UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI
Corso di Laurea in Informatica

Tesi di Laurea

Progettazione e realizzazione in Java di programmi di test per la libreria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Lucia Guglielmetti

Anno Accademico 2009/2010

Ai miei genitori, Maria Rosa e Giancarlo.
Ai miei fratelli, Carolina e Oreste.
A mio cognato, Andrea.
Ai miei nipoti, Camilla, Filippo e Matilde.

Ringraziamenti

Mi sembra doveroso, giunta a questo significativo momento della mia carriera scolastica, dedicare qualche parola alle persone che hanno contribuito più o meno direttamente al raggiungimento della laurea.

Il primo ringraziamento va quindi ai miei genitori e ai miei fratelli che in tutti questi anni hanno saputo sostenermi, permettendomi di portare a termine il corso di studi senza mai farmi mancare nulla.

Un sentito ringraziamento va anche al relatore, il Prof. Gianfranco Rossi, che nonostante i tanti impegni quotidiani è riuscito comunque a seguirmi durante il periodo di tirocinio e la stesura della tesi con grande disponibilità.

Desidero inoltre ringraziare tutti i miei compagni di corso e i ″matematici″ con i quali ho condiviso un lungo cammino fatto di momenti divertenti e momenti meno felici ma grazie ai quali ho trovato sempre la forza di continuare il percorso intrapreso.

Un grazie speciale va inoltre a Roberto, Gianni, Fabio, Riccardo, Daniele, Tino, Berna, Luca LR, Ama, Albertone, Stefano, Giuseppe, Chiara, Cinzia, Guenda, Carlotta, Federica, Camilla, che mi hanno aiutata con infinita pazienza, sostenuta nei momenti di sconforto, sopportandomi anche quando a volte riversavo su di loro le mie tensioni.

Non per ultimo voglio ricordare una persona che si è sempre interessata ai miei studi, spronandomi nei momenti piu difficili, quando alcuni esami sembravano insormontabili, sò che, se oggi potesse essere presente, sarebbe molto orgoglioso di me e sono certa che da lassù un grande aiuto lo avrà sicuramente dato.

A tutte queste persone un grazie di cuore.

Indice

Capitolo 1

Introduzione

Il testing è una parte essenziale e molto importante per la creazione di un programma robusto ed efficiente, che permette di ridurre in modo sostanziale i costi di manutenzione del programma sviluppato [2][3].

Purtroppo nessun testing può ridurre a zero la probabilità di non avere bug o failure, in quanto le possibili combinazioni di valori di input validi sono enormi, e non possono essere riprodotte in un tempo ragionevole. Tuttavia un buon testing può rendere la probabilità di malfunzionamenti abbastanza bassa da essere accettabile dall'utente.

In tempi relativamente recenti sono stati sviluppati diversi strumenti software per facilitare l'attività di testing. Tra questi, JUnit [7].

JUnit è uno dei più famosi framework open-source per i test unitari, creato da Kent Beck ed Erich Gamma nel 1997. Il successo di JUnit è stato tale che la stessa filosofia è andata oltre la sola versione Java. La famiglia di framework che comprende tutti i diversi porting di JUnit si chiama xUnit.

In questo lavoro di tesi ci si propone di utilizzare JUnit per creare ed eseguire un insieme di test che permettano di verificare il comportamento delle principali classi di una libreria Java denominata JSetL [5][6].

JSetL è una libreria che combina la programmazione OO di Java con i concetti fondamentali del CLP. È pensato principalmente come strumento di supporto alla programmazione dichiarativa, ovvero un paradigma di programmazione che consiste nello specificare che *cosa* il programma deve fare, piuttosto che *come* farlo, tipico invece della programmazione imperativa.

Per creare l'insieme dei test da eseguire tramite JUnit sarà utilizzata la tecnica dell'*equivalence testing* che prevede la suddivisione degli input in classi di equivalenza.

Una volta eseguiti i singoli test si intraprenderà la fase dell'integration testing utilizzando la strategia *bottom-up* in cui vengono integrati i test unit in sottoinsiemi più grandi per poi essere analizzati nel loro insieme.

Nel nostro caso la prima classe presa in considerazione sarà quella delle variabili logiche (`LVar`), dato che da essa dipendono tutte le altre classi analizzate. Dopodichè si controllerà il buon funzionamento della classe `IntLVar`, derivata dalla classe `LVar`, e quindi degli insiemi e delle liste logiche (classi `LSet` e `LList`) che possono contenere al loro interno variabili logiche. Successivamente si effettuerà il testing della classe `IntLSet`, derivata da `LSet`, ed infine si prenderanno in considerazioni i test per i vincoli (classe `Constraint`) presenti in tutte le classi precedenti.

