartengono all'insieme delle soluzioni possibili. Questo probabilmente è dovuto al fatto che gran parte delle soluzioni presentano un numero importante di variabili da considerare e molto spesso il numero di soluzioni è anche esso considerevole.

Questo è un punto cruciale che deve essere assolutamente controllato e che grazie agli strumenti visti nel capitolo 5, verrà risolto. Inoltre è da evidenziare il fatto che i test per essere eseguiti devono essere presi uno per uno manualmente ed eseguiti per una specifica implementazione. Quindi è necessario trovare uno strumento esterno che possa permettere ai vari test di essere lanciati a gruppi o singolarmente in modo automatico per una specifica implementazione; a tal proposito sarà utilizzato l'ambiente JavaTest Harness.

6.1.3 Considerazione sui test dello strumento TCK di JSR331

Tutti i test hanno una forma diversa e un fattore in comune, ovvero, prima di ricercare la soluzione con una tecnica precisa, definiscono e rappresentano il problema con i diversi vincoli. Si può pensare quindi di suddividere il singolo test in tre distinte funzioni:

* **testRepresentation(...)** che si occupa di controllare la fase di rappresentazione e definizione dei vincoli del problema.

* **testResolution(...)** che si occupa di testare i metodi di ricerca delle soluzioni e delle loro presentazione all'utente in maniera leggibile.

* **testSolution(...)** che preso come parametro la soluzione e l'array di soluzioni fornite dall'implementazione ne controlla la correttezza attraverso gli opportuni strumenti.

Nelle sezioni successive sarà mostrato come i test sono stati adattati al funzionamento nell'ambiente d'esecuzione JavaTest Harness. Inoltre si vedranno tutti quegli strumenti che permettono di costruire la soluzione trovata, la soluzione attesa e come queste due sono messe a confronto per determinare se il test ha avuto un risultato corretto.

6.2 Adattamento della suite per JavaTest Harness

Richiamando in parte gli argomenti del capitolo 4 vediamo in questo capitolo come i test già esistenti sono stati modificati per poter comunicare con l'ambiente d'esecuzione JavaTest Harness.

Si parte da un semplice esempio di test così come è presente all'interno del TCK nella sua forma nativa:

```
import javax.constraints.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class AllInterval {

  int n;
  Var[] x;
  Problem p = ProblemFactory.newProblem("All Interval");
    public static void main(String[] args) {
        int n_in = 10;

        if (args.length >= 1) {
            n_in = Integer.parseInt(args[0]);
        }
        System.out.println("\nn: " + n_in + "\n");
        AllInterval allInterval = new AllInterval();
        allInterval.define(n_in);
        allInterval.solve();
    }
    public void define(int n_in) {
        n = n_in;
        x = p.variableArray("x", 1, n, n);
        Var[] diffs = p.variableArray("diffs", 1, n-1, n-1)
  ;
        p.postAllDifferent(x);
```

```
        p.postAllDifferent(diffs);
        for(int k = 0; k < n-1; k++) {
            p.post(diffs[k],"=", x[k+1].minus(x[k]).abs());
        }
        p.post(x[0], "<", x[n-1]);
        p.post(diffs[0], "<", diffs[1]);
    }

    public void solve() {
        Solver solver = p.getSolver();
        SearchStrategy strategy = solver.getSearchStrategy
  ();
        strategy.setVars(x);
        strategy.setVarSelectorType(VarSelectorType.
  MIN_DOMAIN_OVER_WEIGHTED_DEGREE);
        int num_sols = 0;
        SolutionIterator iter = solver.solutionIterator();
        while (iter.hasNext()) {
            num_sols++;
            Solution s = iter.next();
            for(int i = 0; i < n; i++) {
                System.out.print(s.getValue("x-"+i) + " ");
            }
            System.out.println();
        }
        System.out.println("\nIt was " + num_sols + "
  solutions.\n");
        solver.logStats();
    }
}
```

Come si può notare dal codice presentato il problema è suddiviso in due parti, una che definisce il problema e l'altra che si occupa della sua risoluzione. Quello che bisogna assolutamente modificare, per permettere la comunicazione tra test e interfaccia grafica, è l'inserimento di uno stato che determina il risultato d'uscita del test e un ulteriore metodo che verifica la correttezza della o delle soluzioni.

In generale le modifiche che bisogna apportare ad un test, riguardano principalmente i seguenti punti:

1. Introduzione dello script per individuare il test all'interno della suite da parte della classe Test Finder

```
/**
```

```
* @test
* @sources AllIntervall.java
* @executeClass AllIntervall
*/
```

2. Importazione delle classi necessarie per comunicare con l'interfaccia grafica di JavaTest Harness e degli strumenti per la creazione della soluzione trovata, di quella attesa e del loro confronto.

```
import solutionTool.*;
import com.sun.javatest.Test;
import com.sun.javatest.Status;
```

3. La classe che implementa il test deve ereditare da `solutionTool` per poter disporre degli strumenti di confronto delle soluzioni come se fossero metodi propri. Inoltre deve implementare l'interfaccia `Test`.

```
public class NomeTest extends AssertionSolution
  implements Test
```

4. Implementazione obbligatoria del metodo `run` preso dall'interfaccia `Test` e che ha il compito di determinare quale sia lo stato del test al termine dell'esecuzione. Al suo interno vengono chiamate le tre funzioni: `testRepresentation(...)`, `testResolution(...)` e `testSolution(...)` e alla fine viene ritornato lo stato determinato dalla buona uscita di quest'ultime. Qualora una di queste sollevi un'eccezione il test è da ritenersi non superato.

```
public Status run(String[] args, PrintWriter out,
  PrintWriter err) {
    String stringErr;
    int n_in = 10;
    stringErr = testRepresentation(n_in);
    if (stringErr != "OK"){
      return s = Status.failed(stringErr);
    }
    stringErr = testResolution();
    if(stringErr != "OK"){
      return s = Status.failed(stringErr);
    }
    stringErr = testSolutions(sol);
    if(stringErr != "OK"){
```

```
        return s = Status.failed(stringErr);
    }
    return s;
}
```

5. Aggiungere necessariamente tra le variabili globali la variabile `s` che identifica lo stato, e il vettore di soluzioni che vengono fornite dall'esecuzione del test.

```
static Status s = Status.passed("Test AllIntervall
   passed without exception");
Solution[] sol; // Array contenente le soluzioni
   trovate.
```

6. Il metodo main che richiama l'esecuzione della funzione `run` ha il compito di creare il test e di ritornare all'interfaccia grafica lo stato che viene determinato.

```
public static void main(String[] args) {
  PrintWriter err = new PrintWriter(System.err,true);
  Test t = new AllInterval();
  s = t.run(args,null,err);
  s.exit();
}
```

7. Ogni metodo che può sollevare una qualunque eccezione all'interno delle tre funzioni che richiamano le varie parti del test deve essere racchiuso in un blocco **try-catch** e ritornare l'opportuno messaggio d'errore al metodo `run`. I metodi che devono essere obbligatoriamente racchiusi in questo tipo di blocchi sono:

   - Asserzioni per il controllo delle soluzioni

   ```
   try{
     isASubSet(foundSolution, expectedSolution);
   }catch(Throwable e){
     e.printStackTrace();
     return "Solutions found doesn't belong to the
      solution set";
   }
   ```

   - Metodi che impostano i vincoli del problema

```
try{
    p.post(x[0], "<", x[n-1]);
}catch(Throwable e){
    e.printStackTrace();
    return "Method post(x[0], '<', x[n-1]) failed"
   ;
}
```

- Metodi che configurano i parametri di ricerca della soluzioni

```
try{
    sol = solver.findAllSolutions();
}catch(Throwable e){
    e.printStackTrace();
    return "Method solutionIterator() failed";
}
```

Il metodo che si occupa di controllare la correttezza delle soluzioni è denominato `testSolution(...)` e richiama i metodi che costruiscono le soluzioni e i metodi di confronto a seconda della complessità della soluzione vista nel capitolo 5.

Per quanto riguarda le soluzioni composte da poche variabili il metodo `testSolutions(...)` prende come parametro un oggetto di tipo `Solution` e ogni asserzione che controlla i valori delle diverse variabili all'interno della soluzione è racchiusa in un blocco **try-catch**. Qualora l'asserzione fallisce il messaggio d'errore viene restituito al metodo `run` e da questo all'interfaccia grafica che si occupa di renderlo visibile all'utente finale.

```
String testSolutions(Solution solution) {
  try{
    assertNotNull(solution);
  }catch(Throwable e){
    e.printStackTrace();
    return "No solution found, but there must be a solution
   ";
  }
  try{
    assertEquals(solution.getValue("a"),3);
  }catch(Throwable e){
    e.printStackTrace();
    return "For a the expected result is 3, but the found
   result is " + solution.getValue("a");
```

```
  }
  try{
    assertEquals(solution.getValue("b"),7);
  }catch(Throwable e){
    e.printStackTrace();
    return "For b the expected result is 7, but the found
   result is " + solution.getValue("b");
  }
  solution.log();
  return "OK";
}
```

Con questi semplici accorgimenti è possibile far funzionare qualunque test nell'ambiente d'esecuzione JavaTest Harness. Vediamo nella sezione successiva come invece viene effettuato il controllo delle soluzioni, nel caso di soluzioni complesse, utilizzando gli strumenti visti nel capitolo 5 e come si devono costruire le soluzioni o gli insiemi di soluzioni.

6.3 Controllo di una singola soluzione complessa

Il modo di procedere cambia completamente con quei test che invece forniscono soluzioni complesse composte da un numero consistente di variabili. Gli strumenti utilizzati per il controllo di correttezza sono quelli visti nel capitolo 5. Bisogna però, invece specificare come le soluzioni vengono costruite. Lo standard JSR331 modella il concetto di variabile e soluzione secondo quanto implementato nelle classi `Var` e `Solution` del package **javax.constraints**.

Tuttavia quello che importa è che ogni soluzione è concepita come un insieme di variabili distinte e che ognuna di queste è identificata da un nome univoco. Inoltre i valori delle variabili, per quanto riguarda i test, appartengono all'insieme degli interi, per la maggior parte dei casi, oppure all'insieme delle stringhe, per un numero di casi più limitato. I risultati per questo tipo di soluzioni sono contenuti in un apposito file chiamato *expectedSimpleSolution.txt*. Al suo interno la diverse soluzioni ai problemi sono scritte rispettando un formato ben delineato secondo lo schema riportato di seguito.

```
NomeTest
var-1 valore-1 var-2 valore-2 ... var-n valore-n
!!!
```

Al posto dello spazio che separa la coppia <var-i, valore-i> possono essere presenti altri tipi di delimitatore, così da ammettere nel file anche le soluzioni

che prevedono stringhe contenenti degli spazi. Ovviamente l'ordine con cui le variabili sono inserite nel file può essere diverso rispetto all'ordine di dichiarazione delle variabili nel problema, l'importante è che ad ognuna corrisponda il suo valore specifico.

