UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI

Corso di Laurea in Informatica

Tesi di Laurea

Test della libreria JSetL tramite JUnit

Relatore:
Prof. Gianfranco Rossi

Candidato:
Michele Giacobini

Anno Accademico 2012/2013

Ai miei genitori Roberto e Carla,
alle mie sorelle Antonella, Fiamma e Sara,
alla mia fidanzata Martina

Indice

Capitolo 1

Introduzione

Nello sviluppo di un prodotto software una delle ultime tappe, ma non per questo meno importante, è la fase di test. Ogni blocco di codice o modulo realizzato in rari casi è esente da errori e per cercare di limitarli questa fase è necessaria [2]. Esistono varie tipologie di test, ma prima è meglio cercare di capire che cos'è e come opera.

A livello intuitivo, un test per un programma **P** è rappresentato da un predefinito input per **P** per il quale sono noti i risultati e da uno strumento in grado di confrontare i valori attesi con quelli risultanti in output dall'esecuzione di **P**. Detto ciò si potrebbe pensare che basti conoscere tutti i possibili valori di ingresso del programma per eseguire un test completo ed esaustivo. In realtà non è così semplice, in quanto un programma normalmente opera su enormi quantità di input validi, talvolta addirittura infiniti.

Un modo per ovviare a questo problema è quello di suddividere l'insieme degli input in classi di equivalenza, in modo tale che testando ogni rappresentante si raggiunga una buona copertura degli input.

Anche se potessimo comunque testare tutti gli input non si avrebbe mai la certezza di generare codice esente da errori. Infatti, secondo la ***tesi di Dijkstra***, l'esecuzione di un test può rilevare la presenza di errori, ma non dimostrarne l'assenza.

Fondamentale per la fase di test è pertanto la selezione di un insieme di input validi che riescano a testare il programma nel modo più completo possibile.

Esistono diverse tipologie di test che mirano a testare differenti aspetti di un programma, come ad esempio il *test di integrazione* che mira alla correttezza delle interfacce, oppure il *test di sistema* che mira a testare l'affidabilità, la sicurezza e le prestazioni del programma.

La metodologia di test considerata in questa tesi sarà quella del *test unitario* che mira alla correttezza degli algoritmi. Questa tipologia di test viene attuata su singole unità software ovvero sui componenti minimi di un programma dotati di funzionamento autonomo. In altre parole questo tipo di test viene effettuato per verificare che un requisito sia soddisfatto dal codice che è stato scritto. A seconda del linguaggio di programmazione, una singola unità autonoma può corrispondere per esempio a una singola funzione nella programmazione procedurale, oppure una singola classe o un singolo metodo nella programmazione orientata agli oggetti.

Per quanto riguarda il *test unitario* in ambiente *Java*, è necessario individuare quali siano le singole unità che devono essere testate. Queste sono rappresentate dalle singole classi, per ciascuna delle quali sarà pertanto eseguito un singolo test allo scopo di verificare il corretto funzionamento dei propri metodi.

L'idea del *test unitario* in questo ambiente di sviluppo è quella di valutare ogni singolo metodo di una classe in funzione dei valori attesi. *Java* mette a disposizione il tool **JUnit** per l'automazione del test di unità in modo tale da rendere più semplice ed immediato lo sviluppo dei singoli test (**test case**). **JUnit** è uno strumento molto efficiente che permette anche l'organizzazione di più test in un singolo file (**test suite**), rendendo possibile l'esecuzione di un numero arbitrario di test in modo automatizzato ottenendo una risposta semplice ed esauriente e dando anche la possibilità di vedere in dettaglio in quali casi è fallito il test.

In questo lavoro di tesi i test saranno sviluppati per la libreria *JSetL*. Questo è uno strumento realizzato interamente in *Java* presso il Dipartimento di Matematica dell'Università di Parma e combina le caratteristiche dei linguaggi *object-oriented* con quelle dei linguaggi *dichiarativi*. Questa categoria di linguaggi si focalizza sulla descrizione delle proprietà della soluzione desiderata (il *cosa*), lasciando indeterminato l'algoritmo da usare per trovare la soluzione (il *come*), che invece caratterizza i linguaggi *imperativi*. Questa libreria è già stata in parte testata in precedenti lavori di tesi [15] [16].

Nella prima parte di questo lavoro di tesi si riprenderà ed approfondirà l'analisi di alcune classi fondamentali di *JSetL* per l'individuazione dei relativi casi di test attraverso la tecnica dell'*equivalence partitioning test*, che prevede la suddivisione del dominio di input in classi di equivalenza in modo tale da poter testare un rappresentate per ogni classe.

Il problema principale nell'utilizzo di *JUnit* per l'automazione dei test è che questo strumento anche se efficiente permette di eseguire test solamente su soluzioni composte da una singola variabile. Diventa pertanto laborioso

il testing di *JSetL* in tutti quei casi in cui la soluzione è *complessa*, ovvero formata da più variabili ed anche nei casi nei quali la soluzione attesa non è unica, ma appartiene ad un insieme di soluzioni possibili.

Saranno quindi sviluppati dei metodi da aggiungere a quelli di *JUnit* allo scopo di ovviare a questa problematica, rendendo così più semplice l'esecuzione dei test per questi tipi di soluzioni. Mentre con JUnit in questo contesto era possibile testare una variabile alla volta, l'utilizzo dei nuovi metodi permetterà di eseguire il test su una soluzione completa di tutte le proprie variabili ed anche su insiemi di soluzioni di questo tipo.

Questo lavoro di tesi è strutturato nel modo seguente:

Nel Capitolo 2 sarà fatta un'introduzione alla fase di testing, nella quale saranno descritte le differenti tipologie di test e le tecniche utilizzate per la loro realizzazione.

Il Capitolo 3 focalizza l'attenzione sull'analisi dei vari domini di input per le variabili di *JSetL* e la conseguente generazione dei test case per l'*equivalence partitioning test*.

Nel Capitolo 4 sarà affrontato il problema delle *soluzioni complesse singole* ovvero formate da più variabili, ma con un singolo valore. Sarà costruito un modello di codifica per soluzioni di questo tipo e saranno descritti i metodi e le classi che sono state create per la loro gestione finalizzata al testing.

Nel capitolo 5 sarà invece affrontato il problema delle *soluzioni complesse multiple* ovvero insiemi di soluzioni ciascuna costituita da più variabili, evidenziando anche in questo caso il tipo di codifica generato, le classi ed i metodi sviluppati per il testing di questo tipo di soluzioni.

Capitolo 2

I casi di test

La generazione di un buon set di **Casi Di Test** sta alla base del conseguimento di risultati costruttivi nella realizzazione di un prodotto software il più possibile esente da errori.

In questo capitolo si cercherà di descrivere in termini generali come poter creare un buon insieme di casi di test e, nel nostro caso particolare, come questi sono stati generati per testare la libreria **JSetL** [6, 7].

2.1 Il testing

Il testing è una delle fasi del ciclo di vita del software atte a individuarne le carenze di *correttezza, completezza e affidabilità* delle componenti software in corso di sviluppo. Le finalità del test, come già accennato sono due:

1. **Validazione:** Conferma (tramite esame e fornitura di evidenza oggettiva) che i requisiti particolari per uno specifico utilizzo sono stati soddisfatti. In progettazione e sviluppo, la validazione è il processo che esamina un prodotto per determinarne la conformità con le esigenze utente.

2. **Verifica:** Conferma (tramite esame e fornitura di evidenza oggettiva) che i requisiti specificati sono stati soddisfatti. In progettazione e sviluppo, la verifica è il processo che esamina il risultato di una attività per determinarne la conformità con i requisiti formulati per l'attività stessa.

2.1.1 Livelli di test

Un test può essere eseguito a differenti livelli nel processo di sviluppo del software, a seconda dello scopo con cui è stato concepito. Pertanto possiamo suddividere i test in:

- **Test di Unità**
- **Test di Integrazione**
- **Test di Sistema**
- **Test di Accettazione**
- **Test di Regressione**

Test di Unità

I test eseguiti in questa fase mirano alla correttezza degli algoritmi. Il testing di unità verifica il funzionamento isolato delle parti software che sono testabili separatamente. In funzione del contesto, le parti testate possono essere singoli moduli oppure un componente di dimensioni maggiori, costituito da unità strettamente correlate tra loro.

Tipicamente, il testing di unità viene effettuato con accesso al codice testato e con il supporto di strumenti per il debugging. Il testing a livello unitario è focalizzato su unità isolate del prodotto (moduli, classi, funzioni).

Test di Integrazione

I test eseguiti in questa fase mirano alla correttezza delle interfacce.

Il testing di integrazione è il processo di verifica dell'interazione tra componenti software. Le strategie classiche di test di integrazione, top-down o bottom-up, vengono utilizzate con il software tradizionale, strutturato in modo gerarchico. Le strategie moderne di integrazione sistematica sono invece guidate dall'architettura, cioè i componenti e sottosistemi software vengono integrati sulla base delle funzionalità identificate.

Il testing a livello di integrazione è un'attività continuativa. Tranne i casi di software molto piccoli e semplici, le strategie di test di integrazione sistematiche ed incrementali sono da preferire rispetto alla strategia di mettere tutti i componenti insieme nello stesso momento, in quello che viene chiamato *big bang testing*.

I test di integrazione verificano il corretto funzionamento di due (o più) moduli di un software.

Test di Sistema

I test eseguiti in questa fase mirano a testare l'affidabilità, la sicurezza e le prestazioni del prodotto software.

Il testing a livello di sistema si preoccupa del comportamento di un sistema nel suo complesso. La maggior parte degli errori dovrebbe essere già stata identificata durante il testing unitario e di integrazione.

Il test di sistema viene di solito considerato appropriato per verificare il sistema anche rispetto ai requisiti non funzionali, come quelli di sicurezza, velocità, accuratezza ed affidabilità. A questo livello vengono anche testate le interfacce esterne nei confronti di altre applicazioni, componenti standard, dispositivi hardware e ambiente operativo.

Test di Accettazione

Questo tipo di test viene imposto dal cliente per verificare che il programma si comporti nel modo stabilito.

Il testing di accettazione controlla il comportamento del sistema rispetto ai requisiti (comunque espressi) del committente. I clienti effettuano, o specificano, attività tipiche di uso del sistema per controllare che i loro requisiti siano stati soddisfatti.

Questa attività di test può coinvolgere o meno gli sviluppatori del sistema. Il testing è condotto in un ambiente operativo reale, per determinare se un sistema soddisfa i propri criteri di accettazione (ad esempio i requisiti iniziali, e le attuali esigenze dei suoi utenti) e per permettere al cliente di determinare se accettare o meno il sistema.

Test di Regressione

Questa tipologia di test verifica che non siano stati introdotti errori in versioni del software successive.

Nello sviluppo di un prodotto software capita spesso che l'introduzione di una nuove funzionalità in un vecchio prodotto comprometta la qualità di funzionalità preesistenti del prodotto stesso. Pertanto, per assicurare la qualità del prodotto finale, bisognerebbe ripetere l'intera fase di test ad ogni modifica. Il nome di questo tipo di test deriva proprio dal fatto che lo scopo è quello di verificare se la qualità sia regredita.

In questo lavoro di tesi l'oggetto da testare è una libreria open-source (*JSetL*). Questo implica che il suo sviluppo e la sua progettazione non seguono in modo puntuale le fasi dello sviluppo software.

Per il testing di questa libreria è in corso d'opera il test a livello unitario. D'altra parte gli altri tipi di tests visti precedentemente non sarebbero attuabili per un prodotto di questo tipo.

Per ciò che concerne il *test di sistema*, questo andrebbe a testare delle caratteristiche che questo prodotto non implementa. Il *test di accettazione* non è necessario essendo richiesto dal committente (che in questo caso non c'è). Anche il *test di regressione* è inutile, in quanto non esistono (ancora) versioni di *JSetL* successive a quella attuale. L'unico a poter essere eseguito, oltre a quello unitario, è il *test di integrazione*, ma solo dopo che il primo è giunto a termine.

E' comunque giusto sottolineare anche che qualora si giungesse a questo livello, un testing di integrazione nel senso stretto della definizione non sarebbe possibile, in quanto a livello teorico un test di questo tipo dovrebbe essere applicato contemporaneamente a tutte le librerie dell'ambiente di sviluppo Java. In realtà, si può pensare di eseguire un test di integrazione su tutti i moduli (classi e metodi) della libreria *JSetL*, "integrando" in un'unica classe tutti i singoli test che saranno eseguiti sequenzialmente (cosa che è già stata in parte implementata con la creazione della classe `AllTest()`).

2.2 Testing dei linguaggi Object-Oriented

In questo lavoro di tesi sarà utilizzato Java, appartenente alla famiglia dei linguaggi *object-oriented*. Questo tipo di linguaggi deve essere trattato in maniera leggermente diversa dal normale.

Le caratteristiche che possiedono (come astrazioni sui dati, ereditarietà, genericità, $\cdots$) infatti, hanno un impatto diretto sul modo in cui procedere in questa fase [4].

Si può osservare che i livelli di test visti in precedenza mal si adattano al caso di questo tipo di linguaggi. Mentre prima col termine di unità veniva fatto riferimento ad un singolo programma, ad una sua funzione oppure ad una procedura, nel caso dei linguaggi *object-oriented* è necessario innanzitutto individuare da cosa è rappresentata l'unità da testare, se una classe, oppure un metodo di una classe. Testare singolarmente tutti i metodi di ciascuna classe appartenente ad un programma sarebbe troppo costoso. Per questo motivo, i metodi vengono testati all'interno del contesto della classe di appartenenza che pertanto rappresenterà l'unità di testing per i linguaggi *object-oriented*.

Tenendo presente quanto detto, è comunque possibile cercare di far aderire i concetti della suddivisione in livelli di testing, anche nel caso di questo tipo di linguaggi.

Dal fatto che il funzionamento di un programma si basa sull'iterazione fra classi, le quali svolgono le loro funzioni attraverso i propri metodi, un possibile testing su più livelli potrebbe essere il seguente:

- **Basic Unit Testing**: test di una singola operazione di una classe.
- **Unit Testing**: test di una classe nella sua globalità.
- **Integration Testing**: test delle interazioni tra più classi.

Questo modello tuttavia non riuscirebbe a tener conto di tutte le caratteristiche proprie di questi linguaggi. I principali metodi attualmente utilizzati sono tre:

1. **Fault-Based testing**: questo tipo di testing si basa sui possibili difetti presenti nel codice. Naturalmente richiede una classificazione dei difetti ricorrenti che spesso si ottiene a partire dai bug report di applicazioni esistenti.
2. **Scenario-Based testing**: è un tipo di testing nel quale si pone l'attenzione sui possibili errori dovuti all'azione dell'utente, piuttosto che su come si comporta il programma. Richiede una classificazione dei casi d'uso dell'utente, che in seguito saranno utilizzati per la generazione dei casi di test.
3. **Testing della classe**: questo tipo di testing viene eseguito su ogni classe di un programma. La completa copertura di una classe richiede di:

   - Testare **tutte le operazioni** che implementa.
   - Settare ed interrogare **tutti gli attributi** che possiede.
   - Esercitare l'oggetto in **tutti i possibili stati**.