La gerarchia di esecuzione dei test seguita può essere descritta nella seguente figura:

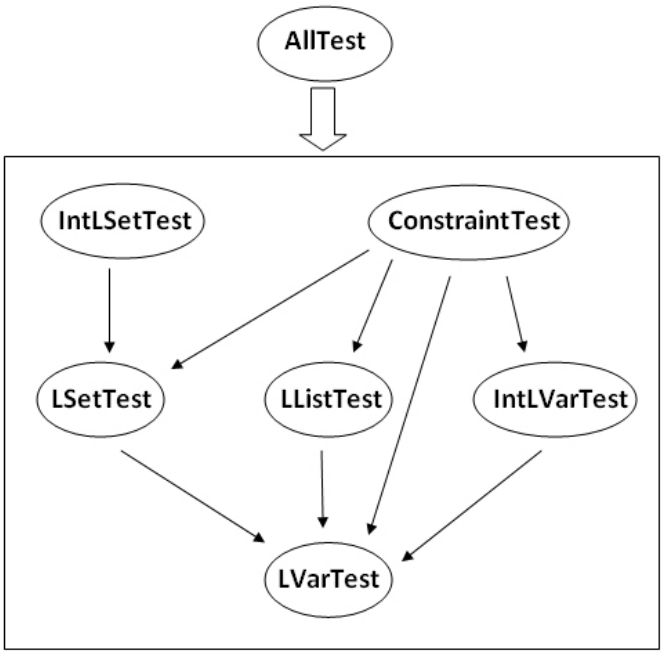

Figura 1.1: Diagramma di esecuzione dei testing

L'utilizzo di JUnit permetterà di attivare in modo estremamente semplice l'insieme dei test predisposti ed in modo altrettanto semplice di verificare la presenza di eventuali comportamenti discordanti da quelli previsti, evidenziando le parti del codice in cui si sono manifestati.

Il lavoro di tesi è organizzato nel seguente modo.
Nel capitolo 2 verrà data una descrizione del significato del testing e verranno presentate le varie fasi in cui si può dividere il lavoro di testing.
Il capitolo 3 sarà dedicato alla presentazione e all'utilizzo del JUnit.
Nel capitolo 4 troveremo una piccola panoramica di JSetL con alcune caratteristiche principali delle classi trattate.
Il capitolo 5 presenterà il lavoro svolto per ogni classe testata. Inizialmente verrà descritta ogni classe, verranno elencate le casistiche utilizzate per individuare i test, per poi elencare i metodi JSetL testati con alcuni esempi.
Nel capitolo 6 infine ci sarà spazio per conclusioni ed indicazioni di lavori futuri.

Capitolo 2

Il Testing

Il **testing** è una procedura usata per verificare comportamenti a run-time di un programma.

L'obiettivo del testing è quello di realizzare un software il più possibile esente da errori.

In questo capitolo verranno descritte le caratteristiche e le fasi del testing in modo d'avere una visuale il più completa possibile su tale procedura.

2.1 Caratteristiche principali

Per iniziare, occorre distinguere i failure del software, dai bug del software.

Il failure è un comportamento di un programma discordante dai requisiti espliciti o impliciti. In pratica, si verifica quando, in assenza di malfunzionamenti della piattaforma, il programma non fa quello che l'utente si aspetta.

Un bug è una sequenza di istruzioni, che, quando eseguita con particolari dati in input, genera un malfunzionamento. In pratica, si ha un malfunzionamento solo quando viene eseguito il programma che contiene il difetto, e solo se i dati di input sono tali da evidenziare l'errore.

Nessun testing può ridurre a zero la probabilità di non avere bug o failure, in quanto le possibili combinazioni di valori di input validi sono enormi, e non possono essere riprodotte in un tempo ragionevole. Tuttavia un buon testing può rendere la probabilità di malfunzionamenti abbastanza bassa da essere accettabile dall'utente. Il principale ostacolo al testing è sintetizzato nella Tesi di Dijkstra. Tale tesi afferma che il testing può indicare la presenza di errori, ma non ne può garantire l'assenza. Un corollario diretto di tale tesi è che il software error-free non esiste o non è certificabile.

L'attività di testing consiste nell'esecuzione di un prefissato insieme di *test* sul programma da controllare. I test si dividono principalmente in due famiglie:

- **I test unitari.** Sono test che vanno a verificare la correttezza direttamente del codice, in ogni sua piccola parte.

  Questi test, che hanno avuto una grande diffusione all'interno della extreme programming, vanno ideati ed eseguiti dal programmatore stesso per ogni porzione di codice da lui sviluppata.

- **I test funzionali.** Sono dei test che vanno a verificare che il sistema software nella sua completezza funzioni correttamente.

  Questi test trattano il sistema come se fosse una scatola nera alla quale danno degli input e verificano la correttezza degli output.

  Devono essere ideati ed eseguiti da personale che non ha partecipato alla costruzione del sistema.

Al fine di effettuare dei test accurati è buona abitudine suddividere i programmi di grosse dimensioni in moduli e su ciascuno di essi applicare un test unitario; solo successivamente verrà applicato un test funzionale sull'intero programma.

Un modulo è un'unità di programma con una sua struttura interna, definito con un determinato scopo, che offre all'esterno un certo insieme prefissato di servizi utilizzabili da altri moduli.