Per la creazione della soluzione attesa bisogna, quindi, leggere i valori che sono contenuti nel file. A questo scopo è stata creata la classe `ReadFromFile`. Per la soluzione fornita al problema bisogna prendere tutte le variabili e i valori che sono stati trovati durante l'esecuzione del test; a tal proposito è stata creata la classe `PrepareTest` dalla quale, per motivi pratici, eredita la classe `AssertionSolution`. In questo modo infatti è possibile accedere ai metodi delle due classi senza necessità di dover creare degli oggetti esterni rendendo il codice molto più leggibile.

Per quanto riguarda la classe `ReadFromFile`, essa è composta da due attributi che sono `pathName` per individuare il percorso del file system all'interno del quale si trova il file e `problemName` che individua la soluzione da leggere all'interno del file. Ogni metodo all'interno della classe ha lo scopo di leggere da file i valori e le variabili necessarie per costruire l'oggetto `IntegerSolution` o `StringSolution` a seconda dei casi. Di seguito sono riportati brevemente i metodi presenti all'interno della classe e che riguardano la lettura di questa tipologia di soluzioni:

-
```
public IntegerSolution readSolutionWithDelimiter(String
    delimiter)
```

```
public StringSolution readStringSolutionWithDelimiter(
   String delimiter)
```

i metodi leggono da file le coppie <var-i, valore-i> separate da un delimitatore specificato nel parametro `delimiter` e costruiscono gli oggetti `IntegerSolution` o `StringSolution` contenenti la soluzione attesa del problema.

-
```
public IntegerSolution readSolution(){

  File file = new File(pathName);
  IntegerSolution solution = new IntegerSolution();
  int i = 0;
  try{
    BufferedReader buffer = new BufferedReader(new
   FileReader(file));
    String line;
    while ((line = buffer.readLine()) != null){
```

```
        if (line.equals(problemName)){
          String line2 = "";
          while(!line2.contains("!!!")){
            line2 = buffer.readLine();
            if (line2.contains("!!!")){
              break;
            }
            StringTokenizer str = new StringTokenizer(
    line2,"␣");
            int nToken = str.countTokens();
            int j = 0;
            while (j < nToken){
              solution.setElementName(i, str.nextToken())
    ;
              solution.setElement(i,Integer.parseInt(str.
    nextToken()));
              j = j + 2;
              i++;
              solution.increaseElement();
            }
          }
        }
      }
      buffer.close();
    }catch(Throwable e){
      e.printStackTrace();
    }
    return solution;
  }
```

```
public StringSolution readStringSolution()
```

i metodi leggono da file le coppie <var-i, valore-i> tenendo come separatore di default il carattere spazio al fine di costruire l'oggetto `IntegerSolution` o `StringSolution`.

La classe `PrepareTest` è stata creata con lo scopo di costruire gli oggetti `IntegerSolution` o `StringSolution` contenenti, invece, la soluzione determinata dall'applicazione sotto test. I suoi metodi prevedono di ricevere come parametri di ingresso sia l'oggetto di classe `Solution` contenente la soluzione sia le variabili che compongono il problema. Queste possono essere passate alla funzioni in due modi differenti: sotto forma di array di oggetti di classe

`Var` oppure possono essere estratte dall'oggetto di classe `Problem` attraverso gli opportuni metodi.

I metodi che permettono di costruire la soluzione trovata sono i seguenti:

```
public IntegerSolution prepareSimpleIntegerTest(Problem
    p, Solution sol){

  Var[] variables = p.getVars();
  IntegerSolution solution = new IntegerSolution();
  String names[] = new String[variables.length];
  int elements[] = new int[variables.length];
  for(int i = 0; i < variables.length; i++){
    names[i] = variables[i].getName();
    elements[i] = sol.getValue(names[i]);
  }
  solution.addSolution(names, elements);
  return solution;
}
```

```
public StringSolution prepareSimpleStringTest(String[]
   stringSol, Problem p, Solution sol)
```

i due metodi, preso in ingresso il problema `p` che contiene le variabili e la soluzione `sol` che contiene i valori, per ognuna di esse costruiscono la soluzione trovata.

```
public IntegerSolution prepareSimpleIntegerTest(Var[]
   variables, Solution solution)

  IntegerSolution sol = new IntegerSolution();
  String names[] = new String[variables.length];
  int[] elements = new int[variables.length];
  for (int i = 0; i < variables.length; i++){
    names[i] = variables[i].getName();
    elements[i] = solution.getValue(names[i]);
  }
  sol.addSolution(names, elements);
  return sol;

}
```

```
public StringSolution prepareSimpleStringTest(String[]
   stringSol, Var[] variables, Solution solution)
```

con questi due metodi invece dato un array di variabili `variables` che si intendono controllare e la soluzione che ne determina i valori si costruisce l'oggetto `IntegerSolution` o `StringSolution` che contiene la soluzione trovata dall'implementazione.

Quello che quindi bisogna fare per poter applicare il confronto tra le soluzioni ottenute e quelle attese è costruire la soluzione attesa leggendo da file, specificando prima il suo percorso e il nome del test che si intende eseguire. Una volta letta la soluzione si procede a creare quella che l'implementazione sotto test ha trovato. A tal proposito si passano all'opportuno metodo della classe `PrepareTest` le variabili del problema o l'oggetto di classe `Problem` che le contiene e l'oggetto di classe `Solution` che contiene tutti i valori della soluzione. Dopo questa fase di preparazione si procede al confronto tramite l'opportuna asserzione e qualora questa sollevi un eccezione si rilancia un messaggio d'errore.

Un esempio chiarificatore della procedura descritta è riportato di seguito:

```
IntegerSolution foundSolution = new IntegerSolution();
IntegerSolution expectedSolution = new IntegerSolution();
ReadFromFile readFromFile = new ReadFromFile();

String testSolutions(Solution solution){
  foundSolution = prepareSimpleIntegerTest(p,solution);
  readFromFile.setPathName("data/expectedSimpleSolution.txt
  ");
  readFromFile.setProblemName("CoinsGrid");
  expectedSolution = readFromFile.readSolution();
  try{
    AssertEqualSolution(foundSolution, expectedSolution);
  }catch(Throwable e){
    e.printStackTrace();
    return "The solution found is not correct, it's not
   equal to the expected solution";
  }
  return "OK";
}
```

Con questi semplici strumenti è possibile controllare la correttezza della soluzione per quei casi di test che ne forniscono una composta da un numero consistente di variabili. I passi necessari per il controllo prevedono quindi:

1. Inserimento nel file della soluzione corretta che viene determinata a priori o secondo altri risultati di altre applicazioni.

2. Creazione della soluzione attesa letta da file tramite la classe `ReadFromFile`.

3. Creazione della soluzione trovata dall'applicazione che esegue il test tramite la classe `PrepareTest`.

4. Confronto delle soluzioni secondo l'asserzione `assertEqualSolution(...)` che si vuole utilizzare con conseguente determinazione dello stato del test.

6.4 Controllo di un insieme di soluzioni

Il discorso visto fino ad ora è valido anche per tutti quei test che anziché fornire un'unica soluzione, ne forniscono un insieme anche di notevole complessità.

Come visto in precedenza la procedura che permette il confronto tra soluzioni ottenute e soluzioni determinate prevede l'utilizzo delle classi `ReadFromFile` e `PrepareTest`. Le soluzioni attese dai singoli test sono sempre presenti in un file denominato *expectedComplexSolution.txt*. Al suo interno le soluzioni sono scritte secondo uno schema preciso che viene riportato di seguito:

```
NomeTest
var-1 valore-1 var-2 valore-2 ... var-n valore-n
var-1 valore-1 var-2 valore-2 ... var-n valore-n
!!!
```

dove per ogni riga del file è presente una soluzione all'interno del quale le coppie <var-i, valore-i> possono esser messe in un ordine diverso da quello di dichiarazione nel problema.

I metodi della classe `ReadFromFile` che permettono di leggere insiemi di soluzioni sono i seguenti:

- ```
  public SetIntegerSolution readMoreSolution(){

    SetIntegerSolution solSet = new SetIntegerSolution();
    File file = new File(pathName);
    int i = 0;
    try{
      BufferedReader buffer = new BufferedReader(new
     FileReader(file));
      String line;
      while ((line = buffer.readLine()) != null){
        if (line.equals(problemName)){
  ```
  ```

```
        String line2 = "";
        while(!line2.contains("!!!")){
          line2 = buffer.readLine();
          if (line2.contains("!!!")){
            break;
          }
          StringTokenizer str = new StringTokenizer(
  line2,"␣");
          int nToken = str.countTokens();
          i = 0;
          IntegerSolution solution = new
  IntegerSolution();
          for (int j = 0; j < nToken; j = j+2){
            solution.setElementName(i,str.nextToken());
            solution.setElement(i, Integer.parseInt(str
  .nextToken()));
            i++;
            solution.increaseElement();
          }
          solSet.addSolution(solution);
        }
      }
    }
    buffer.close();
  }catch(Throwable e){
    e.printStackTrace();
  }
  return solSet;
}
```

```
public SetStringSolution readMoreStringSolution()
```

i due metodi leggono da file riga per riga le diverse soluzioni del problema e ne costruiscono l'insieme tenendo come delimitatore di default tra le diverse coppie <var-i, valore-i> delle soluzioni, il carattere spazio.

- ```
  public SetIntegerSolution readMoreSolutionWithDelimiter
    (String delimiter)
  ```

```
public SetStringSolution
  readMoreStringSolutionWithDelimiter(String delimiter
  )
```
```

le due funzioni leggono le soluzioni riga per riga tenendo in considerazione che il carattere separatore tra un coppia <var-i, valore-i> e l'altra all'interno della stessa soluzione è specificato nel parametro `delimiter`.