   E' da tenere in considerazione anche il fatto che l'ereditarietà rende più difficile l'individuazione dei casi di test, in quanto le informazioni da testare non sono localizzate, ma distribuite nella gerarchia di ereditarietà. Questa proprietà inoltre, impone di dover testare singolarmente anche tutte le classi derivate [9].

Per l'individuazione dei casi di test in *JSetL*, è stata seguita la strategia del *testing della classe*. Pertanto l'unità che viene testata è la *classe*, o più precisamente un'istanza di essa (un "oggetto"). Il concetto di classe è da intendersi come *stato* più *operazioni*, dove lo stato è rappresentato dagli attributi della classe e le operazioni dai suoi metodi.

In questo ambito quindi, è necessario testare se i metodi della classe interagiscano correttamente tra di loro, attraverso il settaggio e l'interrogazione dei suoi attributi, ovvero testando i cambiamenti di stato. Gli attributi vengono testati uno alla volta rispetto all'insieme degli stati che un oggetto può assumere: se uno (o più) dei metodi non cambia lo stato dell'oggetto secondo quello atteso, allora il metodo è dichiarato contenente errori.

2.3 Generazione dei casi di test

Per poter generare dei casi di test funzionali, occorre poter stabilire dei criteri atti a valutarne l'adeguatezza.

Lo studio dei criteri per la valutazione dell'adeguatezza di un insieme di casi di test nasce a metà degli anni '70 per opera di Goodenough and Gerhart[12].

Da un punto di vista teorico, questi criteri possono essere considerati come delle regole che determinano quali delle istruzioni di un modulo software debbano essere provate da un insieme di casi test che sia completo. Con questo si intende che il successo di questi test implica l'assenza di errori all'interno del modulo preso in esame. In ogni caso, da questa definizione si possono dedurre le proprietà a cui dovrebbe essere conforme un tale criterio:

- **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 quello di fornire una metrica per la valutazione della qualità di una suite di casi di test. In tal caso, dato un programma P , le

sue specifiche S ed un insieme di casi di test T , un criterio di adeguatezza è definito come una funzione **C**: $P \times S \times T \rightarrow [0,1]$. Più il valore è prossimo a 1, più l'insieme di casi di test risulta essere adeguato rispetto al criterio **C**.

Nel corso degli anni sono stati proposti moltissimi criteri di adeguatezza. Una fra le suddivisione più utilizzate, è quella tra criteri **White Box** e **Black Box**. Dei primi fanno parte tutti i casi di test progettati a partire dall'implementazione interna del modulo, mentre degli altri fanno parte quelli progettati a partire dalla specifica del programma.

2.4 Criteri di Adeguatezza per i Casi di Test

2.4.1 Black Box testing

Questo tipo di testing viene detto anche *funzionale* e, come già accennato, la progettazione e generazione dei casi di test è fondata sull'analisi degli output generati dai componenti di un sistema software, in risposta a specifici input definiti sulla base della sola conoscenza dei requisiti specificati per tali componenti.

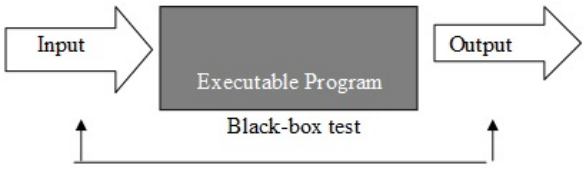


Figura 2.1: Black Box Testing. Un black-box test tiene in considerazione solo l'input e l'output del modulo software, non preoccupandosi dell'implementazione inderna del codice.

Pertanto la chiave è selezionare un sottoinsieme di test (ciascuno costituito da un insieme di dati di ingresso) tale che, se la loro esecuzione sul programma oggetto del testing dà esito positivo, allora il programma svolgerà adeguatamente le sue funzionalità per qualsiasi insieme di dati di ingresso [1] [3].

- **Tests of Requirements:** Il testing basato sui requisiti è una tecnica di convalida in cui vengono progettati vari test per ogni requisito.

Il principio della *verificabilità* dei requisiti afferma che i requisiti dovrebbero essere testabili, cioè scritti in modo tale da poter progettare test che dimostrino che il requisito è stato soddisfatto.

Per iniziare, bisogna assicurarsi che ogni singolo requisito sia stato testato almeno una volta. Come risultato, è possibile ricondurre ciascun requisito al caso di test ad esso associato e viceversa. Il primo caso di test che dovrebbe essere progettato per ciascuno dei requisiti riguarda il *success path* più utilizzato per quel requisito. Col termine di *success path* si intende l'esecuzione di una qualche funzionalità desiderabile, senza nessun tipo di condizione di errore. A questo punto si procede progettando più casi di test di questo tipo, basati su tutti i possibili modi con cui l'utente intende utilizzare questa funzionalità. Inoltre è necessario progettarne anche alcuni che testino i *failure path*. Intuitivamente, i *failure path* dovranno contenere al loro interno qualche tipo d'errore, come ad esempio errori dovuti all'inserimento di input errati. In questi casi bisogna essere certi che il modulo si comporti in modo predicibile e comprensibile. Infine è necessario pianificare il test in modo da eseguire prima i requisiti più problematici.

- **Equivalence Partitioning:** Questa strategia si fonda sul fatto che sarebbe inutile progettare più casi di test che testino lo stesso aspetto di un modulo software. Un buon caso di test pertanto deve individuare una classe di errori che non sia già stata coperta dai casi di test precedenti.

  Il partizionamento degli input in classi di equivalenza è una strategia che può essere usata per ridurre il numero di casi di test da implementare, dividendo il dominio degli input del modulo software in classi di equivalenza.

  Per ciascuna di queste classi, l'insieme dei dati di input dovrebbe essere trattato allo stesso modo dal modulo testato e produrre gli stessi risultati. I casi di test pertanto dovrebbero essere progettati in maniera tale che i vari input ricadano all'interno delle classi di equivalenza. Per ciascuna classe di equivalenza, i casi di test possono essere progettati secondo le seguenti linee guida:

  1. Se le condizioni di input specificano un range di valori, creare una classe di equivalenza valida ed una non valida;
  2. Se le condizioni di input richiedono un determinato valore, creare una classe di equivalenza per il valore valido ed una per quello non

valido. In questo caso è necessario testare individualmente tutti i valori validi e diversi valori non validi;

3. Se le condizioni di input specificano un membro di un insieme, creare una classe di equivalenza valida ed una non valida;
4. Se una condizione di input è un valore booleano, creare una classe di equivalenza valida ed una non valida.

- **Boundary Value Analysis:** Questo tipo di strategia analitica di solito ha una funzione duale con la precedente. Come detto da Boris Beizer, famoso autore di libri di testing, *"I bugs si nascondono negli angoli e si raccolgono nelle zone di confine"*[13].
  Ciò è dovuto al fatto che i programmatori spesso commettono errori sui valori di confine tra le classi di equivalenza del dominio di input. E' giusto pertanto porre la nostra attenzione su questi valori.

  Questa strategia di testing, propone la creazione di casi di test alle *"estremità"* delle classi di equivalenza. Un valore di confine, viene definito come un input che corrisponde ad un valore di minimo o di massimo, specifico per il modulo software testato.

  Nella creazione di un test di questo tipo, si procede nel modo seguente:

  1. Se le condizioni di input specificano un range da **a** a **b** (ad esempio a = 100 e b = 300), creare i casi di test in questo modo:
     - Creare un caso di test per il valore al di sotto del limite inferiore **a** (per il valore 99);
     - Creare un caso di test per il limite inferiore **a** (per il valore 100);
     - Creare un caso di test per il valore immediatamente al di sopra del limite inferiore **a** (per il valore 101);
     - Creare un caso di test per il valore immediatamente al di sotto del limite superiore **b** (per il valore 299);
     - Creare un caso di test per il limite superiore **b** (per il valore 300);
     - Creare un caso di test per il valore immediatamente al di sopra del limite superiore **b** (per il valore 301).
  2. Se le condizioni di input specificano un insieme di valori "speciali", testare questi valori.

- **Decision Table Testing:** Le tabelle di decisione sono utilizzate per la memorizzazione delle regole complesse che devono essere implementate nel modulo software e quindi testate.

  Una semplice tabella decisionale si può osservare nella figura 2.2. Nella tabella le condizioni rappresentano i possibili input. Le azioni sono eventi che dovrebbero essere attivati dipendentemente dalla composizione delle condizioni di input. Ciascuna colonna della tabella rappresenta un'unica combinazione di tali condizioni (e viene chiamata *regola*), che comporta l'attivazione delle azioni associate a tale regola. Ogni regola (o colonna della tabella) può corrispondere ad un caso di test.

	Regola 1	**Regola 2**	**Regola 3**
Condizioni			
A dipende dalle proprietà di B	Si	Si	No
A ha abbastanza denaro per pagare	Si	No	--
Azioni			
A rimane in gioco	Si	No	Si

Figura 2.2: Una semplice tabella delle decisioni: se un giocatore (A) dipende da proprietà appartenenti ad un altro giocatore (B), A deve pagare l'utilizzo di tali proprietà a B e, se non ha abbastanza denaro, A esce dal gioco.

- **Failure ("Dirty") Test Cases:** Per adottare questa strategia, bisogna pensare ad ogni possibile azione che un utente qualsiasi possa intraprendere per guastare il modulo software. Bisogna assicurarsi del fatto che tale modulo sia robusto, ovvero che risponda propriamente nel momento in cui l'utente inserisce input errati.

  Questo tipo di test, infatti viene anche detto *robustness testing*: i casi di test vengono scelti appositamente fuori dal dominio di input, allo scopo di testare la robustezza del modulo in presenza di dati inattesi e input errati. Pertanto seguendo questo tipo di strategia è necessario porsi delle domande inerenti le eventuali azioni intraprese da un utente atte alla generazione di errori, come ad esempio:

  1. Cosa succede se l'utente inserisce dati non validi?
  2. E' possibile che qualche tipo di input generi la divisione per zero?

3. Cosa succede se il modulo fallisce improvvisamente, oppure se i dispositivi di input/output sono scollegati?
4. Il modulo risponde in maniera adeguata a queste condizioni d'errore?

Naturalmente, nel prodotto software in esame non saranno applicati tutti i concetti visti in questo paragrafo. Principalmente, come si vedrà nel prossimo capitolo, sarà seguita la strategia dell'*equivalence partitioning*, con qualche accenno (anche se marginale) al *boundary value analysis* ed al *failure test cases*. La ragione di questa scelta è che le altre strategie mal si adattano a questo tipo di contesto.

2.4.2 White Box testing

Questo tipo di testing viene anche detto *strutturale*, poichè utilizza la struttura interna del programma (o di un suo componente), per ricavare i casi di test.

Tramite il testing *White Box*, si possono formulare criteri di copertura più precisi di quelli formulabili con il testing *Black Box*.

Criteri di copertura

I criteri di copertura del codice sono quelli di gran lunga più utilizzati in pratica. Questo è dovuto principalmente a due fattori:

1. all'esistenza di tools in grado di automatizzare il processo di analisi;
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, etc) 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 sia raggiungibile o meno è noto essere indecidibile e quindi anche il problema di sapere se una copertura del 100% sia raggiungibile o meno, è anch'esso indecidibile.

I criteri di copertura maggiormente utilizzati sono:

1. **Copertura degli Statement:** Il criterio di adeguatezza misura la percentuale di *statements* del modulo che sono state eseguite dalla test suite in esame. E' certamente il criterio più semplice ed anche il più facile da soddisfare e presenta le seguenti caratteristiche:

   - Richiede che ogni istruzione venga eseguita almeno una volta durante il testing;
   - Si basa sull'idea che un difetto in una istruzione non possa essere rilevato senza eseguire l'istruzione stessa;
   - E' un criterio di copertura debole, che non assicura la copertura sia del ramo true che false di una decisione.

2. **Copertura delle Decisioni:** E' simile al precedente, tuttavia al posto di misurare la copertura delle istruzioni, misura la percentuale di copertura dei cammini uscenti dalle istruzioni. In pratica, per ciascuna istruzione condizionale, cerchiamo di coprire entrambi gli archi uscenti:

   - Richiede che ciascuna alternativa di un'istruzione condizionale sia eseguita almeno una volta. In questo caso ogni decisione è stata sia vera che falsa in almeno un caso di test.

3. **Copertura dei Cammini:** Consiste nel determinare quanti, fra tutti i possibili cammini nel grafo, sono coperti da un certo insieme di test. Con *cammino* si intende un'esecuzione del modulo dal nodo iniziale al nodo finale del grafo.
   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.

   Esiste anche una variante chiamata **Copertura dei cammini indipendenti**, che sfrutta il concetto algebrico di *indipendenza lineare*. In pratica viene costruito un campione di cammini da testare, detto **insieme dei cammini di base**. Un cammino si dice indipendente (rispetto

ad un insieme di cammini), se introduce almeno un nuovo insieme di istruzioni o una nuova condizione.

4. **Copertura delle Decisioni e Condizioni:** Assomiglia al *criterio di copertura delle decisioni*, 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.

In questo lavoro di tesi sarà utilizzato il criterio di *copertura degli statements*. Per questo tipo di testing esistono dei tools appositi che eseguono questa analisi in maniera automatica. Per la stesura dei test è stato utilizzato *Eclipse*[14] (ambiente di sviluppo integrato per applicazioni Java) che mette a disposizione il plug-in **Coverage**. Questa estensione permette di evidenziare la percentuale di linee di codice del progetto che sono state eseguite dai test.

Anche se di semplice utilizzo, il criterio di **coverage** è molto probabilmente la metrica più importante che si può estrarre dall'esecuzione degli Unit Test.

Capitolo 3

Generazione dei casi di test in JSetL

In questo capitolo verranno dapprima introdotti la libreria *JSetL* e lo strumento di testing *JUnit*. In seguito, verrà esaminato il modo in cui i concetti generali del testing visti nel capitolo precedente sono stati sfruttati per il testing di alcune funzionalità di *JSetL*, attraverso l'utilizzo di *JUnit*.

3.1 JSetL

JSetL [5] [7] è una libreria Java, sviluppata presso il Dipartimento di Matematica dell'Università di Parma [6], che combina i paradigmi della programmazione *Object-Oriented* tipici di Java, con le caratteristiche dei linguaggi `CLP` (**C**onstraint **L**ogic **P**rogramming) , come: variabili logiche, collezioni di oggetti (liste e insiemi) possibilmente parzialmente specificate, il non determinismo, l'unificazione e la risoluzione di vincoli .

L'unificazione può coinvolgere variabili logiche, come liste ed insiemi di oggetti. I vincoli sono definiti sulle operazioni di base della teoria degli insiemi, come ad esempio *appartenenza, unione* e *intersezione*, e sulle operazioni sugli interi come *uguaglianza, disuguaglianza* e *confronto*.

Per la risoluzione dei vincoli sugli interi, *JSetL* implementa la tecnica del *dominio finito (FD)*, mentre per la risoluzione dei vincoli sugli insiemi segue sia la tecnica dei *set finiti (FS)* per insiemi di interi completamente specificati, sia la meno efficiente, ma più generale e completa procedura di risoluzione dei vincoli propria del **CLP(*SET*)**, per insiemi generici di elementi di qualsiasi tipo, possibilmente parzialmente specificati.

Il non determinismo inoltre, ottenuto attraverso tecniche di *backtracking* e *choice points*, viene sfruttato sia nei metodi specifici di ricerca delle soluzioni, ovvero i metodi di *labeling*, sia nei vincoli su insiemi generici.

Infine `JSetL` permette di definire nuovi vincoli e di gestirli come quelli built-in.

3.2 JUnit

JUnit è un framework open-source scritto in Java per effettuare il testing in modo semplice e organizzato. E' stato progettato da Erich Gamma (autore, con altri, dei Design Patterns) e Kent Beck (autore di Extreme Programming) [8]. Per il progetto dei casi di test questo framework sfrutta le tecniche delle *annotazioni* e delle *asserzioni*, stabilendo delle regole per il loro corretto utilizzo. Un test viene scritto sulla base di opportune classi del package *junit.framework* e tali regole impongono:

- Che ogni classe da testare sia un'estensione della classe `junit.framework.TestCase()`
- Che ogni metodo di test sia una procedura pubblica (`public void`), il cui nome abbia prefisso il termine `test`.
- Che venga utilizzata la classe `junit.framework.Assert` per eseguire i test sulle singole unità.

Esistono differenti tipi di asserzioni per differenti tipi di test, ad esempio i metodi `Assert.assertTrue(boolean condition)` e `Assert.assertFalse(boolean condition)` verificano rispettivamente il successo oppure il fallimento dei metodi che ricevono in input.

Inoltre in **JUnit** sono state definite delle annotazioni, le quali permettono di annotare ogni singolo metodo che desideriamo faccia parte dei nostri test case e di automatizzare l'inizializzazione di metodi e parametri globali, come la loro chiusura. In seguito, ciascuna di queste verrà valutata a run-time.

I principali tipi di annotazioni sono:

- **@Test**: annota ciascun caso di test. Ogni metodo preceduto da questa annotazione, verrà testato a run-time.
- **@Before**: annota un metodo da eseguire prima dell'esecuzione di un caso di test, normalmente allo scopo di inizializzare eventuali parametri che serviranno in fase di test.

- **@After**: annota un metodo da eseguire dopo l'esecuzione di un caso di test, allo scopo di rilasciare le risorse che erano state attivate tramite `@Before`

Nell'esempio seguente viene mostrata la realizzazione di un caso di test, secondo le specifiche di *JUnit*:

```
@Test
public void testOrTestConstraint1a() throws Failure{
      SolverClass solver = new SolverClass();
      LVar xx = new LVar("xx");
      solver.add(xx.eq(1).orTest(xx.eq(2)));
      assertTrue(solver.check());
      String CS = solver.getConstraint().toString();
      assertTrue("_xx = 1 OR _xx = 2".equals(CS));
}
```

3.3 Black Box: Testing di JSetL

Come accennato nel capitolo precedente, si è scelto di seguire il modello del *testing unitario* (cfr. 2.1.1). All'interno di questo contesto, per il testing della libreria *JSetL* è stata adottata la strategia *Black Box* della suddivisione dell'insieme di input in classi di equivalenza (cfr. 2.3.1 *equivalence partitioning*).

Come primo passo è necessario porre l'attenzione su quali siano i possibili tipi di input.

I tipi di dato che sono definiti in *JSetL* sono:

- **Variabili logiche:** possono essere inizializzate o non inizializzate e possono assumere un qualsiasi tipo di valore, il quale può anche essere determinato dall'applicazione di uno dei vincoli sulle variabili stesse.

- **Liste ed Insiemi:** possono essere parzialmente specificate, dato che possono contenere variabili logiche non inizializzate. La principale differenza tra loro consiste nel fatto che per le prime sono importanti ordine e ripetizione degli elementi, mentre per le seconde ciò non vale.

- **Vincoli:** gli operatori di uguaglianza, disuguaglianza ed insiemistici di base (differenza, unione, intersezione) in *JSetL* sono trattati come vincoli. Questi vengono aggiunti al constraint store e poi risolti attraverso il metodo `solve()` del constraint solver.

A questi bisogna aggiungere i tipi built-in di Java (come interi, stringhe e booleani).

In prima istanza, dal fatto che JSetL sfrutta i concetti della programmazione CLP, si può subito intuire che una prima suddivisione degli input avverrà in base alla definizione dei termini in CLP.

Un *termine* t viene definito ricorsivamente nel seguente modo:

- Una variabile è un *termine*.
- Se $t_1,\cdots,t_n$ sono termini e $f \in F$ (dove F è l'insieme dei *simboli di funzione*), con $ar(f) = \text{n}$, allora $f(t_1,\cdots,t_n)$ è un *termine*.

Questa è una definizione formale enunciata in modo tale da potersi adattare a qualsiasi contesto. Nel caso particolare di *JSetL* abbiamo:

- $\text{n} = 0$: *costanti*.
- $\text{n} > 0$: *termini composti* che in *JSetL* possono appartenere ad una delle seguenti tre categorie:
  1. **Liste**
  2. **Insiemi**
  3. **Constraint**

Nei prossimi paragrafi sarà effettuato un esame dettagliato e sistematico dei domini delle classi di *JSetL* testate.

E' necessario infine, notare che *JSetL* dà la possibilità di creare un'istanza della classe `Constraint` con argomenti di tipo:

- **Ground**: dato un termine t con *vars(t)* denotiamo l'insieme delle variabili che occorrono in t. Se *vars(t)* $= \emptyset$ allora t è detto ground. In altre parole t è ground se non contiene variabili non inizializzate.
- **Non Ground**: in analogia con il significato di ground, un termine è non ground, se contiene variabili non inizializzate.

3.4 Il testing di *JSetL*

3.4.1 Test della classe `Constraint`

JSetL attraverso l'utilizzo dei **constraints** (istanze della classe `Constraint`), permette di specificare condizioni su *variabili logiche* e *collezioni logiche* di elementi. I **constraints** sono gestiti dal *constraint solver* (istanza della classe `SolverClass`), che li memorizza nel *constraint store*, contenente la collezione dei *constraints* attualmente memorizzati. In un secondo momento, a seguito dell'invocazione di uno dei metodi di risoluzione dei vincoli (ad esempio `solve()` e `check()`), il *solver* cercherà una soluzione che soddisfi tutti i vincoli memorizzati all'interno del *constraint store*.

Un `Constraint` in *JSetL*, è un'espressione che può assumere una delle seguenti forme:

1. **Atomica**:
   - Il constraint vuoto, indicato con [ ].
   - x_0`.op(`x_1`)` oppure x_0`.op(`x_1`,`x_2`)`

   dove `op` è il nome del `Constraint` e x_0, x_1 e x_2, sono espressioni il cui tipo dipende da `op`. In particolare, `op` può essere uno dei metodi predefiniti che implementa le operazioni di uguaglianza, disuguaglianza e confronto tra interi, oltre a quelle di base sugli insiemi come ad esempio unione e appartenenza.

2. **Composta**:
   - c_1`.and(`c_2`)`;
   - c_1`.or(`c_2`)`;
   - c_1`.orTest(`c_2`)`;
   - c_1`.impliesTest(`c_2`)`

   dove c_1, c_2 sono oggetti di tipo `Constraint` e `and`, `or`, `orTest`, `impliesTest` rappresentano rispettivamente gli operatori logici di congiunzione, disgiunzione, implicazione tra c_1 e c_2.

3. **Negata**:
   - c_1`.notTest()`

   dove c_1 è un oggetto di tipo `Constraint` e `notTest`, l'operatore logico di negazione.

Per come è stata precedentemente definita la strategia di testing Object-Oriented (cfr. 2.2), è necessario anche evidenziare i metodi ("public") di questa classe.

Questi possono essere suddivisi in tre categorie:

1. **Costruttori**:
   - `Constraint().`
   - `Constraint(Constraint c).`
   - `Constraint(String extName, Object o1).`
   - `Constraint(String extName, Object o1, Object o2).`
   - `Constraint(String extName, Object o1, Object o2, Object o3).`
   - `Constraint(String extName, Object o1, Object o2, Object o3, Object o4).`
2. **Metodi di utilità generale**:
   - `boolean equals(Constraint c).`
   - `boolean equals(Object o).`
   - `boolean isGround().`
   - `Object getArg(int i).`
   - `String getName().`
   - `String toString()).`
   - `String toStringInternals()).`
   - `getAlternative().`
   - `void fail()).`
3. **Meta-Constraints**:
   - `Constraint and(Constraint c).`
   - `Constraint impliesTest(Constraint c).`
   - `Constraint or(Constraint c).`
   - `Constraint orTest(Constraint c).`
   - `Constraint notTest().`

Individuazione delle classi di equivalenza

Dai metodi della classe visti in precedenza è possibile ottenere una suddivisione degli input nel modo seguente:

- **Constraint**: nei Meta-Constraints e nel metodo `equals(Constraint c)`.
- **int**: nel metodo `getArg(int i)`.
- **String**: nei costruttori della classe.
- **Object**: nei costruttori della classe e nel metodo `equals(Object o)`.

E' quindi possibile definire il dominio D degli input della classe `Constraint`, come l'insieme:

$$D = \{\texttt{String, int, Object, Constraint}\}$$

A questo punto è necessario notare che, per quanto riguarda il tipo di input `int`, sono considerati validi solo valori maggiori (strettamente) di zero e minori di un certo k, equivalente al numero degli argomenti del constraint oggetto di invocazione. In questo caso pertanto non si potrà effettuare un *equivalence partitioning* (cfr. 2.3.1), ma eventualmente solo un'analisi dei valori di confine e dei valori non validi (cfr. 2.3.1 *boundary value analysis* e *failure test cases*), testando ad esempio il comportamento del metodo `getArg(int i)`, in seguito al passaggio in input di un numero negativo, oppure dei valori di confine, a seconda che il constraint abbia uno oppure k argomenti.

Anche i valori di tipo *String* si prestano poco ad una suddivisione in classi di equivalenza. Non esistono infatti stringhe particolari che potrebbero indurre una tale suddivisione. L'unica eccezione prevista come possibile input è la stringa vuota. Anche per questo tipo di input pertanto è inutile eseguire un *equvalence partitioning*. L'unica tipologia di testing applicabile in questo caso è quella di *boundary value analysis*, eseguendo il test sulla stringa vuota.

Rimangono pertanto gli input di tipo `Constraint` e quindi l'individuazione delle classi di equivalenza sarà effettuata all'interno di questo contesto.

Per quanto riguarda questo tipo di input, facendo riferimento alla definizione di `Constraint` (cfr. 3.4) è possibile effettuare una prima suddivisione in:

- **infissi:** di questa classe fanno parte i metodi che prendono in input un solo argomento, come ad esempio:

$$y_1\texttt{.op(}y_2\texttt{)}$$

dove analogamente a prima, op è il nome del `Constraint` e y_1 e y_2 sono espressioni il cui tipo dipende da op.

- **Constraint Multipli:** in questo caso si fa riferimento al fatto che in *JSetL* è possibile eseguire una catena di chiamate ai metodi delle classi, come ad esempio accade nell'istruzione:

$$x_1\texttt{.}op_1\texttt{(}x_2\texttt{).and(}x_2\texttt{.}op_2\texttt{(}x_3\texttt{)).or(}x_3\texttt{.}op_3\texttt{(}x_4\texttt{))}$$

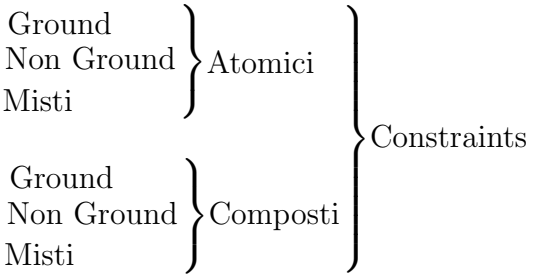
dove x_n $(1 \leq n \leq 4)$ è un'espressione il cui tipo dipende da op, mentre op_n $(1 \leq n \leq 3)$ è il nome del `Constraint`.

Come è stato fatto notare nel paragrafo precedente (cfr. 3.4), la classe `Constraint` possiede dei metodi che permettono di concatenare più vincoli assieme. Questi vengono chiamati **Meta-Constraints** e implementano gli operatori booleani di:

- **Congiunzione**:
  - `Constraint and(Constraint c)`
  che restituisce il vincolo `this` $\wedge$ `c`.
- **Disgiunzione**:
  - `Constraint or(Constraint c)`
  - `Constraint orTest(Constraint c)`
  che restituisce il vincolo `this` $\vee$ `c`.
- **Implicazione**:
  - `Constraint impliesTest(Constraint c)`
  che restituisce il vincolo `this` $\longrightarrow$ `c`.
- **Negazione**:
  - `Constraint notTest()`
  che restituisce il vincolo $\neg$`this`.

Inoltre è possibile distinguere tra constraint *ground* e constraint *non ground* in cui uno o più argomenti siano non ground. Data la presenza di termini appartenenti ad entrambe le categorie, queste tipologie di constraints saranno indicate informalmente con il termine di *constraints misti*.

In analogia con le osservazioni effettuate fino a questo punto, è stato proposto il seguente partizionamento per input di tipo `Constraint`:



E' necessario porre l'attenzione anche al modo in cui un `Constraint` di questo tipo viene creato. Infatti, analizzando i metodi della classe è possibile notare che questi oggetti possono essere creati attraverso i connettori logici implementati nei Meta-Constraints, ma anche in modo equivalente attraverso i costruttori della classe.

In realtà tutti i metodi (come ad esempio `getName()`, `getArg()` $\cdots$), dovrebbero essere testati sia su constraints creati dai costruttori, sia su constraints creati come risultato di altri metodi. Dato che questi due modi di creare un constraint sono equivalenti, per semplificare è sufficiente pertanto testarne solo uno dei due.