Per quanto riguarda la classe `PrepareTest` i metodi utilizzati per costruire la soluzione trovata invece sono i seguenti:

- ```
  public SetIntegerSolution prepareComplexTest(Problem p,
      Solution[] solution){

    SetIntegerSolution sol = new SetIntegerSolution();
    IntegerSolution auxSolution = new IntegerSolution();
    for (int j = 0; j < solution.length; j++){
      if (solution[j] == null) break;
      auxSolution = prepareSimpleIntegerTest(p, solution[
     j]);
      sol.addSolution(auxSolution);
    }
    return sol;
  }
  ```

  ```
  public SetStringSolution prepareComplexTest(String[]
    stringSol, Problem p, Solution[] solution)
  ```

  i due , preso in ingresso il problema `p` che fornisce le variabili e le soluzioni `solution` che invece ne stabiliscono i valori esatti, per ognuna di esse creano l'insieme delle soluzioni trovate al termine dell'esecuzione del test.

- ```
  public SetStringSolution prepareComplexTest(String[]
    stringSol, Var[] variables, Solution[] solution)
  ```

  ```
  public SetIntegerSolution prepareComplexTest(Var[]
    variables, Solution[] solution){

    SetIntegerSolution sol = new SetIntegerSolution();
    IntegerSolution auxSolution = new IntegerSolution();
    for (int j = 0; j < solution.length; j++){
      if (solution[j] == null) break;
      auxSolution = prepareSimpleIntegerTest(variables,
     solution[j]);
      sol.addSolution(auxSolution);
    }
  ```

```
  return sol;

}
```

dato un vettore di variabili `variables` che si intendono controllare per ogni soluzione e l'array contenente tutte le soluzioni `solution` del problema viene creato l'insieme che le contiene.

La procedura che si deve seguire per il controllo di correttezza delle soluzioni è molto simile a quella vista per il caso precedente di un'unica soluzione complessa. Infatti, per prima cosa, bisogna determinare dove si trovi il file contenente le soluzioni esatte e quali di queste bisogna leggere specificando opportunamente gli attributi `pathName` e `problemName` della classe `ReadFromFile`. A questo punto bisogna leggere l'insieme delle soluzioni e assegnarlo all'opportuno oggetto di classe `SetIntegerSolution` o `SetStringSolution`. Dopo di che si crea l'insieme delle soluzioni trovate dall'implementazione attraverso gli opportuni metodi della classe `PrepareTest`. Una volta ottenuti i due insieme si procede al confronto con l'opportuna asserzione `IsASubSet(...)` che verifica l'identità tra i due insiemi o che uno sia sottoinsieme dell'altro.

Un esempio chiarificatore della procedura è riportato di seguito:

```
SetIntegerSolution foundSolution = new SetIntegerSolution()
   ;
SetIntegerSolution expectedSolution = new
   SetIntegerSolution();
ReadFromFile readFromFile = new ReadFromFile();

String testSolutions(Solution[] solutions){
    foundSolution = prepareComplexTest(A, solutions);
  readFromFile.setPathName("data/expectedComplexSolution.
   txt");
  readFromFile.setProblemName("CalculsDEnfer");;
  expectedSolution = readFromFile.readMoreSolution();
  try{
    isASubSet(foundSolution, expectedSolution);
  }catch(Throwable e){
    e.printStackTrace();
    return "Solutions␣found␣doesn't␣belong␣to␣the␣solution␣
   set␣or␣not␣found␣in␣first␣100␣solutions";
  }
    return "OK";
}
```

6.5 Controllo di appartenenza all'insieme delle soluzioni

Il caso che raggruppa e utilizza tutti gli strumenti visti fino ad ora è quello in cui un'implementazione fornisce una sola soluzione che non è l'unica al problema. Il controllo che bisogna fare in questo caso è che la soluzione fornita appartenga all'insieme delle possibili soluzioni. Si parte nella procedura di validazione inserendo quest'ultimo nel file *expectedComplexSolution.txt* secondo lo schema da rispettare.

In un secondo momento si specifica la locazione del file e il nome del test creando così l'insieme delle soluzioni ammissibili del problema grazie alla classe `ReadFromFile`. Dopo aver ottenuto una soluzione dall'esecuzione, si passa a creare la soluzione ottenuta con i metodi della classe `PrepareTest` visti in precedenza per le soluzioni composte da un numero consistente di variabili. A questo punto è possibile fare il confronto tra la soluzione trovata dall'applicazione e l'insieme di soluzioni, verificando, tramite l'opportuna asserzione che quest'ultima gli appartenga. Un esempio che riporta la procedura descritta è il seguente:

```
IntegerSolution foundSolution = new IntegerSolution();
SetIntegerSolution expectedSolution = new
   SetIntegerSolution();
ReadFromFile readFromFile = new ReadFromFile();
String testSolutions(Solution solution){
  foundSolution = prepareSimpleIntegerTest(p, solution);
  readFromFile.setPathName("data/expectedComplexSolution.
   txt");
  readFromFile.setProblemName("MapColoringWithViolations");
  expectedSolution = readFromFile.readMoreSolution();
  try{
    AssertIsSolution(foundSolution, expectedSolution);
  }catch(Throwable e){
    e.printStackTrace();
    return "The result is not found in the expected result"
   ;
  }
  return "OK";
}
```

Da quello che si può notare quando le soluzioni assumono una complessità elevata, che non è più gestibile con il meccanismo delle asserzioni di *JUnit*,

la procedura che bisogna fare per controllare la loro correttezza è simile nei diversi casi. I punti che si devono seguire sono i seguenti:

1. Inserimento nell'apposito file della soluzione corretta da trovare o dell'insieme di soluzioni secondo gli schemi visti in precedenza.

2. Costruzione delle due soluzioni o dei due insiemi tramite gli appositi metodi delle classi `ReadFromFile` e `PrepareTest`.

3. Applicazione dell'asserzione `assertIsSolution(...)` per il controllo della correttezza delle soluzioni.

Capitolo 7

Risultati dei test per l'implementazione JSetl di JSR331

In questo capitolo si vedrà come gli strumenti fino ad ora visti riescono a fornire una serie di dati e risultati per coloro che stanno testando una determinata applicazione. Nel dettaglio si confronteranno i risultati che la libreria *JSetl*, sviluppata all'Università degli Studi di Parma, ottiene nei confronti di altri due importanti solver: *Constrainer-light* e *Choco*[7]. In seguito sono riportati i grafici che JavaTest Harness costruisce in base alle informazioni ricavate dal codice dei singoli test. Inoltre verrà fatto un confronto in termini di tempo e risorse acquisite da ogni solver nella risoluzione dei casi di test. In un primo momento saranno visti i risultati ottenuti sui test obbligatori che dichiarano la conformità dell'implementazione allo standard. Successivamente si passerà ai test facoltativi che oltre a misurare il livello di aderenza alle regole imposte nello standard, rilevano anche il grado di efficienza in termini di memoria e tempo d'esecuzione dei diversi solver. Per il caso di JSetl verrà inoltre stilata una lista, suddivisa per categoria di test, dei più comuni errori che si ottengono così da poter attuare correzioni nei casi possibili.

7.1 Attivazione e impostazione dell'ambiente d'esecuzione

Per avviare l'ambiente d'esecuzione JavaTest Harness è necessario collocarsi prima all'interno della cartella che contiene la suite di test da eseguire per una determinata implementazione (ovvero all'interno dello strumento TCK).

La struttura di questa directory è già stata discussa all'interno del capitolo 4 in merito al funzionamento di JavaTest Harness.

Posto di avere a disposizione il pacchetto jar **javatest.jar** il comando che esegue l'applicazione è il seguente:

```
java -jar lib/javatest.jar -newDesktop
```

A questo punto viene aperta un'interfaccia grafica che permette all'utente di scegliere se cominciare una nuova sessione d'esecuzione dei test oppure il recupero di una suite già eseguita con i relativi risultati. Per la trattazione dell'argomento si porrà il caso di una suite di test non ancora eseguita e che quindi necessita di avere impostati tutti i parametri necessari.

Come prima operazione necessaria bisogna indicare la locazione della suite di test. La cartella che la contiene necessita di avere al proprio interno il file **testsuite.jtt** che la identifica come una cartella strutturata per contenere i test da eseguire. Dopo viene creata la directory che dovrà contenere i risultati e che viene specificata dall'utente.

A questo punto viene presentata all'utente l'interfaccia di configurazione che permette di specificare i parametri d'esecuzione. Questa parte è totalmente personalizzabile dall'utente che può scrivere una specifica intervista da inserire nel pacchetto **javatest.jar** e utilizzarla successivamente. Nel caso in questione si utilizza il tipo standard messo a disposizione insieme ad altre tipologie che non saranno qui discusse. In questa fase di configurazione devono essere impostati diversi parametri, alcuni di carattere descrittivo e altri di carattere operativo per l'effettiva esecuzione della suite di test.

Configuration Name: un nome che identifica la configurazione che si sta creando.

Description: breve descrizione che rappresenta la configurazione da creare.

Modo di esecuzione della suite: scelta se eseguire la suite di test sulla macchina che si sta configurando in un opportuno processo, oppure se lasciare l'esecuzione ad un altro agente posto in una locazione diversa. Nel caso in oggetto si sceglie di eseguire i test all'interno della macchina che configura l'ambiente.

Specificazione della Java Virtual Machine: indicazione della locazione della Java Virtual Machine all'interno della macchina. Questa rappresenta l'agente di calcolo scelto quando si decide di eseguire i test sullo stesso hardware che configura l'ambiente d'esecuzione.

Indicazione del Class Path: questo parametro permette all'utente di indicare dove si trovano le classi compilate dell'applicazione che si intende eseguire. Solitamente la locazione ideale per i file compilati è all'interno della directory **classes** all'interno dello strumento TCK.

Indicazione di specifici test da eseguire: è possibile specificare determinati test da eseguire a fronte di altri presenti nella suite.

Indicazione di una Exclude List: con questo parametro, a fronte di test esclusi dall'esecuzione per i più disparati motivi, è possibili indicare una lista che li contiene e non permette loro di essere eseguiti. Se una Exclude List è necessaria, questa deve essere presente all'interno dello strumento TCK ed è rappresentata dal file con estensione **.jtx**, riconosciuta dall'ambiente JavaTest Harness.

Indicazione di una Failure List: solitamente i test che contengono errori possono essere inseriti anche all'interno di una Failure List, tuttavia questa pratica è poco utilizzata e tutti i test che hanno problemi confluiscono nella Exclude List.

Indicazione di un Stato per l'esecuzione: è possibili selezionare i test da eseguire in base allo stato che hanno generato in una precedente esecuzione. Tale scelta è utile quando i test che bisogna eseguire sono per la maggior parte test che non vengono superati. In questo modo è possibile isolarli e trascurare quelli che invece hanno avuto un buon esito.

Indicazione del numero di test in esecuzione contemporaneamente: in base alla potenza della macchina che contiene l'agente d'esecuzione è possibile specificare il numero di thread che devono eseguire i test in concorrenza. Per il caso in questione si è scelto di utilizzare due flussi d'esecuzione separati eseguendo così due test alla volta.

Indicazione del limite massimo di tempo: con questo parametro viene indicato il tempo massimo di esecuzione per un test di modo che non ne esistano alcuni in grado di mandare in stallo l'intera esecuzione.