3.4.2 Test della soluzione di Meta-Constraints

La risoluzione dei vincoli, in particolare quelli ottenuti dall'utilizzo dei metodi Meta-Constraints, merita un discorso a parte. In questo caso, i metodi che dovranno essere sottoposti al testing non saranno quelli della classe `Constraint`, bensì quelli atti alla gestione dei vincoli definiti all'interno della classe `SolverClass()`, ovvero i metodi `void add(Constraint c)`, `void solve(Constraint c)`, `boolean check(Constraint c)` e `Constraint getConstraint()`. Inoltre il testing di questo tipo di *constraints*, interessa anche tutti i metodi appartenenti alla classe `RwRulesMeta()`, che gestiscono le regole di riscrittura dei Meta-Constraints. Questi ultimi essendo "protected", evidenziano uno dei problemi dovuti al testing su linguaggi *object-oriented* (cfr 2.2), ovvero l'"information hiding". Non si potranno infatti creare dei test esclusivi per questi metodi, ma attraverso il testing dei metodi ("public") della classe `SolverClass`, sarà creata un'istanza di `RwRulesMeta`, che pertanto sarà testata in modo indiretto.

Bisogna sottolineare che per l'*equivalence partitioning testing* (cfr. 2.3.1) di questi metodi, si terrà in considerazione il fatto che la risoluzione di un Meta-Constraint (in generale di un `Constraint`) può fallire. Pertanto, in questo caso verranno distinti i casi `true` di successo (per i quali ad esempio `check() = true`), da quelli `false` che invece inducono un fallimento (per

i quali `check() = false`). Il modello di partizionamento in classi di equivalenza, resta comunque quello realizzato per la classe `Constraint()`, con l'unica aggiunta di questa differenza.

Per ciascuno dei Meta-Constraints `and, or, impliesTest` e `notTest` quindi, sarà prodotto un insieme di casi di test che copra adeguatamente tutte le classi di equivalenza individuate. Sia per il caso di *constraints* singoli che multipli, i tests verranno quindi ripartiti secondo la seguente casistica:

- Constraints ground, con `check() = true`.
- Constraints ground, con `check() = false`.
- Constraints non ground, con `check() = true`.
- Constraints non ground, con `check() = false`.
- Constraints misti, con `check() = true`.
- Constraints misti, con `check() = false`.

Il codice seguente mostra un esempio di test per un Meta-Constraint in caso di successo del metodo `check()`

```
@Test
public void testImpliesConstrant2() throws Failure{
      SolverClass solver = new SolverClass();
      LVar x = new LVar("x");
      LVar y = new LVar("y");
      solver.add(x.neq(1).impliesTest(y.eq(2)));
      solver.add(x.eq(1));
      assertTrue(solver.check());
      assertTrue(x.getValue().equals(1));
}
```

Nell'esempio viene mostrato un test per il Meta-Constraint `impliesTest` che come si può osservare in questo caso, prende in input tre *constraints*: uno di disuguaglianza (`x.neq(1)`) e gli altri due di uguaglianza (`x.eq(2)` e `x.eq(1)`).

Come prima cosa viene creato un oggetto (`solver`) di tipo `SolverClass()`. Questa classe possiede il metodo "protected" `LibConstraintsClass()` del quale crea un'istanza, che a sua volta istanzierà la classe `RwRulesMeta()` (che è un'estensione di `LibConstraintsClass()`), permettendo così il testing (indiretto) dei suoi metodi.

Nell'esempio vengono create due `LVar x` e `y` che sono utilizzate nella costruzione dei tre *constraints*. A questo punto il metodo `boolean assertTrue()` controlla che il constraint risultante venga aggiunto correttamente al `ConstraintStore` in seguito alle due chiamate del metodo `add(Constraint c)` del solver. Infine, viene anche controllata la corretta associazione del valore uno ad `x` attraverso il metodo `Object getValue()`, che però essendo un metodo proprio della classe `LVar()`, esula da questo contesto.

3.4.3 Test della classe `LVar`

Una *variabile logica* è un'istanza della classe `LVar` e si può considerare come un oggetto che non contiene alcun valore modificabile. E' comunque possibile gestire variabili di questo tipo attraverso l'utilizzo di relazioni (i *constraints*) che coinvolgono tali variabili logiche e valori appartenenti ad uno specifico dominio. In particolare, il vincolo di uguaglianza (`eq()`) permette di associare un valore ad una `LVar`. Tale valore è *immutabile*, nel senso che ad esempio non potrà essere modificato da un'istruzione di assegnamento.

Una `LVar` possiede una serie di attributi che la caratterizzano e un insieme di metodi utili per la sua analisi (come ad esempio, la lettura degli attributi e il confronto con altre variabili logiche).

E' importante quindi, analizzare gli attributi di una `LVar`, per poter capire che tipo di variabili possono essere create dai costruttori propri di questa classe.

Una variabile logica possiede tre attributi:

- `String extName`: rappresenta il *nome esterno* della variabile. E' un parametro opzionale, nel senso che può anche non essere impostato (nel qual caso viene assegnato di default il simbolo `?`).
- `Object value`: rappresenta il valore associato alla variabile logica. Se non presente alla variabile sarà associato il valore `null`.
- `boolean init`: flag che indica l'inizializzazione della variabile. Se è settato a `true` significa che la variabile è inizializzata, al contrario sarà false.

E' necessario analizzare anche i metodi della classe che anche in questo caso, è possibile suddividere in tre categorie:

1. **Costruttori**:
    - `LVar(String extName)`.

- `LVar(String extName, Object o)`.
- `LVar(String extName, LVar lv)`.

con `extName` parametro opzionale.

2. **Metodi di utilità generale**:
   - `LVar clone()`.
   - `LVar setName(String extName)`.
   - `boolean equals(LVar lv)`.
   - `boolean isBound()`.
   - `String getName()`.
   - `String toString()`.
   - `void output()`.
   - `Object getValue()`.

   il metodo `equals()` è definito anche per input di tipo `Object`
3. **Vincoli**:
   - `Constraint eq(Object o)`.
   - `Constraint neq(Object o)`.
   - `Constraint in(LSet ls)`.
   - `Constraint nin(LSet ls)`.

   i metodi `in()` e `nin()` sono definiti anche per input di tipo `Set<?>`

Individuazione delle classi di equivalenza

Dai metodi e dai costruttori propri di questa classe, è possibile visualizzare quali siano gli elementi del dominio di input di una `LVar`:

- **String**: nei costruttori della classe e nel metodo `setName(String extName)`.
- **Object**: nei costruttori della classe e nei metodi `equals(Object o)`, `eq(Object o)` e `neq(Object o)`.
- **LVar**: nei costruttori e nel metodo `equals(LVar lv)`.
- **LSet**: nei metodi `in(LSet ls)` e `nin(LSet ls)`.

Si avrà quindi che il dominio D della classe `LVar` risulterà essere:

$$D = \{\texttt{String, Object, LVar, LSet}\}$$

Per quanto riguarda gli elementi di tipo `Object`, è necessario precisare che questo tipo di input racchiude in sé i tipi già elencati nel dominio (ovvero gli oggetti Java e *JSetL*), come anche che le costanti primitive (`int, char, double,` $\cdots$).

Anche in questo contesto comunque vale quanto detto per la classe `Constraint` (cfr. 3.4.1). Pertanto per oggetti Java e costanti primitive sono preferibili strategie di testing come il *boundary value analysis* e il *failure test cases* (cfr. 2.3.1).

L'analisi di *equivalence partitioning* si focalizzerà quindi, sull'individuazione delle classi di equivalenza per oggetti *JSetL* (`LVar` e `LSet`).

In prima istanza si può notare come un oggetto di tipo `LVar` possa essere creato con associato un `extName`, oppure senza alcun nome associato. Questa caratteristica induce una prima suddivisione del dominio in due classi:

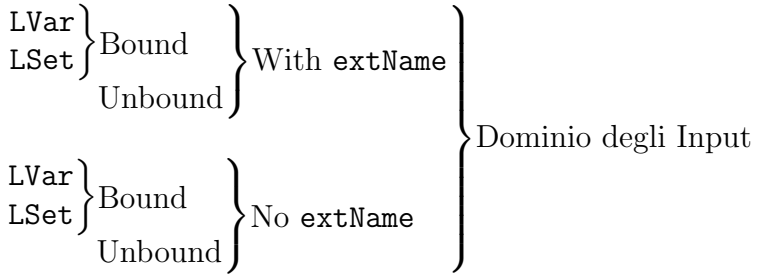
- `LVar` **With** `extName`.
- `LVar` **No** `extName`.

L'analisi procede prendendo in considerazione la caratteristica principale di un `LVar`. Questa, essendo una variabile logica può o meno, aver associato un valore. Da questa sua peculiarità si può trarre un'ulteriore suddivisione degli input in:

- **Bound**: quando il dominio di una `LVar` si riduce ad un unico valore viene detta *bound*, ovvero "legata" a tale valore.
- **Unbound**: al contrario, una `LVar` viene detta *unbound* se non ha alcun valore associato.

Si può notare infine, che nel caso in cui un `LVar` con associato o meno un `extName` sia bound, questa potrà essere "legata" ad un'altra `LVar`, la quale a sua volta potrà essere bound\unbound, oppure ad un `LSet`, che a sua volta potrà essere chiuso\aperto (crf. 3.7).

In conclusione, l'analisi precedente ha portato alla costruzione del seguente partizionamento degli gli input per classe `LVar`:



Verrà adesso mostrato un esempio relativo ad un rappresentante di una delle classi di equivalenza:

```
@Test
public void testGetValInitWithLVarInit(){
      LVar y = new LVar("y",3);
      LVar x = new LVar(y);
      assertEquals(3,(x.getValue()));
}
```

Nell'esempio precedente è stato riportato il test per il caso di una `LVar` con un `extName` e bound. Si può notare infatti che il valore associato alla `LVar` `x`, è sua volta una `LVar` anch'essa bound (in questo caso ad una costante primitiva intera).

Questo test controlla la corretta associazione alla variabile x del valore con il quale era stata inizializzata y (`assertEquals(3,(x.getValue()))`).

3.4.4 Test della classe `LSet`

Un *insieme logico* viene creato attraverso un'istanza della classe `LSet`. Anche questa classe di oggetti viene gestita attraverso l'utilizzo dei *contraints*. In particolare attraverso il metodo di inserimento, è possibile creare insiemi *unbound* parzialmente specificati, ovvero formati da una struttura dati con un certo numero di elementi e da un resto ignoto.

Un oggetto di tipo `LSet` infatti è definito dalla coppia $\langle$*elems, rest*$\rangle$ dove con *elems* si intende un insieme $\{e_1,\cdots,e_n\}$ con n ≥ 0 di elementi di qualsiasi tipo, mentre *rest* può rappresentare l'insieme vuoto oppure un `LSet` che può essere a sua volta *unbound*. In questo caso l'`LSet` si dice *aperto*, altrimenti si dice *chiuso*.

E' da notare inoltre, che la cardinalità di un LSet parzialmente specificato non viene determinata in modo univoco. Si pensi ad esempio all'insieme logico parzialmente specificato $\{1, x\}$. In questo caso la cardinalità dell'`LSet` potrebbe essere sia uno che due, dipendentemente dal fatto che ad x, in un secondo momento, possa essere o meno associato il valore uno.

Un oggetto di tipo `LSet` possiede due attributi:

- `extName`: rappresenta il *nome esterno* della variabile. E' un parametro opzionale, nel senso che può anche non essere impostato (nel qual caso viene assegnato di default il simbolo `?`). Può assumere anche il valore `_emptyLSet` nel caso in cui l'`LSet` sia vuoto.
- `elems` e `rest`: conterranno gli elementi che possono essere inseriti in un `LSet`.
- `init`: flag che indica l'inizializzazione della variabile. Di default assume valore `false`, diventa `true` nel momento in cui l'`LSet` viene inizializzato.

Anche in questo caso l'analisi parte dall'osservazione dei metodi della classe, che possono essere suddivisi in:

1. **Costruttori**:
   - `LSet(String extName).`
   - `LSet(String extName, Set<?> s).`
   - `LSet(String extName, LSet ls).`

   con `extName` parametro opzionale.
2. **Metodi per creare `LSet` bound**:
   - `static LSet empty().`
   - `static LSet mkSet(String extName, int n).`
   - `LSet ins(Object o).`
   - `LSet insAll(Object[] c).`

   il parametro `extName` è opzionale. Il metodo `insAll` è definito anche per input di tipo `Collection`
3. **Metodi di utilità generale**:
   si dividono in due categorie
   (a) **Metodi di base**:
   sono gli stessi già presentati per le `LVar` (cfr. 3.6), riadattati per gli `LSet`.
   (b) **Metodi propri delle collezioni logiche**:

- `Object get(int i)`.
- `LSet getRest()`.
- `int getSize()`.
- `boolean isClosed()`.
- `boolean isEmpty()`.
- `boolean isGroung()`.
- `boolean testConstraint(Object o)`
- `Iterator iterator()`.
- `void printElems(char sep)`.
- `Vector toVector()`.

4. **Vincoli**:
   possono essere

   (a) **di confronto**:

   - `Constraint eq(LSet ls)`.
   - `Constraint neq(LSet ls)`.

   definiti anche per input di tipo `Set<?>`.

   (b) **di appartenenza**:

   - `Constraint contains(LVar lv)`.
   - `Constraint ncontains(LVar lv)`.

   definiti anche per input di tipo `Object`.

   (c) **derivati dalla teoria degli insiemi**:

   - `Constraint diff(LSet ls1, LSet ls2)`.
   - `Constraint disj(LSet ls)`.
   - `Constraint inters(LSet ls1, LSet ls2)`.
   - `Constraint less(LSet ls, LVar lv)`.
   - `Constraint size(Integer i)`.
   - `Constraint subset(LSet ls)`.
   - `Constraint union(LSet ls1, LSet ls2)`

   i metodi `diff()`, `disj()`, `inters()`, `less()` `subset()` e `union()` sono definiti anche per input di tipo `Set<?>`. Il metodo `less()` come secondo argomento può prendere anche un input di tipo `Object`.

Individuazione delle classi di equivalenza

Dai metodi della classe è possibile pertanto individuare gli elementi del dominio di un `LSet`:

- **String**: nei costruttori e nei metodi `mkSet()`, `equals()` e `setName()`.
- **Object**: nei metodi `equals()`, `insAll()`, `less()`, `contains()`, `ncontains()` e `testConstraint()`.
- **LSet**: nei costruttori e nei metodi `equals()`, `eq()`, `neq()`, `diff()`, `disj()`, `inters()`, `less()`, `subSet()`, `union()`
- **LVar**: nei metodi `contains()`, `ncontains()`, `less()` e `forallElems()`.
- **int**: nei metodi `mkSet()` e `get()`.
- **Constraint**: nel metodo `forallElems()`.
- **Collection**: nel metodo `insAll()`
- **Integer**: nel metodo `size()`.
- **char**: nel metodo `printElems()`.

Si avrà quindi, che il dominio D della classe `LSet` risulterà:

$$D = \{\texttt{Object, Integer, Collection, String, LSet, LVar, Constraint, int, char}\}$$

E' necessario fare attenzione al fatto che il tipo `Object` rappresenta un oggetto generico e quindi comprende già tutti gli oggetti Java (`Integer, String, Collection, Set,` $\cdots$), tutti gli oggetti *JSetL* (`LVar, LSet, Constraint,` $\cdots$), come anche le costanti primitive (`int, char,` $\cdots$).

Anche in questo caso la suddivisione in classi di equivalenza non sarà effettuata per tutti i tipi di input. Infatti per quanto riguarda gli oggetti Java, nel caso di input di tipo `Integer`, sono considerati validi solo quelli maggiori strettamente di zero, nel caso di input di tipo `String`, l'unica eccezione riguarda la stringa vuota e vale lo stesso anche per input di tipo `Collection`.

Anche per le costanti primitive si può fare un discorso analogo, pertanto come nel caso della classe `LVar` (cfr. 3.6.1), le metodologie di testing più appropriate per oggetti Java e costanti primitive sono il *boundary value analysis* e il *failure test cases* (cfr. 2.3.1).

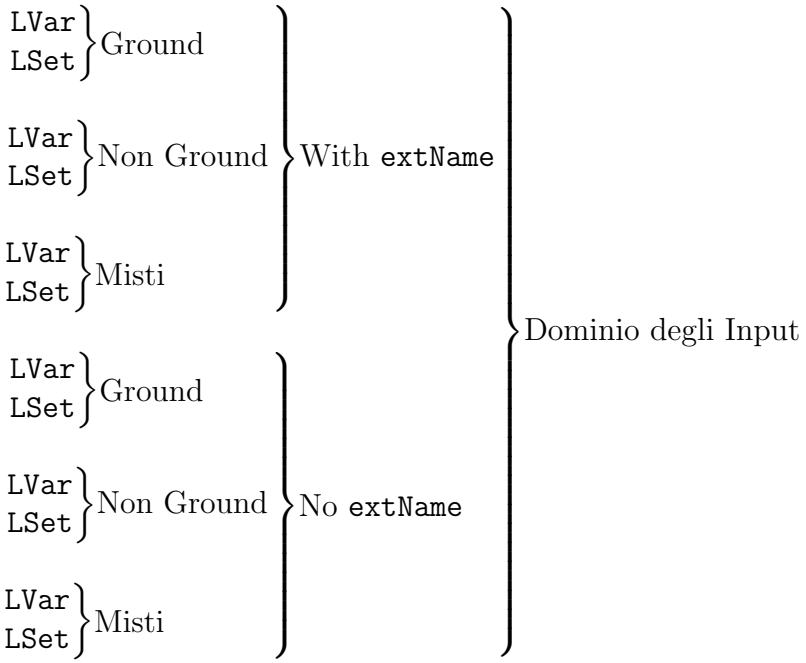
Anche per oggetti ti tipo `LSet`, una prima suddivisione potrà essere effettuata tra:

- `LSet` **With** `extName`.
- `LSet` **No** `extName`.

Dal fatto che un `LSet` può essere chiuso oppure aperto si possono ottenere oggetti di questo tipo che siano *completamente specificati*, *parzialmente specificati*, oppure *completamente non specificati*. Da ciò scaturisce che un'ulteriore suddivisione potrà essere fatta tra **ground** nel caso di `LSet` chiusi, **non ground** nel caso *LSet* aperti e **misti** nel caso di `LSet` parzialmente specificati.

A questo punto è necessario notare che nel caso di `LSet` ground e misti, questi potranno essere associati ad altri oggetti *JSetL*, che a loro volta potranno essere bound/unbound nel caso di `LVar` (cfr. 3.6.1), chiusi/aperti nel caso di `LSet`.

La figura seguente mostra il partizionamento scaturito dall'analisi del dominio degli input di un `LSet`:



Di seguito sarà mostrato un esempio relativo al testing di questa classe:

```
@Test
public void testIsGroundInitWithConstrLSetOpen() {
      LSet r = new LSet("r");
      LSet x1 = r.ins(1).ins(2);
```

```
        LSet x = new LSet(x1);
        x.output();
        assertFalse(x.isGround());
}
```

Questo è un esempio di test relativo al caso di `LSet` *parzialmente specificato* e senza `extName` associato. Si nota che l'`LSet x` oggetto del test, viene costruito invocando il costruttore della classe (`LSet(LSet ls)`), prendendo come input un altro `LSet x1`. Diversamente `x1` è stato creato attraverso la concatenazione di due chiamate al metodo `ins(Object o)` sull'`LSet r`, quest'ultimo *completamente non specificato*. In questo modo `x` sarà costituito da una parte (*specificata*) di elementi, mentre la restante parte resta ignota (quindi l'output di x1 risulterà essere: `? = {2,1|_r}`). Ciò spiega l'utilizzo di `assertFalse(x.isGround())`.

3.5 White Box: Il tool *Coverage*

Nel processo di testing l'*analisi di copertura* è una parte decisamente critica. Per quanto un testing di unità sia un punto di fondamentale importanza in tale processo, rischia di diventare poco utile se non si è a conoscenza di quanta o quale parte di codice benefici realmente dei tests.

È per questo motivo che, oltre a verificare che il prodotto finale rispecchi le specifiche attraverso le tecniche *Black Box* viste in precedenza, una parte essenziale e complementare a questo tipo di testing è data dalle tecniche **White Box**. Queste strategie di testing permettono di eseguire un'analisi di copertura del codice, attraverso la quale è possibile ottenere la percentuale di codice effettivamente sottoposta al test (ovvero la percentuale di *code coverage*). Quest'ultima, si può ottenere controllando l'esecuzione dei tests mediante l'uso di appositi strumenti software.

Anche così facendo, non si ha comunque la certezza che, anche in presenza di una buona percentuale di copertura del codice, il software risponda come previsto. Questo è dovuto ad alcune proprietà dei paradigmi di programmazione che rendono difficile il lavoro di questi programmi, come ad esempio quelli della *programmazione dinamica* (come il polimorfismo o l'uso di variabili dinamiche allocate a tempo di esecuzione).

Il **Code Coverage** definisce pertanto una misura utilizzata per il testing del software, che descrive il grado con cui il codice sorgente è stato testato da parte di una suite di test.

Il criterio di *coverage* utilizzato in questa analisi è quello di *copertura degli statements*. I test sono selezionati pertanto, in base alla loro capacità di

coprire gli statements del codice (ovvero i nodi del grafo, che rappresentano i metodi delle classi).

La percentuale di codice coperto dalla suite di test viene calcolata nel seguente modo:

$$Statement\ Coverage = \frac{numero\ istruzioni\ eseguite}{numero\ totale\ delle\ istruzioni}$$

Per effettuare questo di tipo test è stato utilizzato il pulg-in **Coverage**. Il suo funzionamento si basa sulla marcatura di ogni linea di codice con un flag, che viene settato contestualmente all'esecuzione della relativa linea di codice.

Capitolo 4

Soluzioni complesse singole

Con il termine di "*soluzioni complesse*" si intendono tutte quelle soluzioni che possiedono più di una variabile.

JUnit (cfr. 3.2) è in grado di eseguire un test su soluzioni composte da una singola variabile e quindi la gestione di soluzioni complesse diventa piuttosto laboriosa (si pensi ad esempio al problema delle n-Regine, con n grande).

E' necessario precisare che è possibile identificare due categorie di soluzioni complesse, quelle *singole* ovvero composte da un singolo valore per ogni variabile e quelle *multiple*, ossia composte da più possibili valori per ogni variabile.

Per eseguire test su soluzioni formate da più di una variabile sono state create delle strutture apposite per la memorizzazione e la gestione delle variabili di una soluzione.

In questo capitolo sarà analizzato il modo nel quale sono state gestite le *soluzioni complesse singole*, partendo dalla loro codifica. Sarà inoltre analizzata la strategia utilizzata per l'esecuzione dei test su quei programmi che restituiscono in output soluzioni di questo tipo.

La parte riguardante le *soluzioni complesse multiple*, sarà trattata nel capitolo successivo.

4.1 Rappresentazione delle soluzioni

Una buona codifica delle soluzioni costituisce un buon punto di partenza per la realizzazione di procedure atte all'esecuzione del test.

Tale codifica permette il corretto caricamento delle differenti variabili in apposite strutture, che andranno a contenere pertanto la soluzione che verrà utilizzata per il test.

Si può notare in primo luogo che in *JSetL* è possibile suddividere i valori delle variabili logiche in due categorie principali: **atomici** come `int` o `String`, e **strutturati** come `LList` o `LSet`.

Per rappresentare le soluzioni di ciascuna categoria sono state utilizzate due codifiche differenti. I motivi di questa scelta sono essenzialmente due:

1. Per quanto riguarda i tipi di dato **atomici**, erano già state create procedure per il testing di soluzioni contenenti dati di questo tipo in ambiente *Java JSR-331*, nel lavoro di tesi sviluppato da *Riccardo Zangrandi* [16].

2. Per quanto concerne i tipi di dato **strutturati**, essi rappresentano una categoria di dati differente dalla precedente con delle caratteristiche proprie che impongono una diversa strategia di codifica.

4.1.1 Valori atomici: la classe `SimpSol`

Questa classe ha lo scopo di rappresentare una soluzione le cui variabili sono formate da elementi appartenenti alla categoria dei tipi **atomici** (come `int` e `String`).

La classe è provvista di una serie di metodi `public` che permettono la manipolazione e la gestione degli elementi costituenti la soluzione.
Questi metodi sono:

`int getNumberOfElements()`
: Restituisce il numero di variabili attualmente contenute nella soluzione.

`Vector<String> getElements()`
: Restituisce i valori delle variabili della soluzione.

`void setElementName(int index, String name)`
: Assegna il nome contenuto in `name` alla variabile di indice `index` della soluzione.

`void setElement(int index, String element)`
: Inserisce il valore `element` nella posizione indicata da `index` all'interno del vettore contenente la soluzione.

`void addSolution(String[] names, String[] elements)`
: Inserisce un'intera soluzione. I nomi ed i valori delle variabili sono passati come parametri e sono contenuti rispettivamente in `names` e in `elements`.

`String getElement(int i)`
: Restituisce il valore dell'i-esima variabile.

`Vector<String> getElementsName()`
Restituisce i nomi delle variabili attualmente memorizzate.

`String getElementName(int i)`
Restituisce il nome dell'i-esima variabile.

`String getElementWithName(String name)`
Restituisce il valore della variabile di nome `name`.

Di seguito è possibile osservare un esempio di utilizzo di alcuni dei metodi della classe applicati a due oggetti di classe `simpSol` di nome `found` ed `expected`.

1.
```
String[] vars = {"34", "65", "12"};
String names[] = new String[vars.size()];
String elements[] = new String[vars.size()];
SimpSol foundSolution = new SimpSol();
for(int i = 0; i < vars.size(); i++){
          names[i] = "x-" + i;
          elements[i] = vars.get(i);
}
foundSolution.addSolution(names, elements);
```

2.
```
for (int i = 0; i < expected.getNumberOfElements(); i++){
     if (found.getElementsName().contains(
          expected.getElementName(i))){
          if (!expected.getElement(i).toString().
               equals(found.getElementWithName(expected.
                    getElementName(i)))){
                    ok = false;
          }
     }
}
```

Nella prima parte dell'esempio è possibile notare la creazione dell'oggetto `foundSolution` di tipo `SimpSol`, che sarà utilizzato per rappresentare la soluzione attuale `vars` attraverso il metodo `addSolution(String[ ] names, String[ ] elements)`.

Nella seconda parte invece è stato riportato il ciclo `for()` del metodo `assertEqSolOpz(SimpSol found, SimpSol expected)` della classe `AssertionSol` (cfr. 4.3) che esegue il confronto tra le soluzioni. Per l'esecuzione del test il metodo `assertEqSolOpz(SimpSol found, SimpSol expected)`, esegue le chiamate ai metodi `getElementsName()` per ottenere il vettore

contenente i nomi delle variabili, `expected.getNumberOfElements()`, utilizzato dentro il ciclo per scorrere tutte le variabili presenti nel vettore soluzione, `getElementName(int i)` per ottenere il nome dell'i-esima variabile della soluzione attesa ed infine `getElement(int i)` che restituisce il valore dell'iesima variabile della soluzione attesa

Dal punto di vista dell'implementazione, la classe è costituita da tre campi `private`:

1. `numberOfElements`: specifica il numero di variabili contenute nella soluzione.
2. `elementsName`: `Vector` contenente i nomi delle variabili.
3. `elements`: `Vector` contenente i valori delle variabili.

4.1.2 Valori strutturati: la classe `CompSol`

Lo scopo di questa classe è quello di rappresentare soluzioni le cui variabili siano composte da elementi di tipo *strutturato*, attraverso le apposite strutture presenti al suo interno.

Possiede un insieme di metodi `public` che permettono la sua manipolazione in maniera opportuna.
Tali metodi sono:

`int getNumberOfElements()`
: Restituisce il numero di variabili attualmente contenute nella soluzione.

`void clearElems()`
: Rimuove tutte le variabili di una soluzione.

`LCollection getElement(int i)`
: Restituisce il valore dell'i-esima variabile.

`String getElementName(int i)`
: Restituisce il nome dell'i-esima variabile.

`LCollection getElementWithName(String name)`
: Restituisce il valore della variabile di nome `name`.

`Vector<String> getElementsName()`
: Restituisce i nomi delle variabili attualmente memorizzate.

`Vector<LCollection> getElements()`
: Restituisce i nomi delle variabili della soluzione.

`void addSolution(Vector<LCollection> varFound)`
Inserisce la soluzione contenuta in `varFound`.

`void setElement(Vector<LCollection> varFound)`
Inserisce il valore `varFound` alla soluzione.

`void setElement(LCollection varFound)`
Inserisce il valore `varFound` alla soluzione.

Sotto è riportato un semplice caso di utilizzo di alcuni dei metodi descritti.

1.
```
LVar a = new LVar('a');
LVar b = new LVar('b');
LVar c = new LVar('c');
LList lst = LList.empty().ins(d).ins(c);
LList sol = LList.empty().ins(lst).ins(b).ins(a);
CompSol foundSolution = new CompSol();
foundSolution.setElement(sol);
```

2.
```
for (int i = 0; i < expected.getNumberOfElements(); i++){
    if (found.getElementsName().contains(expected.
        getElementName(i))){
        if(!expected.getElement(i).toString().
            equals(found.getElementWithName(expected.
                getElementName(i)).toString())){
                ok = false;
        }
    }
}
```

Analogamente all'esempio visto in precedenza riguardante le variabili **atomiche** (cfr. 4.1.1), anche in questo caso nella prima parte è possibile osservare la creazione dell'oggetto `foundSolution` di tipo `CompSol`, che sarà utilizzata per rappresentare la soluzione attuale `sol` attraverso il metodo `setElement(LCollection sol)` della classe `CompSol`.

Nella seconda parte invece è stata riportata la porzione di codice del ciclo `for()` appartenente al metodo `assertEqSolOpz(CompSol found, CompSol expected)`, che adempie al compito di eseguire il confronto tra soluzioni.