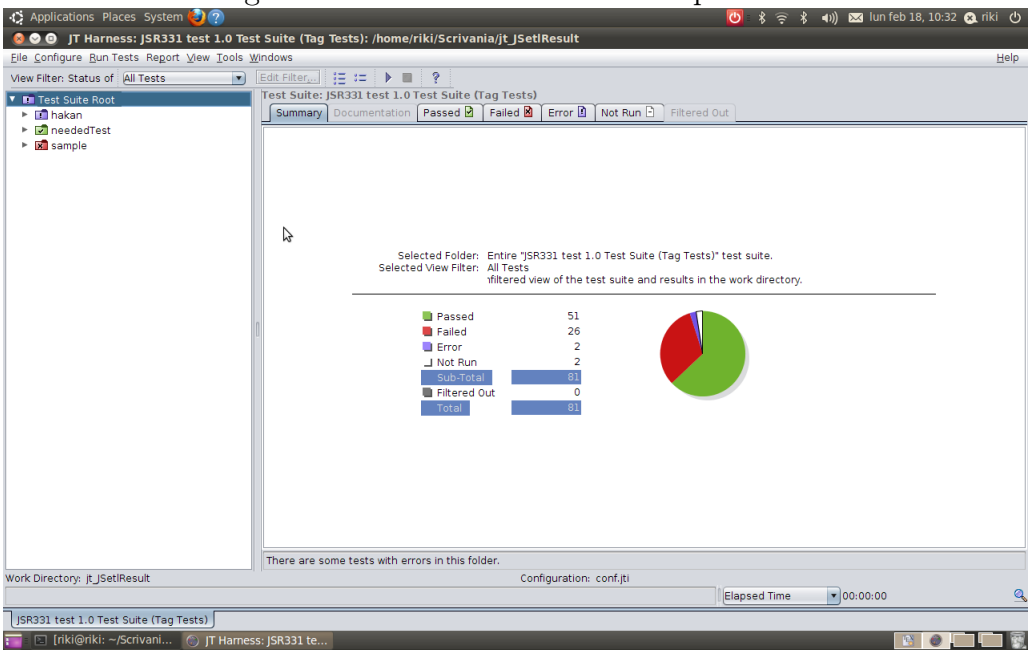
7.2 Esecuzione della suite per JSetl

Dopo aver opportunamente settato i parametri visti si apre all'utente un interfaccia grafica che riporta a sinistra la suite di test suddivisa secondo le tre tipologie viste, ovvero, **Test obbligatori, Sample, Hakan**. A destra

troviamo il grafico a torta che rappresenta i risultati in modo molto intuitivo secondo i diversi colori rappresentati dagli status. In alto troviamo la barra degli strumenti che permette di interagire con i test, mentre in basso troviamo la barra del progresso che riporta la percentuale d'esecuzione e lo status dei diversi test.

Per lanciare l'esecuzione dell'intera suite è necessario premere sul pulsante start e una volta eseguiti tutti i test, tranne quelli inclusi nella Exclude List, il risultato che si ottiene per JSetl è il seguente.

Figura 7.1: Ambiente d'esecuzione per JSetl

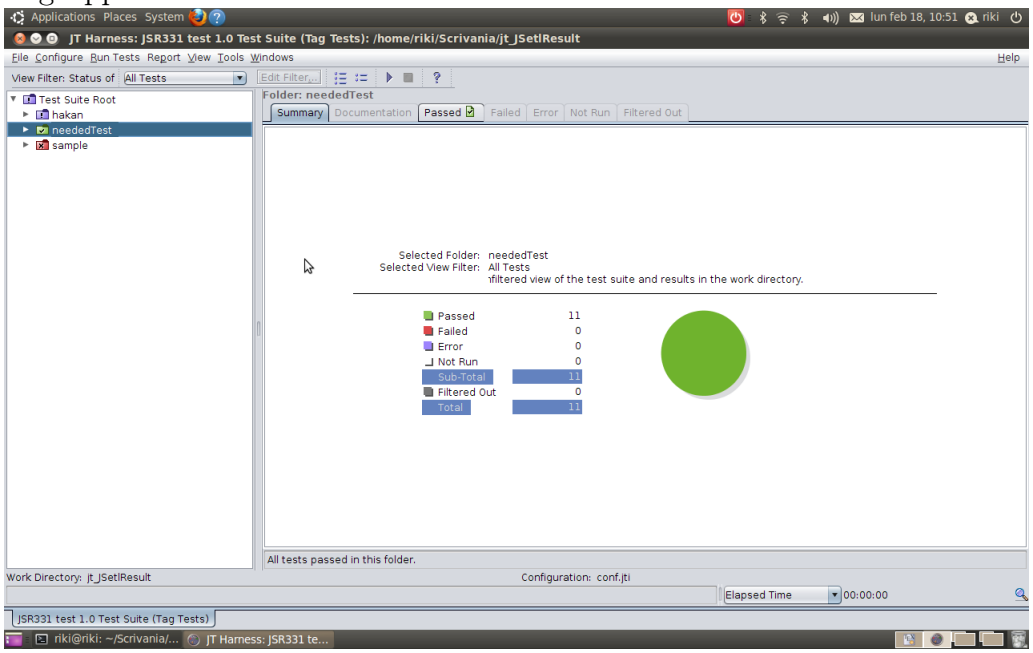


Su un totale di 81 test, 51 sono passati correttamente, 26 sono falliti, 2 hanno fornito un risultato che non è stato trovato nel limite di quelli che si potevano verificare, mentre 2 non sono stati eseguiti in quanto membri di una Exclude List. Il motivo per la quale sono stati inseriti all'interno di quest'ultima è il tempo d'esecuzione troppo elevato che non porta a nessun risultato e fa presumere l'ingresso in un loop infinito da parte dell'applicazione.

I risultati ottenuti per ogni categoria vedono i test obbligatori come correttamente eseguiti nella loro totalità, mentre i fallimenti sono consistenti per quanto riguarda i test di Hakan e sono presenti in misura più ridotta nella categoria dei Sample.

7.2.1 Risultati per i test obbligatori

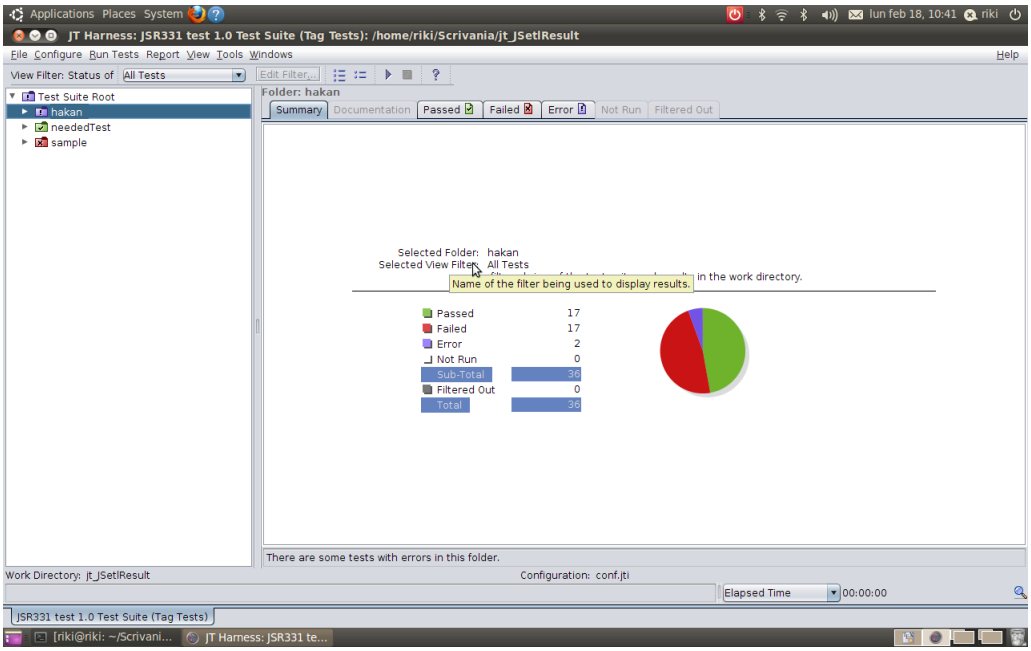
I test obbligatori servono a dichiarare se una determinata implementazione è conforme alle regole contenute all'interno del documento di specifica di JSR331. Come visto nel capitolo 6 questi test sono in un numero contenuto ed ognuno di essi si propone lo scopo di verificare il corretto funzionamento di un gruppo di metodi o di uno solo.



Come si può vedere JSelt supera correttamente tutti questi test. Per cui l'implementazione può essere ritenuta in linea con tutte le direttive.

Un confronto sui tempi di esecuzione e sull'efficienza dell'occupazione della memoria è superfluo. Infatti la semplicità che è presente nei test, espressa attraverso operazioni che richiamano metodi di rapida esecuzione, non richiede grandi occupazioni di memoria e i tempi per ottenere il risultato dovuto sono nell'ordine di 100 msec. Per questi motivi è inutile tenere dati e statistiche che riguardano i parametri di efficienza dell'implementazione.

7.2.2 Risultati per i Test di Hakan



Per i test di Hakan la percentuale di errori e fallimenti è la più elevata che si riscontri durante l'esecuzione. Infatti su un totale di 36 test, 17 vengono superati correttamente, 17 incontrano un fallimento, ovvero i risultati ottenuti non sono uguali a quelli attesi. Mentre i rimanenti due test trovano delle soluzioni che non sono verificabili, in quanto è stato posto un limite su quelle che l'applicazione doveva trovare.

L'eccezione che viene sollevata nella maggior parte di questi fallimenti è collegata ai diversi metodi di ricerca delle soluzioni:

- findOptimalSolution()

- findSolutionIterator()

- findSolution()

I test che non sollevano il tipo di eccezione visto in precedenza contengono errori riconducibili ai seguenti due casi:

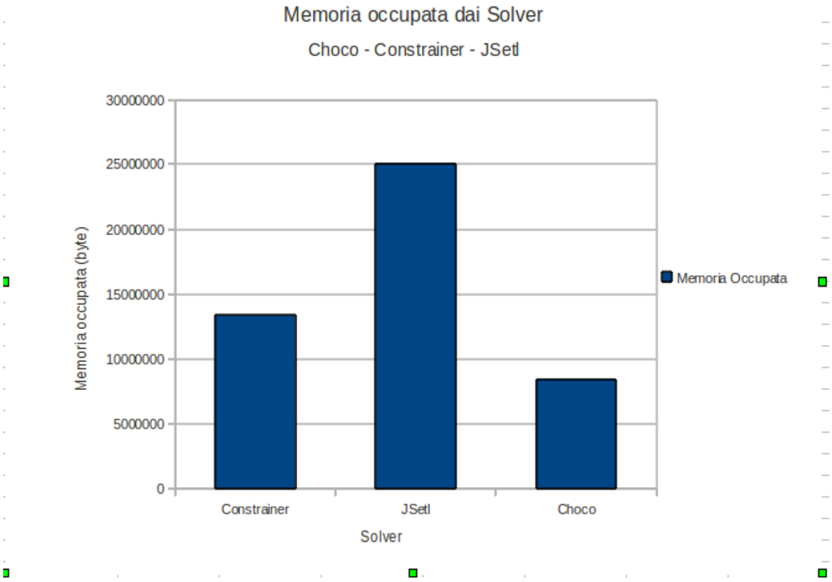
- Array out of Index

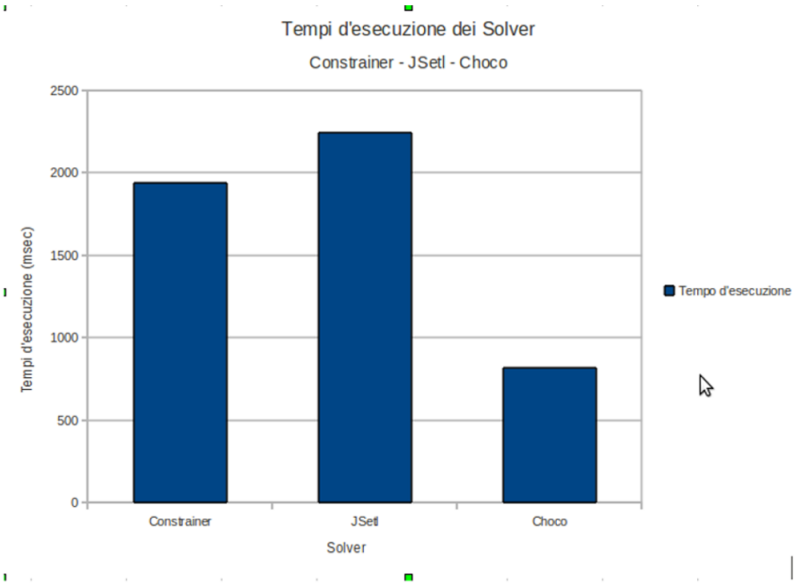
- Stack Overflow

Questa seconda tipologia è spesso riconducibile ad errori di programmazione presenti all'interno della libreria e sono facilmente individuabili quando sono presenti cicli e chiamate di funzioni ricorsive. Mentre per l'altro tipo di errori

l'individuazione dell'evento che li ha generati è più complesso. Infatti esistono lunghe code di chiamate di funzioni che devono essere risalite per trovare il metodo che ha generato l'eccezione.

Per questa tipologia di test, creata oltre che per testare il livello di standardizzazione dell'implementazione anche per valutarne il livello di adeguatezza in termini di occupazione di memoria e tempi d'esecuzione, si possono fare dei confronti con gli altri solver tramite dei grafici che evidenziano questi aspetti.

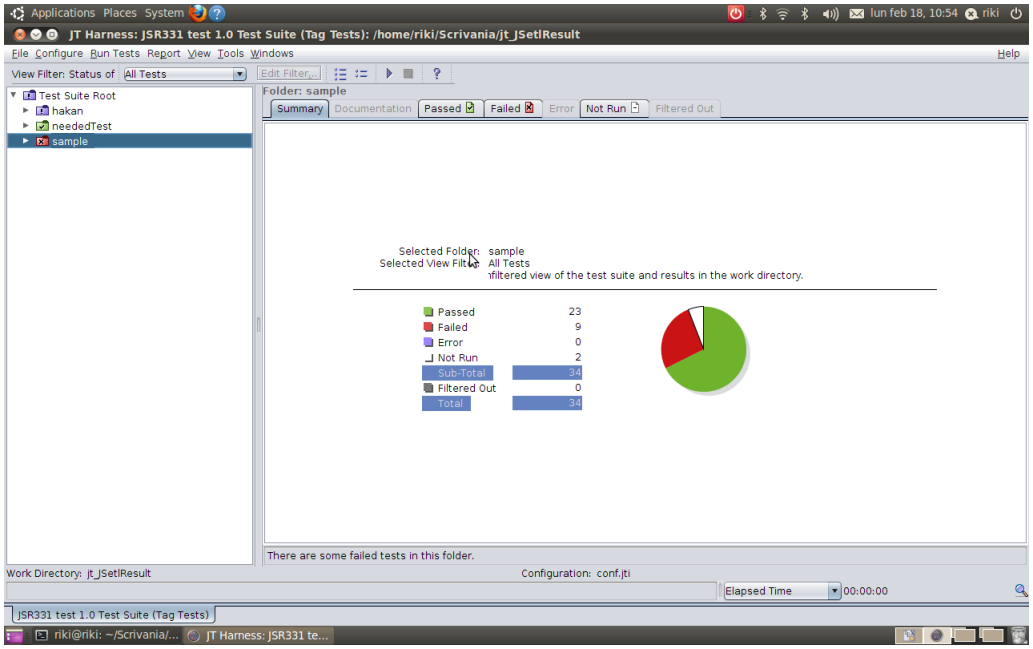




Da quello che si può dedurre guardando i risultati ottenuti si può concludere che JSetl che implementa lo standard JSR331 necessita ancora di ottimizzazioni per quanto riguarda specialmente il risparmio di memoria. Questo potrebbe essere possibile limitando il numero di variabili necessarie per determinare la soluzione ottimale e modificando tale procedimento che, come visto in precedenza, è quello più utilizzato in questa collezione di test.

7.2.3 Risultati per i Test Sample

Per quanto riguarda questa collezione di test, l'obiettivo che si propongono è quello di testare l'adattamento dell'implementazione JSetl allo standard JSR331. Inoltre si dà un'importanza agli aspetti che riguardano la sua efficienza coniugando il test di conformità o compatibilità con il test del prodotto. I problemi che sono presenti in questi programmi spesso pongono dei limiti semplici per validare l'adeguamento dell'implementazione, mentre utilizzano dei modi di ricerca delle soluzioni più complessi per verificare l'efficienza globale. I risultati ottenuti sono riportati di seguito nell'ambiente JavaTest Harness.



Come si può vedere su un totale di 34 test da eseguire, 23 sono eseguiti correttamente, 9 sollevano un'eccezione, mentre i restanti 2 test sono stati inseriti nell'apposita Exclude List e pertanto non vengono eseguiti. Il motivo di tale scelta è dovuto al fatto che in una prima esecuzione della suite questi non hanno portato ad alcun risultato sia esso corretto o errato. Ciò ha fatto presumere l'ingresso da parte dell'applicazione in un loop infinito.

Gli errori che maggiormente si sono riscontrati sono di due tipi:

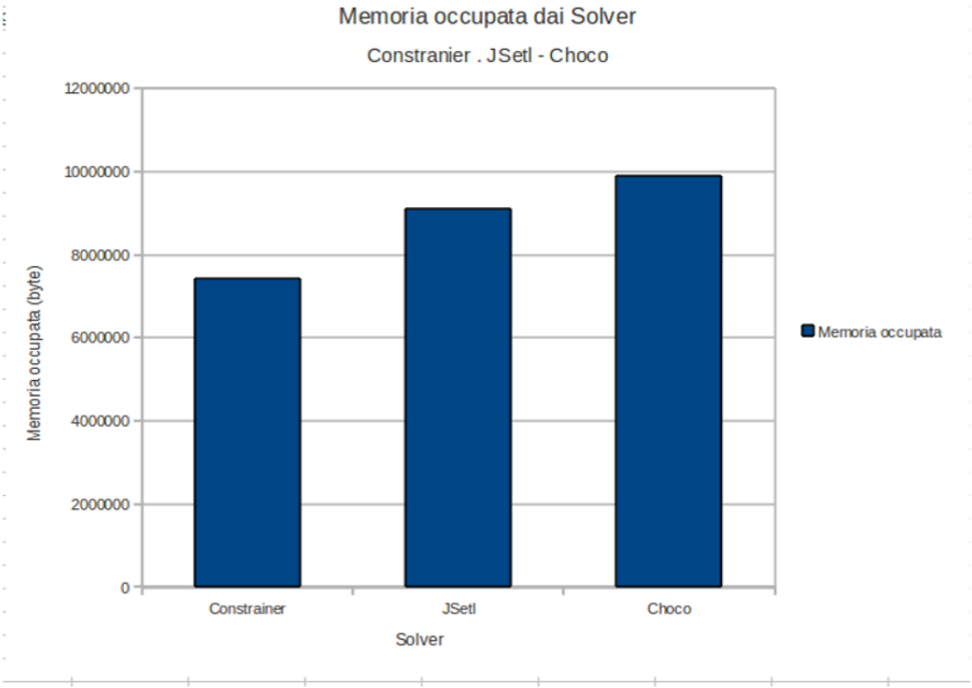
- Errori sui metodi di ricerca delle soluzioni

- **findSolution()**
- **findOptimalSolution(...)**
- **findSolutionIterator()**

- Soluzioni fornite che non sono uguali o in linea con quelle attese

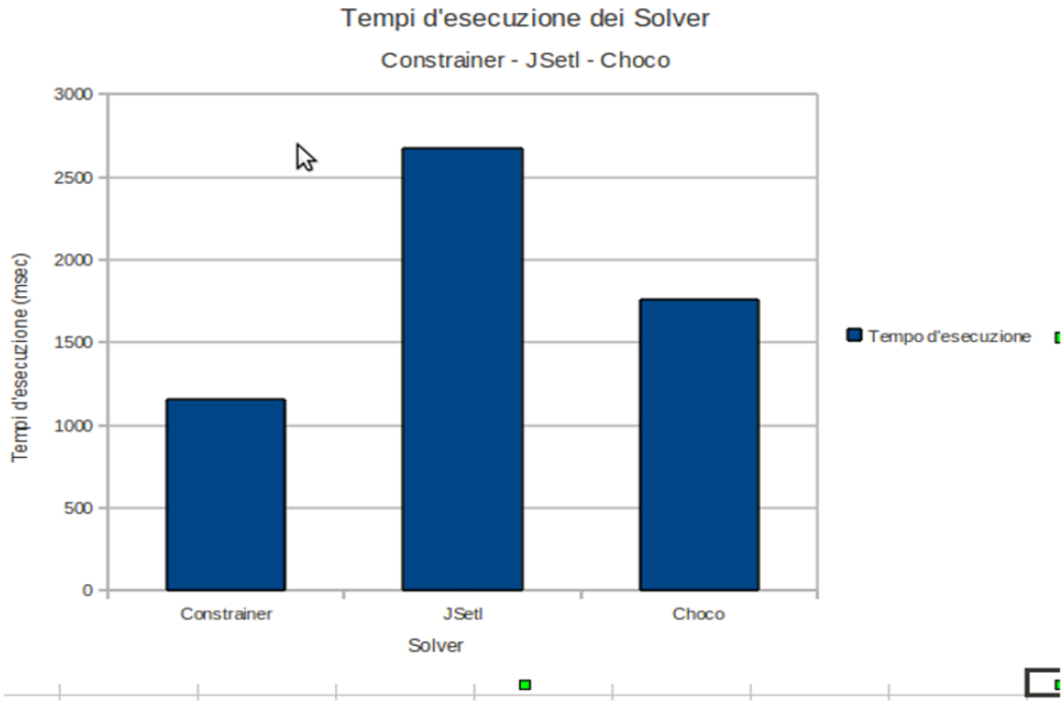
Per quanto riguarda la prima tipologia di errori è difficile, come per i test di Hakan, individuarne la fonte che ha generato questi tipi di eccezioni. Infatti è sicuramente necessario migliorare il metodo di ricerca delle soluzioni ottimali, che per altro è quello che genera più eccezioni. Il calcolo che attualmente viene fatto molto spesso non porta ad ottenere nessun risultato e genera quindi un fallimento.