Nell'inizializzazione del ciclo la `CompSol expected` (contenente la soluzione attesa) effettua una chiamata al metodo `getNumberOfElements()` in modo da iterare su tutte le proprie variabili. In seguito viene controllato che il vettore dei nomi della `CompSol found` (contenente la soluzione attuale) ottenuto effettuando la chiamata al metodo `getElementsName()`, contenga ciascuno dei nomi delle variabili contenute in `expected` (`expected.getElement`

`Name(i)`). Infine viene controllata l'uguaglianza tra il valore della soluzione attesa (`expected.getElement(i)`) e quello della soluzione attuale ottenuto tramite la chiamata al metodo `getElementWithName(expected.getElement Name))`.

Per ciò che riguarda l'implementazione, questa struttura dati è composta da un campo `private` costituito da un `Vector` di nome *elements*, nel quale saranno memorizzati i valori delle variabili della soluzione.

4.1.3 Codifica di soluzioni singole

Come accennato all'inizio del capitolo, sono state adottate due codifiche distinte a seconda del tipo di dato trattato.

Lo scopo della codifica delle soluzioni è quello riuscire ad esprimere le soluzioni attraverso una stringa, in modo tale che queste possano essere lette e rappresentate in modo corretto dalle apposite strutture create per questo scopo (cfr. 4.1).

Per quanto riguarda soluzioni formate da tipi di dato atomici, ogni variabile deve possedere un nome della forma `x-n`, con n $\in \mathbb{N}$ crescente, ed essere separata dalla successiva con un singolo spazio vuoto.

Un esempio di quanto detto può essere osservato di seguito:

x-0 9567 x-1 1085 x-2 10652

L'esempio precedente mostra una soluzione che prevede tre variabili. Si può notare come la soluzione sia compresa tra i due delimitatori (*Nome* e *!!!*) e che è della forma ⟨nomeVar, Valore⟩, come precisato in precedenza.

Per le soluzioni le cui variabili sono formate da tipi di dato **strutturato**, la codifica cambia leggermente.

Infatti non viene più richiesto di esplicitare il nome delle singole variabili, in quanto il suo inserimento avviene in automatico al momento del caricamento della variabile stessa, sfruttando il metodo `setName()` di *JSetL*.

Per separare le differenti variabili le une dalle altre, è stato introdotto il simbolo "`$`".

Di seguito si può osservare un esempio della codifica di una soluzione contenente variabili di questo tipo:

[26 [a c b e d]] $

Questo esempio mostra la codifica di una soluzione contenente un'unica variabile con valore di tipo **strutturato**, precisamente la lista [[26[a,c,b,e,d].

Allo scopo di semplificare la lettura delle soluzioni e rendere il codice meno pesante (si pensi ad esempio all'inserimento di una soluzione composta da un numero molto elevato di variabili), oltre che effettuare il passaggio di queste direttamente come parametro di funzione, è stata prevista anche la possibilità di memorizzare tali soluzioni su file.

In questo modo è possibile sia avere un file di soluzioni per ciascun programma, che avere un unico file per tutti i programmi sui quali si sta eseguendo il test. Per questo motivo, è stato necessario poter distinguere le soluzioni appartenenti ad un programma rispetto a quelle appartenenti ad un altro.

In particolare ogni soluzione deve essere salvata sul file all'interno di due delimitatori, il primo dei quali indica il nome del problema di riferimento, mentre il secondo è costituito da tre punti esclamativi per indicare la fine della soluzione stessa. In questo modo i metodi di lettura delle soluzioni (cfr. 4.2.2), riusciranno a distinguere le soluzioni appartenenti al programma oggetto del test, rispetto a quelle appartenenti agli altri programmi.

4.2 Costruzione delle soluzioni

In questo capitolo saranno analizzati i metodi e le strutture create per la cattura delle soluzioni attuali ottenute in output dai vari programmi utilizzati nel test e per la lettura ed eventuale ricostruzione delle soluzioni attese distinguendo tra variabili con valore di tipo **atomico** oppure `strutturato`.

4.2.1 Soluzioni calcolate: la classe `PrepareSol`

Questa classe è stata creata allo scopo di gestire le soluzioni ottenute dall'esecuzione dei programmi che si vogliono testare. Pertanto al suo interno sono stati implementati metodi per la gestione di tutti i tipi di dato trattati.

Relativamente al caso di soluzioni composte da variabili con elementi di tipo **atomico** si distinguono due metodi `public`:

`Static SimpSol prepareSimpSol(String vars)`
: Metodo che viene utilizzato nel caso in cui la soluzione sia composta da una singola variabile.
Prende in input una stringa `vars` contenente il valore della variabile e restituisce un oggetto di tipo `SimpSol` contenente la soluzione finale.

`Static SimpSol prepareSimpSol(Vector<String> vars)`
: Metodo che viene utilizzato nel caso in cui la soluzione sia composta da più variabili.
In questo caso i valori sono passati in input tramite il `Vector vars` e poi memorizzati e restituiti all'interno di un oggetto di tipo `SimpSol`.

Per ciò che riguarda variabili con elementi di tipo **strutturato**, sono stati implementati i seguenti due metodi `public`:

`Static CompSol prepareCompSol(LCollection sol)`
: Questo metodo viene utilizzato per la gestione di soluzioni formate da una singola variabile di tipo `LList`.
Restituisce un oggetto di tipo `CompSol`, con la soluzione memorizzata al suo interno.

`Static CompSol prepareCompSol(Vector<LCollection> sol)`
: Questo metodo viene utilizzato per gestire soluzioni composte da più variabili.
In input prende un `Vector` contenente le differenti variabili e restituisce un oggetto di tipo `CompSol`, contenente la soluzione composta da tutte le sue variabili.

Ad esempio avendo una soluzione calcolata del tipo:

```
LList l1 = LList.empty().ins('d').ins('e').ins('b').
      ins('c').ins('a');
LList l = LList.empty().ins('l1').ins('26');
```

che in particolare rappresenta la soluzione [26,[a,c,b,e,d]], è possibile memorizzarla in un oggetto di tipo `CompSol` in questo modo:

```
CompSol c = new CompSol();
c = PrepareSol.prepareCompSol(l);
```

4.2.2 Soluzioni attese: la classe `ReadFrom`

La classe `ReadFrom` permette la lettura ed il caricamento della soluzione attesa, precedentemente codificata (cfr. 4.1.3), memorizzata all'interno di un file oppure direttamente come parametro.

Per quanto riguarda soluzioni costituite da variabili di tipo **atomico** le funzioni (`public`) che le gestiscono sono le seguenti:

`SimpSol readSolution(String name, String path)`
: Apre il file nel percorso definito in `path`, trova le soluzioni del problema di nome `name` e le carica in un oggetto di tipo `SimpSol`.

`SimpSol readSolution(String sol)`
Prende in input la stringa `sol` e restituisce la soluzione caricata in un oggetto di tipo `CompSol`.

In modo analogo le seguenti funzioni (`public`) gestiscono soluzioni con variabili di tipo **strutturato**:

`CompSol readCompSol(String name, String path)`
Apre il file nel percorso definito in `path`, trova la soluzione del problema di nome `name` e le carica in un oggetto di tipo `CompSol`.

`CompSol readCompSol(String sol)`
Prende in input la stringa `sol` e restituisce la soluzione caricata in un oggetto di tipo `CompSol`.

Ad esempio avendo una soluzione attesa come [26,[a,c,b,e,d]], è possibile memorizzarla in un oggetto di tipo `CompSol` in questo modo:

```
SetCompSol expSol = new SetCompSol();
expSol = ReadFrom.readCompSol("[ 26 [ a c b e d ]] $");
```

Dal punto di vista dell'implementazione sono stati creati due campi `private`, `pathName` e `problemName`, utilizzati rispettivamente per memorizzare il percorso del file contenente la soluzione attesa ed il nome del problema (di norma è lo stesso del programma oggetto del test) che servirà per l'identificazione della soluzione corrispondente all'interno del file.

Dato che le variabili **strutturate** sono molto più complesse da gestire rispetto a quelle **atomiche**, per poter leggere e caricare una soluzione composta da variabili di questo tipo, la classe `ReadFrom` sfrutta i metodi della classe `BuiltCompSol` il cui principale scopo è quello di ricreare i livelli di profondità degli elementi costituenti le variabili.

La classe è composta da tre campi `private` di tipo `Vector`, due dei quali (*partSol* e *buildSol*) servono a contenere temporaneamente una parte della soluzione durante la sua ricostruzione, mentre il terzo (*compSol*) serve per la memorizzazione della soluzione finale.

Per generare la complessità di una soluzione partendo da una stringa, la classe ha a disposizione il seguente insieme di metodi `public` necessari per la manipolazione delle variabili di una soluzione:

`Vector<Object> getCompSol()`
Restituisce la soluzione finale precedentemente memorizzata in `compSol`.

`void clearCompSol()`
: Pulisce i `Vector compSol, buildSol` e `partSol`, dopo che una soluzione è stata passata all'oggetto di invocazione di uno dei metodi della classe `ReadFrom`.

`void mkSol(String sol)`
: Questo metodo serve a ricreare i livelli di una soluzione (composta da `LList` o `LSet`). Al suo interno attraverso un oggetto di tipo `StringTokenizer`, vengono letti i caratteri di una soluzione uno alla volta. La soluzione finale viene memorizzata in `compSol`. Prende in input la stringa `sol` contenente la soluzione.

`void cleanRefreshVecs(int delComp, int delBuild, int delPart)`
: Metodo che aggiorna i `Vector buildSol` e `partSol`. Prende in input il livello al quale è arrivata la ricostruzione (`levLst`) e il numero di elementi da rimuovere da `partSol` (`delPart`), oppure da `buildSol` (`delBuild`).

4.3 Il confronto: la classe `AssertionSol`

Come nel caso della classe `PrepareSol` anche questa classe contiene metodi per la gestione di tutti i tipi di dato trattati.

Per ciò che riguarda variabili di tipo `atomico`, questa classe mette a disposizione il seguente metodo `public`:

`void assertEqSol(SimpSol found, SimpSol expected)`
: Questo metodo prende due oggetti di tipo `SimpSol` in input (la soluzione attuale `found` e quella attesa `expected`) ed esegue il confronto tra le variabili memorizzate in essi.

Analogamente per variabili di tipo **strutturato** è possibile utilizzare il seguente metodo `public`:

`void assertEqSol(CompSol found, CompSol expected)`
: Questo metodo prende due oggetti di tipo `CompSol` in input (la soluzione attuale `found` e quella attesa `expected`) ed esegue il confronto tra le variabili memorizzate in essi.

Dal punto di vista dell'implementazione la classe è composta inoltre da due metodi `private`, che costituiscono il nucleo centrale della classe, infatti

sono questi che eseguono effettivamente il confronto tra le soluzioni attese e quelle ottenute in output dai vari programmi oggetto del testing.
Questi metodi sono:

`boolean assertEqSolOpz(SimpSol found, SimpSol expected)`
: Metodo che prende in input due oggetti di tipo `SimpSol` e ne esegue il confronto.

`boolean assertEqSolOpz(CompSol found, CompSol expected)`
: Metodo che prende in input due oggetti di tipo `CompSol` e ne esegue il confronto.

Per l'esecuzione del test, i metodi `public` effettueranno una chiamata ad uno di questi metodi a seconda che i valori delle variabili siano di tipo `SimpSol` oppure `CompSol`. Per come sono stati implementati i due metodi `private` è possibile verificare che le soluzioni siano uguali, ed inoltre anche che quella attuale sia un sottoinsieme di quella attesa. Infatti questo metodo ritorna `true`, anche nel caso in cui la soluzione attesa abbia più variabili di quella attuale, a patto che per ogni variabile di quest'ultima, ce ne sia una di nome e valore equivalenti nella soluzione attesa.

Per l'esecuzione del confronto viene esaminato il vettore dei nomi della soluzione (`found`) ottenuta dal programma, controllando che ogni nome di variabile della soluzione attesa (`expected`) vi sia contenuto.

Se questo controllo viene superato con successo, viene eseguito il confronto uno a uno tra i valori di ciascuna variabile attesa con quelli di ciascuna variabile della soluzione attuale.

Si osservi il seguente esempio:

```
CompSol foundSolution = new CompSol();
CompSol expectedSolution = new CompSol();
foundSolution = PrepareSol.prepareCompSol(result);
expectedSolution =
ReadFrom.readCompSol("[ 26 [ a c b e d ]] $");
AssertionSol assertOneOf = new AssertionSol();
assertOneOf.assertEqSol(foundSolution, expectedSolution);
```

Nella porzione di codice riportata viene mostrata la costruzione di un oggetto di questo tipo. Innanzitutto è necessario creare ed inizializzare le due soluzioni da testare (`foundSolution` e `expectedSolution`) (cfr. 4.1). In seguito sarà creata un'istanza della classe `AsertionSol` (`assertOneOf`), che prenderà in input le due soluzioni ed eseguirà il confronto.

4.4 Esecuzione del test

4.4.1 Procedura per l'esecuzione del test

I passaggi necessari per l'esecuzione di un test attraverso l'utilizzo degli strumenti descritti sono stati sviluppati in modo tale da apportare il minor numero di modifiche ai programmi testati.

In seguito verranno mostrati i passi della procedura necessari per impostare un qualsiasi test:

1. Creazione di un'istanza della classe `SimpSol` (oppure `CompSol`) per la memorizzazione della soluzione restituita dal programma oggetto del testing, inizializzata attraverso uno dei metodi della classe `PrepareSol`.

2. Creazione di un'istanza della classe `SimpSol` (oppure `CompSol`) per la memorizzazione della soluzione attesa e sua inizializzazione attraverso l'ausilio dei metodi della classe `ReadFrom`.

3. Creazione di un'istanza della classe `AssertionSol`, i metodi della quale andranno ad eseguire il confronto, testando la correttezza della soluzione.

4.4.2 Integrazione con JUnit

I programmi nei quali è stata applicata questa strategia di testing erano già stati implementati per testare le varie funzionalità di *JSetL*, ma non presentavano la struttura classica del test.

Per giungere alla realizzazione del test nella sua forma finale è stato pertanto necessario effettuare qualche modifica al codice in modo tale da rimanere il più possibile aderenti alle regole di esecuzione di un test in *JUnit* (cfr. 3.2), con il minor numero possibile di modifiche al corpo del programma.

In analogia alla strategia classica di costruzione di un test in *JUnit*, si è fatto ricorso all'utilizzo delle *annotazioni* `@Before`, `@After` e `@Test`. Per quanto riguarda l'utilizzo del meccanismo delle *asserzioni* non sono state utilizzate quelle appartenenti alla classe `junit.framework.Assert` (cfr. 3.2), ma quelle realizzate in questo lavoro di tesi (cfr. 4.3).