Vista la natura di questi test si può fare un confronto nei termini di occupazione di memoria e tempi d'esecuzione. Per quanto riguarda il primo parametro si può guardare il grafico riportato di seguito.



L'occupazione di memoria in JSetl è abbastanza simile a quella tenuta dagli altri due solver. Tuttavia bisogna tenere in considerazione che il numero di test risolti da JSetl è inferiore a quello degli altri e inoltre la semplicità di questi test è ancora relativamente bassa. Pertanto possono ancora esserci ampi spazi di miglioramento per quanto riguarda l'utilizzo della memoria a partire dal miglioramento dei metodi per la ricerca delle soluzioni e dalla soluzione ottimale.

Il discorso sui tempi d'esecuzione è nettamente collegato alle funzioni che trovano le soluzioni ai diversi problemi. Se, infatti per ottenere i risultati ottimali di un problema è necessario occupare molta memoria e gestire un procedimento di ricerca poco efficiente, i tempi d'esecuzione si espandono notevolmente rispetto agli altri solver. La situazione sui tempi è riportata di seguito:



L'ambiente JavaTest Harness ha permesso la raccolta di tutte queste informazioni in modo molto più semplice rispetto a come venivano condotte prima le operazioni di testing.

Capitolo 8

Conclusioni e sviluppi futuri

In questo lavoro di tesi sono stati presentati diversi strumenti di supporto all'attività di testing di un sistema software, alcuni, sviluppati a partire da lavori già presenti, mentre altri creati senza alcuna base. Questi strumenti sono poi stati applicati al testing delle implementazioni della specifica Java JSR331.

La possibilità di automatizzare i test attraverso l'ambiente JavaTest Harness ha permesso di evidenziare errori e lacune presenti all'interno dei diversi solver con una velocità maggiore rispetto all'utilizzo dell'ambiente e editor *Eclipse* che prevedeva una serie di iterazioni manuali con l'utente.

Una delle novità che sono state introdotte sono i tool di controllo delle soluzioni in presenza di soluzioni complesse e/o multiple. In precedenza non erano presenti all'interno dello strumento TCK tali verifiche e si tendeva a controllare i risultati forniti confrontandoli con quelli che era prodotti dall'esecuzione delle diverse implementazioni.

L'applicazione degli strumenti di correttezza è risultata molto efficiente in quanto molte delle premesse necessarie al loro funzionamento sono state rispettate. Lo sviluppo di strumenti di contorno quali la classe per la lettura da file e quella per l'estrapolazione delle variabili e dei valori dalla soluzione sono stati necessari, ma sono applicabili solamente al caso concreto dello standard JSR331. Per quanto riguarda invece il package **solutionTool** contenente il codice necessario al controllo delle soluzioni può essere visto come un'estensione del package già esistente **junit.framework**. Infatti gli strumenti sono totalmente indipendenti dallo standard JSR331 e le eventuali relazione di ereditarietà possono essere modificate per aumentare leggibilità del codice e l'efficienza di quei metodi che creano e preparano le soluzioni.

L'utilizzo degli strumenti sviluppati in questo lavoro di tesi hanno portato, quindi, diversi vantaggi sotto due aspetti fondamentali:

Miglioramento della procedura di testing: ottenuto tramite l'utilizzo dell'ambiente d'esecuzione JT Harness e con l'inserimento nel codice dei test degli opportuni accorgimenti per comunicare con l'ambiente. Tra le migliorie che hanno avuto maggior rilievo vi sono:

- Una raccolta delle informazioni e dei risultati delle diverse implementazioni molto più ricca, in quanto l'interfaccia grafica presenta all'utente finale tutti i risultati delle esecuzioni dei test in tabelle e grafici.
- Un'interfaccia grafica per l'utente migliorata, molto più intuitiva, che facilita l'esecuzione dei test e permette di configurare, secondo le proprie esigenze, l'ambiente d'esecuzione.

Automatizzazione della procedura di testing: l'esecuzione dei singoli test e il confronto tra risultati ottenuti con quelli attesi prima era trattato solo a mano ed ora è stato reso automatico tramite gli strumenti sviluppati per trattare soluzioni multiple e complesse.

Sulla base dei lavori che sono stati svolti sarà possibile compiere diverse migliorie a quanto sviluppato.

Per quanto riguarda la libreria *JSetL* gli errori che sono evidenziati dall'ambiente d'esecuzione e riportati nel capitolo 7 possono essere classificati e utilizzati per individuare le eccezioni e porvi rimedio attraverso opportune modifiche del solver. In particolare, se possibile, andrà sicuramente migliorata l'efficienza in termini di occupazione di memoria e tempi d'esecuzione.

Un secondo lavoro possibile per il futuro e alla quale si sta già procedendo è il miglioramento della suite di test presente all'interno dello strumento TCK. Per quanto riguarda i test facoltativi si intende lasciarli nella forma che hanno avuto con le ultime modifiche effettuate e di cui si è ampiamente parlato all'interno del documento. Mentre, per i test obbligatori dello standard *JSR331*, si può pensare di apportare delle modifiche importanti.

Come prima cosa, l'intenzione è quella di scrivere test che sono guidati dai metodi che sono comuni alle diverse implementazioni che vogliono aderire alla specifica. Per ognuna delle diverse funzioni presenti all'interno delle classi descritte e richieste dallo standard si vuole scrivere uno o più casi di test specifici. In questo modo, a differenza di quanto è già presente nello strumento TCK, è possibile condurre una validazione del codice e delle implementazioni in maniera più efficace ed efficiente.

Per scrivere i casi di test si raggrupperanno i diversi dati di input in classi di equivalenza. Ovvero ogni elemento che appartiene ad una determinata classe se messo in input ad una specifica funzione ottiene lo stesso comportamento degli altri elementi presenti all'interno dell'insieme. In questo modo

per ogni rappresentante di ogni classi si scriverà un determinato caso di test e ciò permette di rilevare errori che ad oggi possono essere stati trascurati, vista la grossolanità dei casi di test scritti per la categoria di quelli obbligatori.

Questo nuovo pacchetto di test utilizzerà tutti gli strumenti di automatizzazione dell'esecuzione e di controllo di correttezza delle soluzioni visti in precedenza e i risultati saranno riportati dall'ambiente d'esecuzione *JavaTest Harness*.

Come secondo tipo di lavoro possibile, ma di una maggiore complessità, si può prendere in considerazione la costruzione di un "oracolo"[13] che possa generare in automatico i risultati per i singoli casi di test costruiti.

L'idea di base è quella di svincolare la generazione di soluzioni attese dal concetto di solver. Ovvero dato un dominio di valori e un insieme di vincoli quest'ultimi devono essere memorizzati in un opportuno stack e applicati per ogni valore del dominio. Tutti i risultati che si ottengono sono possibili soluzioni attese da confrontare con quelle ottenute dai diversi solver.

La creazione di questo strumento automatizzerebbe ancora di più la procedura di testing tramite l'ambiente JavaTest Harness. A questo andrebbe associato un ulteriore controllo che verifica la possibile correttezza della soluzione generata automaticamente. Si può pensare di creare questo strumento prendendo i diversi valori di cui è composta la soluzione e verificare se rispettano il dominio e soprattutto i vincoli imposti dai diversi problemi.

Se si riuscissero a creare questi strumenti, con una opportuna documentazione, si potrebbe effettuare un testing più accurato, completo e sicuro.

Bibliografia

[1] Jacob Feldman
JSR-331 Java Constraint Programming API SPECIFICATION
http://openrules.com/downloads/jsr331/JSR331.Specification.v081.pdf

[2] Sun Microsystem Inc.
JavaTM ME TCK Framework Developers Guide, version 1.2.1
http://www.docstoc.com/docs/56536909/Java%28TM%29-ME-TCK-Framework

[3] Sun Microsystems, Inc.
JavaTest TM Agent User's Guide JavaTest Harness, 4.2
http://docs.oracle.com/javame/testtools/javatest/javatest_agent.pdf

[4] Sun Microsystems, Inc.
JavaTestTM Architect's Guide JavaTest Harness
http://docs.oracle.com/javame/test-tools/javatest/javatest_arch_guide.pdf

[5] Sun Microsystems, Inc.
TCK Development Guidelines, version 1.0
http://jcp.org/aboutJava/communityprocess/tck/tck_guidelines-10_tr7.pdf

[6] Alessandro Fantechi
Il Testing
www.dsi.unifi.it/ fantechi/INFIND/testing.ppt

[7] Laburthe Francois, Jussien Narendra, Rochart Guillaume, Cambazard Hadrien
Choco User Guide
http://choco.sourceforge.net/useguide.pdf

[8] JSetL Home Page
http://cmt.math.unipr.it/jsetl.html

[9] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[10] Gianfranco Rossi, Roberto Amadini
JSetL User's Manual Version 2.3
Quaderni del Dipartimento di Matematica, n. 507, Universita' di Parma, 24 Gennaio 2012.

[11] Sum Microsystem Inc.
JavaTM Technology Test Suite Development Guide 1.2
`http://jcp.org/aboutJava/communityprocess/ec-public/TCK docs/tsdg.pdf`

[12] Brian Marick
When Should a Test Be Automated?
`http://www.exampler.com/testing-com/writings/automate.pdf`

[13] Davide Balzarotti, Paolo Costa
Generazione Automatica e Valutazione di Casi di Test
`http://home.dei.polimi.it/ghezzi/PRIVATE/Balza-Costa.pdf`

[14] Fabio Biselli
Contributo alla specifica JSR-331 mediante un'implementazione basata su JSetL, Tesi di laurea per il corso id Informatica
`Università degli studi di Parma`

Appendice A

Generazione di Test Case

Per tutto il lavoro svolto fino ad ora e presentato nei capitolo precedenti non si è mai posto il problema di generare completamente dall'inizio tutta la suite di test che si propone il compito di testare il livello di standardizzazione di una qualunque implementazione delle specifiche per JSR331. Infatti tutti i casi di test necessari allo scopo erano già presenti all'interno dello strumento TCK che doveva validare le diverse applicazioni. Quelle che si è fatto è stato per migliorare tutta la fase di testing rendendola il più possibile automatica e affidabile.