Pertanto la modifica più consistente al programma è stata quella di rimuovere tutte le parti di codice riguardanti eventuali stampe a video delle soluzioni e l'inserimento delle strutture realizzate per il test per soluzioni complesse. Mentre con *JUnit* per l'esecuzione di test su soluzioni di questo tipo è necessario utilizzare gli output di un programma in modo sequenziale,

in questo caso sono state create apposite strutture (cfr. 4.1, 4.2) che consentono l'incapsulamento degli output di un programma, in modo tale da poter eseguire il test direttamente sull'intera soluzione.

4.5 Alcuni esempi di utilizzo

Verranno mostrati di seguito alcuni esempi nei quali è stata implementata la strategia di testing descritta in questo capitolo.

`SendMoreMoney`

Questo programma risolve il problema *Send + More = Money*. Questo fa parte di un insieme di problemi appartenenti alla *Criptoaritmetica*, ovvero l'arte di inventare e risolvere calcoli crittografati [17]. Infatti questo è un problema nel quale la computazione viene eseguita su delle cifre che sono state sostituite da lettere o in generale da un simbolo qualsiasi.

Per la realizzazione del test sono stati creati gli oggetti `foundSolution` di tipo `SimpSol` per la gestione della soluzione restituita dal programma, e `expectedSolution` anch'essa di tipo `SimpSol` per la memorizzazione della soluzione attesa.

Infine è stata creata un'istanza della classe `AssertionSol` così da poter invocare il metodo `assertEqSol(SimpSol found, SimpSol expexted)` (cfr. 4.3) per l'esecuzione del test.

Di seguito è stato riportato il codice del problema così da poter notare le caratteristiche appena descritte.

```
@Test
public static void testSendMoreMoney() throws Failure,
IOException {
    IntLVar s = new IntLVar("S", 0, 9);
    IntLVar e = new IntLVar("E", 0, 9);
    IntLVar n = new IntLVar("N", 0, 9);
    IntLVar d = new IntLVar("D", 0, 9);
    IntLVar m = new IntLVar("M", 0, 9);
    IntLVar o = new IntLVar("O", 0, 9);
    IntLVar r = new IntLVar("R", 0, 9);
    IntLVar y = new IntLVar("Y", 0, 9);
    IntLVar send = new IntLVar("SEND");
    IntLVar more = new IntLVar("MORE");
    IntLVar money = new IntLVar("MONEY");
    IntLVar[] letters = {s, e, n, d, m, o, r, y};
    SolverClass solver = new SolverClass();
    solver.add(s.neq(0).and(m.neq(0)));
    solver.add(IntLVar.allDifferent(letters));
```

```
        solver.add(send.eq(
              s.mul(1000).sum(e.mul(100).sum(n.mul(10).sum(d)))));
        solver.add(more.eq(
             m.mul(1000).sum(o.mul(100).sum(r.mul(10).sum(e)))));
        solver.add(money.eq(
             m.mul(10000).sum(o.mul(1000).sum(n.mul(100).
              sum(e.mul(10).sum(y))))));
        solver.add(money.eq(send.sum(more)));
        solver.add(s.label());
        solver.add(e.label());

             /* Test the solution */

        assertTrue(solver.check());
        Vector<String> vars = new Vector<String>();
        vars.add(send.toString());
        vars.add(more.toString());
        vars.add(money.toString());
        SimpSol foundSolution = new SimpSol();
        SimpSol expectedSolution = new SimpSol();
        foundSolution = PrepareSol.prepareSimpSol(vars);
        String expected = "x-0 9567 x-1 1085 x-2 10652";
        expectedSolution = ReadFrom.readSolution(expected);
        AssertionSol assertOneOf = new AssertionSol();
        assertOneOf.assertEqSol(foundSolution, expectedSolution);
}
```

Come già si era potuto notare nel paragrafo 4.1.3, le soluzioni attese sono state codificate nel modo seguente:

x-0 9567 x-1 1085 x-2 10652

Il *Travelling Salesman Problem*: `TSP_Weighted`

Questo è un problema di ricerca di un cammino all'interno di un grafo.

Dato un grafo pesato, la sua origine ed un intero K, il programma determina se esiste un percorso all'interno del grafo, che inizi dall'origine, passi esattamente una volta su ciascun nodo ed abbia un costo inferiore oppure uguale a K. In generale dato un grafo qualsiasi è possibile che esistano più percorsi di questo tipo, ma nel caso specifico il `TSP_Weighted` restituirà una singola soluzione.

Per la realizzazione del test sono stati creati gli oggetti `foundSolution` di tipo `CompSol` per la gestione della soluzione restituita da `TSP_Weighted` e `expectedSolution` anch'essa di tipo `CompSol` per la memorizzazione della soluzione attesa. Infine è stata creata un'istanza della classe `AssertionSol`

così da poter invocare il metodo `AssertEqSol(CompSol found, CompSol expexted)` (cfr. 4.3) per l'esecuzione del test.

Di seguito si possono osservare le modifiche apportate al codice del problema.

```
@Test
public static void testTSP_Weighted() throws Failure,
IOException {
      char a='a', b='b', c='c', d='d', e='e';
      LSet nodes = LSet.empty().ins(e).ins(d).ins(c).ins(b).
            ins(a).setName("Node");
      LList w1 = LList.empty().ins(4).ins(b).ins(a);
      LList w2 = LList.empty().ins(8).ins(c).ins(a);
      LList w3 = LList.empty().ins(0).ins(b).ins(c);
      LList w4 = LList.empty().ins(1).ins(a).ins(b);
      LList w5 = LList.empty().ins(3).ins(e).ins(b);
      LList w6 = LList.empty().ins(7).ins(a).ins(d);
      LList w7 = LList.empty().ins(8).ins(d).ins(e);
      LSet edges_weights = LSet.empty().ins(w7).ins(w6).ins(w5).
            ins(w4).ins(w3).ins(w2).ins(w1).setName("Weights");
      int limit = 26;

            /* Test a solution */

      LList result;
      result = TSP_WeightedTest.tsp(
            nodes, edges_weights, a, limit);
      CompSol foundSolution = new CompSol();
      CompSol expectedSolution = new CompSol();
      foundSolution = PrepareSol.prepareCompSol(result);
      expectedSolution = ReadFrom.readCompSol(
            "TSP_Weighted","data\\expectedComplexSolution.txt");
      AssertionSol assertOneOf = new AssertionSol();
      assertOneOf.assertEqSol(foundSolution, expectedSolution);
}
```

Come già si era potuto notare nel paragrafo 4.1.3, la soluzione attesa memorizzata opportunamente nel file `expectedComplexSolution.txt`, è stata codificata nel modo seguente:

```
TSP_Weighted
[ 26 [ a c b e d ]] $
!!!
```

Capitolo 5

Soluzioni complesse multiple

Con il termine di "*soluzioni complesse multiple*" si intendono insiemi di soluzioni, ciascuna costituita da più di una variabile.

Molti dei programmi utilizzati nell'esecuzione del test di *JSetL* possono restituire più di una soluzione. In questi è stato necessario aggiornare l'insieme di classi utilizzate in modo da poter gestire anche queste situazioni ed apportare qualche piccola modifica alla procedura di testing (cfr. 5.3).

In questo capitolo sarà analizzato il modo nel quale soluzioni di questo tipo sono state codificate e come in seguito siano state lette e gestite.

5.1 Rappresentazione delle soluzioni

Dato che l'oggetto sul quale si vuole eseguire il test è costituito da un insieme di soluzioni, è sensato pensare a delle strutture che possano permettere la memorizzazione di più soluzioni di tipo `SimpSol` (cfr. 4.1.1) oppure `CompSol` (cfr. 4.1.2).

A tale scopo sono state create le classi `SetSimpSol` e `SetCompSol` composte entrambe da tre metodi `public` che consentono il loro inserimento e la loro lettura.

5.1.1 La classe `SetSimpSol`

La classe `SetSimpSol` è stata implementata per consentire la gestione di insiemi di soluzioni costituite da variabili di tipo **atomico**, ovvero di tipo `SimpSol` (cfr. 4.1.1).

Per l'inserimento e la lettura di tale insieme di soluzioni sono stati creati tre metodi `public`:

`void addSolution(SimpSol solution)`
Questo metodo aggiunge la nuova soluzione di tipo `SimpSol` contenuta in `solution` al `Set` delle soluzioni.

`Set<SimpSol> getSolSet()`
Restituisce il `Set` contenente l'insieme di soluzioni.

`int getNumOfSol()`
Metodo che restituisce il numero di soluzioni attualmente presenti nel `Set`.

Per una migliore comprensione di quanto detto, si osservi il seguente esempio:

```
LVar a = new LVar('a');
LVar b = new LVar('b');
LVar c = new LVar('c');
Vector<String> vars = new Vector<String>();
vars.add(a.getValue().toString());
vars.add(b.getValue().toString());
vars.add(c.getValue().toString());
String names[] = new String[vars.size()];
String elements[] = new String[vars.size()];
SimpSol solFound = new SimpSol();
SetSimpSol foundSolution = new SetSimpSol();
do{
    for(int i = 0; i < vars.size(); i++){
        names[i] = "x-" + i;
        elements[i] = vars.get(i);
    }
    solFound.addSolution(names, elements);
    foundSolution.addSolution(solFound);
}while(solver.nextSolution());
```

In questo esempio si può notare che è stato implementato un ciclo `do{} while()` nel quale, a seguito del ciclo `for()` (definito da zero a `vars.length`, dove `vars` contiene le variabili di una soluzione), viene caricata una singola soluzione in `solFound` che è un oggetto di tipo `SimpSol` (cfr. 4.1.1). Successivamente ad opera del metodo `addSolution(solFound)` la soluzione sarà caricata nel `Set` di soluzioni attuali (`foundSolution`).

Dal punto di vista dell'implementazione la classe è composta da due campi `private`, il `Set<SimpSol> solutionSet` che conterrà l'insieme di soluzioni e il contatore di `numberOfSolution`.

5.1.2 La classe `SetCompSol`

La classe `SetCompSol` gestisce il caso di insiemi di soluzioni le cui variabili sono di tipo **strutturato**, ovvero di tipo `CompSol` (cfr. 4.1.2).

Anche in questo caso pertanto sono stati creati tre metodi `public`, che consentono di operare su insiemi di soluzioni di questo tipo:

`void addSolution(CompSol aSol)`
: Questo metodo aggiunge la nuova soluzione di tipo `CompSol` contenuta in `aSol`, al `Set` di soluzioni.

`Set<CompSol> getSolSet()`
: Restituisce il `Set solutionLList`, contenente l'insieme di soluzioni.

`int getNumOfSol()`
: Metodo che restituisce il numero di soluzioni attualmente presenti nel `Set`.

E' possibile notare di seguito un esempio di utilizzo di questa classe:

```
LVar a = new LVar('a');
LVar b = new LVar('b');
LList l = LList.empty().ins(a).ins(b);
LList l1 = LList.empty().ins(b).ins(a);
Vector<LCollection> vars = new Vector<LCollection>();
vars.add(l);
vars.add(l1);
CompSol solFound = new CompSol();
SetCompSol foundSolution = new SetCompSol();
do{
      Iterator<?> vars = sol.iterator();
      while (var.hasNext())
            solFound.addSolution(vars.next());
      foundSolution.addSolution(solFound);
}while(solver.nextSolution());
```

In questo esempio si può notare l'utilizzo del ciclo `do{} while()`, nel quale vengono aggiunte le variabili di una singola soluzione al `Vector<String>vars` tramite il metodo `add(String)`. In seguito la soluzione viene memorizzata in `solFound` che è un oggetto di tipo `CompSol` (cfr. 4.1.2) tramite il metodo `addSolution(LCollection solFound)`. Infine la soluzione viene caricata nel `Set` di soluzioni `foundSolution` (di tipo `CompSol`) ad opera del metodo `addSolution(CompSol solFound)`.

La classe è stata implementata attraverso l'utilizzo di due campi `private`, un `Set` in questo caso formato da elementi di tipo `CompSol` nel quale saranno contenute le soluzioni e dal contatore `solCount`.

5.1.3 Codifica di soluzioni multiple

Come nel caso di *soluzioni singole* (cfr. 4.1.3), sono state adottate due codifiche a seconda che il dato trattato fosse di tipo **atomico**, oppure **strutturato**. L'unica differenza pratica nella codifica è che nel caso di insiemi di soluzioni esse dovranno essere disposte nel file successivamente una dopo l'altra.

Anche in questo caso per soluzioni formate da tipi di dato atomico sarà mantenuta la stessa codifica sintattica adottata per le soluzioni singole:

```
x-0 1 x-1 7 x-2 5 x-3 0 x-4 2 x-5 4 x-6 6 x-7 3
x-0 1 x-1 7 x-2 5 x-3 0 x-4 2 x-5 4 x-6 6 x-7 3
x-0 1 x-1 4 x-2 6 x-3 0 x-4 2 x-5 7 x-6 5 x-7 3
x-0 2 x-1 5 x-2 3 x-3 0 x-4 7 x-5 4 x-6 6 x-7 1
```

Nell'esempio precedente si può osservare parte della soluzione al problema delle *Otto Regine* (cfr. 5.4), che in realtà ne prevede novantadue. Si nota che le soluzioni sono elencate successivamente una dopo l'altra e che la codifica rimane la stessa del caso di *soluzioni singole*.

Anche per le soluzioni le cui variabili sono formate da tipi di dato **strutturato** vale quanto detto riguardo alle *soluzioni singole* (cfr. 4.1.3).

Si osservi il seguente esempio:

```
[ 1 ] $ [ 2 3 ] $
[ 1 2 ] $ [ 3 ] $
[ ] $ [ 1 2 3 ] $
[ 1 2 3 ] $ [ ] $
```

Questo esempio mostra la codifica delle quattro soluzioni per un problema di nome `SplitLList` (cfr. 5.4), ciascuna delle quali è composta da due variabili. Anche in questo caso la codifica della singola soluzione non cambia.

5.2 Aggiornamento delle strutture di lettura e confronto

Come accennato all'inizio del capitolo per gestire questo tipo di soluzioni in modo più efficiente, è stato necessario apportare alcune modifiche alle classi utilizzate nel caso di *soluzioni singole* (cfr. 4.1).

In conseguenza all'aggiunta delle due nuove classi `SetSimpSol` (cfr. 5.1.1) e `SetCompSol` (cfr. 5.1.2) per la memorizzazione dell'insieme di soluzioni è stato necessario modificare anche le classi `ReadFrom` in modo tale da poter leggere un insieme di soluzioni attese e la classe `AssertionSol` per consentire l'esecuzione del test su soluzioni di questo tipo.
Ad esempio la lettura di più soluzioni da file avviene nel seguente modo:

```
SetCompSol expSol = new SetCompSol();
expSol = ReadFrom.
      readMoreCompSol("TSP_Weighted", "data\\expSolution.txt");
```

Per quanto riguarda la costruzione di soluzioni esplicite (ovvero quelle attuali) non è stato necessario apportare alcuna modifica. Infatti l'aggiunta di una singola soluzione al `Set` può essere eseguita inizializzando un oggetto di tipo `SimpSol` (oppure `CompSol`) all'interno di un ciclo (`do{} while()` oppure `for()`) attraverso uno dei metodi della classe `PrepareSol` (cfr. 4.2.1). In seguito sempre all'interno del ciclo, la singola soluzione sarà memorizzata nel `Set` (come si può notare negli esempi riportati nel paragrafo 4.1).

5.2.1 Insiemi di soluzioni attese: la classe `ReadFrom`

Per fare in modo di poter caricare il `Set` con l'insieme di soluzioni attese è stato necessario aggiungere due metodi a seconda che il tipo di variabili costituenti le soluzioni fosse **atomico** oppure **strutturato**.

Anche in questo caso la lettura di variabili di tipo **strutturato** si avvale dell'ausilio dei metodi della classe `BuiltCompSol` (cfr. 4.2.2).

In questo caso è preferibile la lettura delle soluzioni da file, dato che un insieme di soluzioni poco si presta per essere passato direttamente come parametro.

Per la gestione di soluzioni costituite da variabili di tipo **atomico** è stato sviluppato il seguente metodo:

`SetSimpSol readMoreSolution(String name, String path)`
: Apre il file nel percorso definito in `path`, trova le soluzioni del problema di nome `name` e restituisce l'insieme di soluzioni trovate.

Mentre per gestire soluzioni con variabili di tipo `strutturato`:

`SetCompSol readMoreCompSol(String name, String path)`
: Apre il file nel percorso definito in `path`, trova le soluzioni del problema di nome `name` e restituisce l'insieme di soluzioni trovate.

5.2.2 Confronto tra insiemi di soluzioni: la classe `AssertionSol`

Eseguire un test su *soluzioni multiple* non è molto differente dall'eseguirlo su *soluzioni singole*, a patto che si apportino le modifiche necessarie per poter scomporre il `Set` di soluzioni nei suoi elementi costituenti (`SimpSol` oppure `CompSol`) in modo tale da ricadere nel caso base (cfr. 4.3).

Per questo motivo a questa classe sono stati aggiunti quattro metodi, due per ciascun tipo di soluzione (`atomica` oppure `strutturata`).

Per la gestione del test su soluzioni con variabili di tipo **atomico**, sono stati sviluppati i seguenti metodi:

`void assertIsSol(SimpSol found, SetSimpSol expected)`
: Questo metodo prende in input il `Set` di soluzioni attese (`expected`) ed un oggetto di tipo `SimpSol`, contenente la soluzione attuale (`found`) e ne esegue il confronto.

`void isASubSet(SetSimpSol subSet, SetSimpSol set)`
: Metodo che riceve in input due oggetti di tipo `SetSimpSol`, l'insieme di soluzioni attuali (`subSet`) e quello di soluzioni attese (`set`). Questo metodo oltre a testare l'uguaglianza tra i due insiemi di soluzioni, restituisce `true` anche nel caso in cui la soluzione attuale sia un sottoinsieme di quella attesa.

In modo analogo i seguenti metodi gestiscono il test su variabili di tipo `strutturato`:

`void assertIsSol(CompSol found, SetCompSol expected)`
: Metodo che prende in input un oggetto di tipo `CompSol`, contenente la soluzione attuale (`found`), ed un `Set` di oggetti di tipo `CompSol` contenente l'insieme di tutte le possibili soluzioni attese (`expected`).

`void isASubSet(SetCompSol subSet, SetCompSol set)`
: Metodo che riceve in input due oggetti di tipo `SetCompSol`, l'insieme di soluzioni attuali (`subSet`) e quello di soluzioni attese (`set`). Analogamente al caso precedente restituisce `true` sia in caso di uguaglianza tra le soluzioni, che nel caso in cui quella attuale sia un sottoinsieme di quella attesa.

Anche in questo caso per l'esecuzione del test, i metodi `public` effettueranno si appoggiano ai metodi `private` della classe (cfr. 4.3), scomponendo i `Set` di soluzioni nei propri elementi costituenti.

5.3 Procedura per l'esecuzione del test

Per l'esecuzione dei vari test su *soluzioni multiple* la procedura è stata leggermente modificata con l'aggiunta di 3 passi, per consentire il corretto caricamento delle soluzioni.

Di seguito sarà mostrata la procedura aggiornata:

1. Creazione di un'istanza della classe `SimpSol` (oppure `CompSol`) per la memorizzazione di una singola soluzione restituita dal programma oggetto del testing, che sarà inizializzata da uno dei metodi della classe `PrepareSol`.

2. Creazione di un'istanza della classe `SetSimpSol` (oppure `SetCompSol`) che servirà a contenere l'insieme di soluzioni attuali finale.

3. Creazione di un oggetto di tipo `Vector` (con elementi di tipo `String` oppure `LCollection` a seconda dei casi), nei quali saranno memorizzate le variabili relative ad una soluzione.

4. Implementazione di un ciclo `do{} while()`, necessario per comporre una singola soluzione e il suo caricamento nel `Set` di soluzioni finali.

5. Creazione di un'istanza della classe `SetSimpSol` (oppure `SetCompSol`) per la memorizzazione dell'insieme (anche unitario) delle soluzioni attese e sua inizializzazione attraverso l'utilizzo dei metodi della classe `ReadFrom`, per l'opportuna lettura delle soluzioni da file.

6. Creazione di un'istanza della classe `AssertionSol`, i metodi della quale andranno ad eseguire il confronto, testando la correttezza della soluzione.

5.4 Alcuni esempi di utilizzo

Verranno mostrati di seguito alcuni esempi nei quali è stata implementata la strategia di testing descritta in questo capitolo.

Il problema delle *Otto Regine*: `Queens`

Il problema delle *Otto Regine* è un problema di matematica inerente la problematica di come disporre otto regine su una scacchiera, in modo che per ciascuna regina posizionata non ce ne siano altre sulla stessa riga, colonna e diagonale.

Per l'implementazione del test sono stati creati un oggetto di tipo `SimpSol` (`solFound`), che conterrà una singola soluzione attesa e i due oggetti di tipo `SetSimpSol` per la memorizzazione degli insiemi di soluzioni attuali (`foundSolution`) e di quelle attese (`expectedSolution`). Inoltre è stato creato un `Vector<String>` per catturare le variabili di una soluzione, cosa che avviene all'interno del ciclo `do{}while()` nel quale viene anche caricata un'intera soluzione in `foundSolution`.

Infine attraverso l'istanza della classe `AssertionSol` è stata eseguita la chiamata al metodo `isASubSet(SetSimpSol subSet, SetSimpSol set)` che eseguirà il test sulle soluzioni.

E' stato riportato di seguito il codice del programma:

```
@Test
public void testQueens() throws Failure, IOException{
    int i, j = 0;
    IntLVar[] vars = new IntLVar[N];
    for (i = 0; i < N; ++i)
        vars[i] = new IntLVar(0, N - 1);
    solver.add(IntLVar.allDifferent(vars));
    for (i = 0; i < N - 1; ++i)
        for (j = i + 1; j < N; ++j) {
            MultiInterval dom = new MultiInterval(i - j);
            dom.add(j - i);
            solver.add(vars[i].sub(vars[j]).ndom(dom));
        }
    for (i = 0; i < N - 1; ++i)
    solver.add(vars[i].label(ValHeuristic.MEDIAN));

        /* Test the solution */

    assertTrue(solver.check());
    SimpSol solFound = new SimpSol();
    SetSimpSol foundSolution = new SetSimpSol();
    Vector<String> aSol = new Vector<String>();
    do{
        for(int l = 0; l < vars.length; l++)
            aSol.add(vars[l].toString());
        solFound = PrepareSol.prepareSimpSol(aSol);
        foundSolution.addSolution(solFound);
        aSol.clear();
    }while(solver.nextSolution());
    SetSimpSol expectedSolution = new SetSimpSol();
    expectedSolution = ReadFrom.readMoreSolution(
        "Queens", "data\\expectedComplexSolution.txt");
    AssertionSol assertOneOf = new AssertionSol();
    assertOneOf.isASubSet(foundSolution, expectedSolution);
```

}

Come già accennato (cfr. 5.1.3) le soluzioni attese di questo programma sono novantadue e sono state opportunamente codificate nel file `expectedComplexSolution.txt`.

Di seguito è riportato un esempio di codifica di quattro soluzioni:

```
Queens
x-0 1 x-1 7 x-2 5 x-3 0 x-4 2 x-5 4 x-6 6 x-7 3
x-0 1 x-1 7 x-2 5 x-3 0 x-4 2 x-5 4 x-6 6 x-7 3
x-0 1 x-1 4 x-2 6 x-3 0 x-4 2 x-5 7 x-6 5 x-7 3
x-0 2 x-1 5 x-2 3 x-3 0 x-4 7 x-5 4 x-6 6 x-7 1
!!!
```

Suddividere una lista: `SplitLList`

Questo programma prende una lista in input e restituisce l'insieme delle possibili suddivisioni di essa.

Per l'esecuzione del test sono stati creati gli oggetti `solFound` di tipo `CompSol` per la memorizzazione di una singola soluzione attesa dal programma, `foundSolution` ed `expectedSolution` entrambi di tipo `SetCompSol` per la memorizzazione degli insiemi di soluzioni attuali e di quelle attese rispettivamente. Anche in questo caso è stato creato un `Vector<LCollection>` per la cattura delle variabili appartenenti ad una soluzione ed è stato fatto ricorso all'utilizzo del ciclo `do{}while()` per la creazione di una soluzione ed il suo caricamento in `foundSolution`.

Infine attraverso l'istanza della classe `AssertionSol` è stato possibile chiamare il metodo `isASubset(SetCompSol subSet, SetCompSol set)` deputato all'esecuzione del test.

E' possibile osservare di seguito l'implementazione del test:

```
@Test
public static void testSplitLList() throws Failure,
IOException {
    LList l1 = new LList("l1");
    LList l2 = new LList("l2");
    LList l3 = LList.empty().ins(3).ins(2).
        ins(1).setName("l3");
    ListOps listOps = new ListOps(solver);
    solver.add(listOps.concat(l1,l2,l3));
```

```
        /* Test the solution */

    assertTrue(solver.check());
    SetCompSol foundSolution = new SetCompSol();
    Vector<LCollection> loadSol = new Vector <LCollection >();
    CompSol solFound = new CompSol();

    do{
        loadSol.add(l1);
        loadSol.add(l2);
        solFound = PrepareSol.prepareCompSol(loadSol);
        loadSol.clear();
        foundSolution.addSolution(solFound);
    }while(solver.nextSolution());
    SetCompSol expectedSolution = new SetCompSol();
    expectedSolution = ReadFrom.readMoreCompSol(
    "SplitLList", "data\\expectedComplexSolution.txt");
    AssertionSol assertOneOf = new AssertionSol();
    assertOneOf.isASubSet(foundSolution, expectedSolution);
}
```

Come già si era potuto notare, le soluzioni attese di questo programma opportunamente codificate (cfr. 5.1.3) nel file `expectedComplexSolution.txt`, sono le seguenti:

```
SplitLList
[ 1 ] $ [ 2 3 ] $
[ 1 2 ] $ [ 3 ] $
[ ] $ [ 1 2 3 ] $
[ 1 2 3 ] $ [ ] $
!!!
```

Capitolo 6

Conclusioni

Il lavoro di tesi ha avuto come scopo principale quello di progettare e realizzare un insieme di classi e metodi per consentire il testing di soluzioni costituite da più variabili, ciascuna con la possibilità di assumere più di un valore. Tale valore può essere di tipo strutturato come `LList` e `LSet` oppure atomico come `String` ed `int`. In secondo luogo è stato ripreso il lavoro di testing delle classi della libreria *JSetL*.

Come prima cosa è stata effettuata un'analisi del dominio di input delle tre principali classi di *JSetL*: `Contraint`, `LVar`, `LSet`. Questo lavoro di analisi ha portato ad un partizionamento degli input in classi di equivalenza allo scopo di produrre l'insieme di **Test Case** utilizzati per il testing. I test prodotti sono stati successivamente inseriti nella classe `AllTest`, precedentemente creata in altri lavori di tesi per eseguire automaticamente tutti i test realizzati.

In seguito, a partire da un'analisi di *JUnit*, sono state sviluppate delle procedure per facilitare il testing di *JSetL*. Più precisamente i meccanismi attraverso i quali *JUnit* esegue il processo di testing, in particolare quello delle *asserzioni* sono stati estesi principalmente per consentire il test di soluzioni costituite da più variabili con valori sia di tipo strutturato (`LList` e `LSet`), che di tipo primitivo (`String` e `int`). E' stato inoltre sviluppato un insieme di classi e metodi per la corretta gestione di soluzioni di questo tipo ed una strategia di codifica delle soluzioni che potesse risultare efficace per tale scopo.

Bibliografia

[1] Laurie Williams
Testing Overview and Black-Box Testing Techniques

[2] A. Fantechi
Il Testing
http://www.dsi.unifi.it/~fantechi/INFIND/testing.ppt

[3] Giuseppe A. Di Lucca
Tecniche di Testing
http://www.ing.unisannio.it/dilucca/GSSW/materiale2011/testing_Funzionale_11.pdf

[4] Giuseppe A. Di Lucca
Testing Object-Oriented
http://www.ing.unisannio.it/dilucca/GSSW/materiale2011/testing_OO_11.pdf

[5] Agostino Dovier, Andrea Formisano
Programmazione Dichiarativa in Prolog, CLP e ASP
http://users.dimi.uniud.it/~agostino.dovier/DID/lnc.pdf

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

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

[8] JUnit Home Page.
http://www.junit.org/home

[9] A. Fasolino
Testing di Sistemi Object-Oriented
http://unina.stidue.net/Ingegneria del Software/Materiale/Slides/2011-2012/14/testingOO.pdf

[10] G. Mecca
Programmazione Orientata agli Oggetti in Linguaggio Java
`http://www.db.unibas.it/users/mecca/corsi/2004-2005/progOggettiI/materiale/Mecca-poo06-03-TestECorrezione-JUnit.pdf`

[11] Paolo Tonella
Analisi e Testing del Software
`http://selab.fbk.eu/swat/course.pdf`

[12] John B. Goodenough, Susan L. Gerhart
Toward a Theory of Test Data Selection

[13] Boris Beizer
Software Testing Techniques

[14] Eclipse Home Page
`http://www.eclipse.org/`

[15] Tesi di Lucia Guiglielmetti
Progettazione e realizzazione in Java di programmi di test per la libreria JSetL

[16] Tesi di Riccardo Zangrandi
Automatizzazione e controllo delle soluzioni per i test dello strumento TCK di JSR331

[17] Criptoaritmetica
L'Arte di Inventare e Risolvere Calcoli Crittografati
`http://utenti.quipo.it/base5/numeri/criptarit.htm`