In questa appendice, invece, si vedranno le tecniche che si possono utilizzare nella generazione di casi di test per la validazione di determinati programmi. Tali modalità potranno sicuramente poi essere applicate per generare futuri test che integreranno tutta la suite già presente nel TCK.

I componenti principali di un sistema di generazione di casi di test sono i seguenti:

Test Data Generator: partendo dai dati contenuti nella specifica o dal codice sorgente dell'implementazione genera l'insieme dei casi di test utilizzati per la validazione.

Oracle: l'oracolo è il componente più difficile da realizzare e spesso, per la natura delle soluzioni e dei sui stessi componenti diventa irrealizzabile. Esso incaricato di calcolare i valori di uscita attesi per un dato insieme di casi di test. Per questo scopo sono impiegate specifiche eseguibili oppure su asserzioni presenti nel codice basate su invarianti pre o post condizioni.

Test Manager: è il componente che si occupa di coordinare tutti gli altri nel processo di generazione.

File Comparator: confronta i risultati ottenuti con quelli attesi e segnala eventuali incongruenze.

Report Generator: è il componente che si occupa di presentare in un formato leggibile all'utente tutti i risultati ottenuti per ogni caso di test.

Per quanto riguarda questo sistema si possono trovare somiglianze con quanto visto per lo strumento TCK migliorato di JSR331. Infatti il Test Manager è svolto dall'ambiente JavaTest Harness che avendo i diversi casi di test generati in precedenza avvia la fase di validazione e coordina tutti i componenti. Il ruolo del File Comparator è adempiuto invece della classi che verificano la correttezza delle soluzioni ovvero `PrepareTest`, `AssertionSolution` e le classi che le rappresentano *SetIntegerSolution*, `IntegerSolution`, `StringSolution` e `SetStringSolution`. Il Report Generator è contenuto sempre nell'ambiente JavaTest Harness. Per quanto riguarda invece il Test Data Generator non è stato necessario, infatti i casi di test erano già tutti presenti all'interno delle suite, ma saranno comunque viste le tecniche di generazione. L'oracolo rimane il componente più complesso da realizzare e non sempre è adattabile a tutte le tipologie di test che si intendono svolgere e per tanto mancante in molti di questi sistemi. Per il caso dello standard JSR331 le soluzione attese da ogni test sono già fornite assieme alla suite, in quanto, si tratta di problemi con soluzioni che possono essere calcolate a priori anche da altri programmi.

Le principali tecniche di generazione si suddividono in due tipologie:

- Black-Box testing dove i casi di test sono selezionati sulla base delle direttive funzionali dei programmi o contenute nei documenti di specifica.
- White-Box testing dove i casi di test sono generati in base al codice contenuto nei programmi da validare.

A.1 Black-Box testing

Questa tipologia di testing è la più difficile da implementare in quanto, spesso non si hanno a disposizione specifiche complete che riguardano il programma da validare e la generazione di questo tipo risulta oggettivamente complessa. Le principali tecniche che utilizzano una metodologia a Black-box (ovvero basata sulle specifiche) sono le seguenti:

Classi di Equivalenza: il dominio degli ingressi viene partizionato in classi di equivalenza nei confronti delle quali il programma si comporta

allo stesso modo. Un minimo insieme di casi di test può essere ottenuto estraendo un rappresentate per ogni classe di equivalenza e scrivendo attorno ad esso un caso.

Valori limite: è possibile generare casi di test attorno a quei valori che stanno ai limiti del range di ammissibilità come ad esempio: operazione di push da una pila vuota, oppure l'esplorazione di un albero vuoto.

Guidati da eventi: per testare gli applicativi che sono event-oriented come le interfacce grafiche per l'utente, si possono generare automaticamente sequenze ammissibili di eventi. Si possono quindi costruire test che testino separatamente ogni tipo di evento.

A.2 White-Box testing

Questa tecnica di testing detta anche del test strutturale è molto più utilizzata rispetto a quella vista in precedenza e per tanto più trattata nel seguito. Il nuovo modo di procedere fa riferimento al control flow graph (CFG) del programma.

Definizione A.1. Un CFG per un dato programma F è un grafo diretto G = (N, E, s, e) e dove N è un insieme di nodi, E un insieme di archi ed e s e e sono rispettivamente il punto d'ingresso e i punti di uscita dal programma. Ciascun nodo n $\in$ N davverocorrisponde ad un'istruzione del programma mentre un arco e = (ni , nj) rappresenta il trasferimento del controllo dal nodo ni al nodo nj.

Definizione A.2. (Cammino). Un cammino P attraverso un CFG è una sequenza P = (n1 , n2 , . . . , nm) tale che $\forall$i((1 $\leq$ i $<$ m) $\Rightarrow$ (ni , ni + 1) $\in$ E) Un cammino è detto ammissibile se esiste un insieme di valori di ingresso per cui il cammino è attraversato.

La prima tecnica che utilizza il metodo del white-box è il *random testing*. Esso può esser utilizzato per generare valori di ingresso per ogni tipo di programma dal momento che ogni tipo di dato è rappresentabile come una sequenza di bit. Per ciascuna funzione da testare è quindi possibile generare casualmente sequenze di bit e utilizzarle come parametro d'ingresso. Le performance di questa tecnica sono alquanto scadenti in termini di copertura del codice, limitando così la capacità di scovare errori di carattere locale. Questa tecnica è poco utilizzata e viene usata come benchmark per verificare l'efficacia di altre soluzioni.

Un secondo modo di generare casi di test è seguendo la tecnica del *path-oriented testing*. Con questa maniera di operare si prova ad individuare il cammino che porta all'istruzione che si vuole analizzare e quindi generare un insieme di valori di ingresso per quel cammino. A seconda che l'analisi vengo condotta staticamente o dinamicamente possiamo individuare due diverse categorie.

Per quanto riguarda il modo di operare statico ci si basa sull'analisi del programma senza richiederne una esecuzione. Si assegnano vincoli e espressioni alle variabili durante l'attraversamento del codice. In questo modo si ottiene un sistema di disuguaglianze in funzione delle variabili di ingresso che descrive le condizioni necessarie per attraversare un dato cammino. Nella formulazione generale il problema di risoluzione del sistema è NP-completo, tuttavia se i vincoli sono lineari è risolvibile con tecniche di programmazione lineare. Tuttavia, il metodo statico porta a delle complicazioni durante la creazione del sistema lineare dovute all'utilizzo di array e puntatori che creano alias e alla gestione delle chiamate di funzioni.

Per ovviare a questi problemi si può operare in maniere dinamica ovvero eseguendo un'analisi a run-time del programma. Durante l'esecuzione viene monitorato il flusso del programma. Quando si verifica una deviazione si individuano le variabili in ingresso responsabili del comportamento indesiderato. Un volta trovate si crea il caso di test sulle medesime per validare così un cammino.

Un terza tecnica di generazione è la *goal-oriented testing*.

Definizione A.3. (Cammino non specifico). Dati due cammini p e w diciamo che la loro concatenazione pw è un cammino non specifico se p e w non sono adiacenti, ovvero se (last(p), f irst(w)) $\in$ E.

Definizione A.4. (Chiusura). Dato un cammino non specifico u = p1 p2 . . . pn , si definisce la chiusura di u, indicata come u$*$, l'insieme di tutti i cammini p1 q1 p2 q2 . . . qn$-$1 pn .

L'approccio goal-oriented è più efficiente dell'approccio precedente perché permette di ottenere delle linee guida verso un dato insieme di cammini. Invece di cercare di generare valori di ingresso che attraversino il programma dall'inizio alla fine, genera input per un dato cammino non specifico u. Di conseguenza, per il generatore è sufficiente trovare dei valori di ingresso per uno qualsiasi tra i cammini (specifici) p $\in$ u$*$.

Per esempio, per soddisfare un criterio di copertura strutturale come la copertura delle istruzioni, un cammino deve essere selezionato per ciascuna istruzione non coperta. L'approccio goal-oriented permette di rimuovere questo vincolo. Questo è ottenuto tramite la classificazione delle diramazioni del

CFG del programmo come critica, semi-critica e non essenziale rispetto al nodo obiettivo. Una diramazione critica è un arco del CF G che impedisce di raggiungere il nodo obiettivo. Pertanto, una funzione obiettivo dovrà forzatamente selezionare la diramazione opposta per giungere al nodo obiettivo. In caso la diramazione alternativa non sia percorribile il processo termina senza aver raggiunto l'obiettivo.

Una diramazione semi-critica è una diramazione che conduce al nodo obiettivo ma solo attraverso l'arco di ritorno di un ciclo. In questo caso, se possibile, viene ancora selezionata la diramazione alternativa. Tuttavia, se questo non fosse possibile, il processo non termina ma l'esecuzione continua nella speranza che all'iterazione successiva del ciclo la diramazione alternativa diventi effettivamente percorribile.

Infine, una diramazione non-essenziale è una diramazione che non è ne critica ne semi-critica. In pratica una diramazione che non determina in alcun modo se il nodo obiettivo sarà raggiunto o meno.

Le due tecniche più note che utilizzano questo approccio sono quelle basate sulle catene e quelle basate sulle asserzioni.

Le asserzioni sono condizioni che vengono inserite nel codice per verificarne la corretta esecuzioni. Possono esprimere pre e post-condizioni oppure invarianti. Se un'asserzione non viene soddisfatta, vuol dire che esiste un errore o nell'implementazione del programma oppure nell'asserzione stessa. L'obiettivo del test basato su asserzioni è di trovare un qualsiasi cammino che conduce all'asserzione. Un ulteriore vantaggio fornito da questa tecnica è che l'oracolo viene direttamente fornito nel codice. In tutte le altre tecniche fin qui discusse l'oracolo doveva essere fornito separatamente. Viceversa, qui, il valore atteso per ogni esecuzione è direttamente presente nell'asserzione stessa.

L'approccio a catena utilizza il concetto di sequenza di eventi come un passo intermedio per stabilire il tipo di cammino richiesto per l'esecuzione fino al nodo obiettivo. Una sequenza di eventi è una successione di nodi del programma che devono essere eseguiti. La caratteristica di questo approccio è di utilizzare dipendenze sui dati per individuare una catena di nodi che sono vitali per soddisfare un dato criterio di copertura e poi utilizzare l'approccio sopra-esposto per raggiungere questi nodi.

A.3 Criteri di adeguatezza

Dopo la fase di generazione è utile verificare l'adeguatezza dei casi di test generati rispetto al programma che intendono validare. Nella sua definizione originale, un criterio di adeguatezza è un predicato che definisce quali pro-

prietà di un programma devono essere esercitate da un insieme di test che sia completo, vale a dire un insieme di test il cui successo implica che non ci siano errori nel programma sotto esame. Per essere conforme a questa definizione, un criterio di adeguatezza deve soddisfare due proprietà:

Affidabilità: diciamo che un criterio è affidabile se produce sempre risultati consistenti. Questo significa che se un programma supera tutti i casi di test di un insieme che soddisfa il criterio, allora deve superare anche ogni altro insieme di casi di test che soddisfi il medesimo criterio.

Validità: diciamo che un criterio è valido se per ogni errore in un programma, esiste un insieme di test che soddisfa il criterio e che è in grado di rilevare l'errore.

Purtroppo è stato dimostrato che non esiste nessun criterio computabile che soddisfi questi due requisiti. Per questo motivo, la ricerca si è orientata nella definizione di criteri 'approssimati' che siano applicabili in pratica. Lo scopo di tali criteri è fornire una metrica per la valutazione della qualità di una suite di casi di test.

I criteri sono suddivisi in tre categorie

Criteri basati sui difetti : misurano l'abilità di un insieme di test nel rivelare difetti nel software.

Criteri basati sugli errori : misurano quanto un insieme di casi di test stressa il programma in particolari punti che sono considerati ad alto rischio in base all'esperienza di come e dove un programma tende a non soddisfare le sue specifiche.

Criteri strutturali : definiscono una metrica per la copertura di qualche insieme di elementi legati al programma o alle sue specifiche.

I criteri di copertura del codice sono quelli di gran lunga più utilizzati in pratica e godono di un certo successo anche in ambito aziendale. Questo è dovuto principalmente a due fattori: (1) all'esistenza di tool in grado di automatizzare il processo di analisi e (2) alla capacità di fornire facilmente un singolo numero (la copertura appunto) molto semplice da capire anche ai non esperti e che riassume in modo estremamente sintetico l'estensione dei test che sono stati effettuati fino a quel momento. I criteri di copertura si basano sul fatto che qualsiasi programma può essere rappresentato sotto forma di un grafo in cui i nodi rappresentano le istruzioni (o, a differenti livelli di astrazione i metodi, le classi . . .) e gli archi una qualche forma di dipendenza tra di esse (tipicamente nel flusso di controllo o nel flusso dei dati). A partire

da una formulazione sotto forma di grafo, un criterio di copertura non è altro che una metrica che misura la porzione di grafo che è stata attraversata nell'esecuzione dei casi di test. E' importante notare come per particolari programmi è possibile che nessuna test suite sia in grado di ottenere una copertura completa. Ciò può essere dovuto ad esempio alla presenza di sezioni di codice che non possono mai essere eseguite. Per scongiurare questa possibilità talvolta si ricorre a criteri "modificati" in cui è richiesto che soltanto la parte raggiungibile del programma sia coperta. Purtroppo, il problema di determinare se una porzione di codice è raggiungibile o meno è noto essere indecidibile e quindi anche il problema di sapere se una copertura del 100 per 100 è raggiungibile o meno è anch'esso indecidibile.

Si vedranno ora quali sono i criteri di copertura che sono maggiormente utilizzati. Per quanto riguarda la copertura di control-flow graph le tecniche più note sono:

Copertura degli statement : il criterio di adeguatezza misura la percentuale di statement del programma che sono stati eseguiti dalla test suite sotto esame. E certamente il criterio più semplice ed anche il più facile da soddisfare.

Copertura dei branch : e simile al precedente, tuttavia al posto di misurare la copertura dei nodi, misura la percentuale di copertura degli archi. In pratica, per ciascuna istruzione condizionale, cerchiamo di coprire entrambi gli archi uscenti.

Copertura dei cammini : consiste nel determinare quanti fra tutti i possibili cammini nel control-flow graph sono coperti da una certa test suite. Il problema è che il numero dei cammini è spesso infinito e quindi questo criterio viene generalmente ristretto ponendo limiti al numero massimo di volte in cui deve essere percorso un ciclo.

Copertura delle decisioni : assomiglia al criterio di copertura dei branch, da cui differisce per il fatto che ora non solo vogliamo coprire tutti gli archi uscenti da ciascuna condizione, ma vogliamo farlo per ogni possibile combinazione dei valori di verità dei predicati.

Per quanto riguarda invece le tecniche basate sul data-flow graph distinguiamo:

Copertura delle definizioni : Una test suite è adeguata se per ciascuna definizione di una variabile x, viene testato almeno un cammino che da tale definizione porta ad un uso della variabile.

Copertura degli usi : Una test suite è adeguata se viene testato almeno un cammino che porta da ciascuna definizione di una variabile x a ciascun uso della stessa.

Copertura dei cammini DU : Una test suite è adeguata se vengono testati tutti i possibili cammini che portano da ciascuna definizione di una variabile a ciascun uso della stessa.

Fino ad ora sono stati analizzati i criteri di adeguatezza basati sul codice del programma. Qualora il codice non fosse disponibile è talora possibile estendere alcune delle tecniche precedenti in modo da operare sulle specifiche formali. In particolare, nel caso di specifiche model-based molti dei ragionamenti già visti sono ancora applicabili. In questo tipo di specifiche esiste infatti una qualche forma di descrizione dello spazio degli stati in cui si trova il software e delle possibili transizioni tra di essi. In tal caso possono essere definiti criteri di copertura di combinazioni di predicati e/o dei dati di input. Appartengono a questa categoria anche le specifiche fornite sotto forma di speciali grafi (ad esempio gli automi a stati finiti) a cui ben si presta l'applicazione delle tecniche di copertura dei nodi e degli archi viste nelle sezioni precedenti. Nel caso in cui le specifiche siano invece fornite sotto forma di espressioni algebriche è possibile fare molto poco. Sono state proposte alcune tecniche applicabili in questo contesto (come la copertura dei valori, eventualmente partizionati in base alla complessità).

Appendice B

Codice modificato del test AllIntervall

Di seguito è riportato, per intero, il codice del test modificato, per poter essere eseguito nell'ambiente d'esecuzione JT Harness, che è presente nel capitolo 6.

```
import java.io.PrintWriter;
import javax.constraints.*;
import com.sun.javatest.Test;
import com.sun.javatest.Status;
import solutionTool.*;

/**
* @test
* @sources AllInterval.java
* @executeClass AllInterval
*/
public class AllInterval extends AssertionSolution
  implements Test{
    int n;
    Var[] x;
    Problem p = ProblemFactory.newProblem("All Interval");
    static Status s = Status.passed("Test AllInterval
  passed without exception");
    Solution[] sol = new Solution[100];

    public Status run(String[] args, PrintWriter out,
  PrintWriter err) {
      String stringErr;
        int n_in = 10;
```

```
      if (args.length >= 1) {
          n_in = Integer.parseInt(args[0]);
      }
      System.out.println("\nn: " + n_in + "\n");
    stringErr = testRepresentation(n_in);
    if (stringErr != "OK"){
      return s = Status.failed(stringErr);
    }
    stringErr = testResolution();
    if(stringErr != "OK"){
      return s = Status.failed(stringErr);
    }
    stringErr = testSolutions(sol);
    if(stringErr != "OK"){
      return s = Status.failed(stringErr);
    }
    return s;
  }
  public static void main(String[] args) {
  PrintWriter err = new PrintWriter(System.err,true);
  Test t = new AllInterval();
  s = t.run(args,null,err);
  s.exit();
  }
  String testRepresentation(int n_in){
      n = n_in;
      x = p.variableArray("x", 1, n, n);
      diffs = p.variableArray("diffs", 1, n-1, n-1);
      try{
        p.postAllDifferent(x);
      }catch(Throwable e){
        e.printStackTrace();
        return "Method postAllDifferent(x) failed";
      }
      try{
        p.postAllDifferent(diffs);
      }catch(Throwable e){
        e.printStackTrace();
        return "Method postAllDifferent(diffs) failed";
      }
      for(int k = 0; k < n-1; k++) {
        try{
          p.post(diffs[k],"=", x[k+1].minus(x[k]).abs());
```

```
        }catch(Throwable e){
          e.printStackTrace();
          return "Method post(diffs[k],'=', x[k+1].minus(
x[k]).abs()) failed";
        }
      }
      try{
        p.post(x[0], "<", x[n-1]);
      }catch(Throwable e){
        e.printStackTrace();
        return "Method post(x[0], '<', x[n-1]) failed";
      }
      try{
        p.post(diffs[0], "<", diffs[1]);
      }catch(Throwable e){
        e.printStackTrace();
        return "Method post(diffs[0], '<', diffs[1])";
      }
    return "OK";
  }
  String testResolution(){
      Solver solver = p.getSolver();
      SearchStrategy strategy = solver.getSearchStrategy
();
      try{
        strategy.setVars(x);
      }catch(Throwable e){
        e.printStackTrace();
        return "Method setVars(x) failed";
      }
      try{
        strategy.setVarSelectorType(VarSelectorType.
MIN_DOMAIN_OVER_WEIGHTED_DEGREE);
      }catch(Throwable e){
        e.printStackTrace();
        return "Method setVarSelectorType(VarSelectorType
.MIN_DOMAIN_OVER_WEIGHTED_DEGREE) failed";
      }
      try{
        strategy.setValueSelectorType(ValueSelectorType.
MIN_MAX_ALTERNATE);
      }catch(Throwable e){
        e.printStackTrace();
```

```
        return "Method strategy.setValueSelectorType(
ValueSelectorType.MIN_MAX_ALTERNATE) failed";
      }
      try{
      SolutionIterator iter = solver.solutionIterator();
      }catch(Throwable e){
        e.printStackTrace();
        return "Method solutionIterator() failed";
      }
      while (iter.hasNext()) {
          num_sols++;
          Solution s = iter.next();
          for(int i = 0; i < n; i++) {
              System.out.print(s.getValue("x-"+i) + " ");
          }
          sol[i] = s;
          System.out.println();
      }
      System.out.print("\nIt was " + num_sols + "
solutions.\n");
      solver.logStats();
    return "OK";
  }
  SetIntegerSolution foundSolution = new
SetIntegerSolution();
  SetIntegerSolution expectedSolution = new
SetIntegerSolution();
  ReadFromFile readFromFile = new ReadFromFile();
String testSolutions(Solution[] solution){
  foundSolution = prepareComplexTest(x, solution);
  readFromFile.setPathName("data/expectedComplexSolution.
txt");
  readFromFile.setProblemName("AllIntervall");
  expectedSolution = readFromFile.readMoreSolution();
  try{
    isASubSet(foundSolution, expectedSolution);
  }catch(Throwable e){
    e.printStackTrace();
    return "Solutions found doesn't belong to the
solution set";
  }
  return "OK";
}
```

```
}
```