Un moduto è quindi caratterizzato da:

- l'insieme dei servizi che offre agli altri moduli;
- la modalità con cui tali servizi possono essere effettivamente utilizzati; l'insieme di tali modalità costituisce quella che viene chiamata *l'interfaccia del modulo*;
- l'insieme dei servizi che esso importa dagli altri moduli, ovvero l'insieme dei servizi offerti da altri moduli ed utilizzati dal modulo per le sue funzioni;
- la struttura interna, cioè l'insieme dei tipi, delle variabili e delle funzioni definiti nel modulo stesso, ma non visibili ed utilizzabili all'esterno;
- la forte coesione tra gli elementi interni che si contrappone alla bassa correlazione con le altre parti esterne.

Un modulo è perciò allo stesso tempo un *fornitore* di servizi ed un *cliente*, in quanto utilizzatore di servizi forniti da altri moduli.

L'uso dei moduli permette di procedere nello sviluppo del programma decidendo prima l'architettura globale, e concetrandosi poi su ogni singolo modulo, sulle sue funzioni e sulle sue caratteristiche.

Ciò favorisce anche uno sviluppo razionale da parte di un gruppo di progettisti. Ogni progettista può essere responsabile di un modulo, e può procedere sulla base della specifica di quali sono i servizi che il suo modulo deve offrire, e quali invece sono i servizi che gli altri moduli offriranno una volta progettati e realizzati.

Se la suddivisione in moduli è stata effettuata con criterio, ogni modulo può effettivamente essere sviluppato indipendentemente dagli altri, e può essere analizzato e verificato autonomamente, per mezzo di test che prevedono di attivare i servizi da esso offerti.

Nella programmazione orientata agli oggetti i moduli vengono rappresentati attraverso le classi e anche in questa tesi ogni volta che parleremo di classe ci riferiremo ai moduli.

Un testing accurato può dare una prova certa se un pezzo di codice funziona correttamente, con importanti vantaggi:

- non è influenzato dall'implementazione del programma
- si creano software più efficienti e robusti
- si riducono i costi di manutenzione dei software
- è semplice individuare e correggere gli errori

ma presenta tre problemi principali:

- non tutte le parti del programma possono essere testate
- sono evidenziati gli errori ma non la loro causa
- è costoso, in termini di uso delle macchine e di tempo umano

2.2 Fasi di Testing

La procedura di testing può essere suddivisa in varie fasi:

1. Unit testing

2. Integration testing
3. System testing
4. Acceptance testing
5. Installation testing

2.2.1 Unit testing

Nella fase di *unit testing* i singoli sottosistemi o oggetti vengono testati separatamente.

Questo comporta il vantaggio di ridurre il tempo di testing testando piccole unità di sistema singolarmente.

I candidati da sottoporre al testing vengono presi dal modello ad oggetti e dalla decomposizione in sottosistemi.

Gli unit testing possono essere divisi principalmente in due tipologie: *blackbox testing* e *whitebox testing*.

Il blackbox testing permette di testare le funzionalità del software ignorando il flusso di manipolazione delle strutture dati e delle variabili utilizzate. Può essere effettuato in due modi:

- **Equivalence testing.** Gli input vengono divisi in classi di equivalenza. Ad esempio tutti gli input di numeri negativi e tutti gli input di numeri positivi. Esistono delle tecniche per scegliere le classi di equivalenza degli input del test.

  Ad esempio, se l'input valido è un range, si creano tre classi di equivalenza corrispondenti ai valori sotto, dentro e sopra il range di valori. Se invece l'input valido è un insieme discreto vengono create due classi di equivalenza corrispondenti ai valori dentro e fuori l'insieme.

  Uno degli svantaggi di queste tecniche è che non vengono testate combinazioni miste di input ma solo quelle interne alle classi di equivalenza.

- **Boundary testing.** È un caso speciale dell'equivalent testing. Si selezionano degli input che sono limite per le classi di equivalenza (es. per i numeri positivi si testa il numero 1 o il più grande numero positivo rappresentabile).

Il whitebox testing permette invece di analizzare le strutture dati interne e la copertura del codice testato e può essere suddiviso in quattro sottotipi:

- **Statement testing.** Vengono testati i singoli statement del codice.
- **Loop testing.** Vengono dati vari input in modo da effettuare ciascuna delle seguenti operazioni: saltare un ciclo, entrare in un ciclo una sola volta e ripetere un ciclo più volte.
- **Path testing.** Viene costruito un diagramma di flusso del programma e si controlla se tutti i blocchi del diagramma vengono attraversati.
- **Branch testing.** Ci si assicura che vengano percorsi, dai test, tutti i rami del programma almeno una volta (ogni arco del grafo di flusso deve appartenere almeno ad un cammino di esecuzione esercitato dal test).

2.2.2 Integration testing

Il testing di integrazione rileva bug che non sono stati individuati durante l'unit testing. Nel momento in cui i bug nelle componenti sono stati rilevati e corretti, le singole unità sono pronte per essere integrate in sottoinsiemi più grandi per poi essere analizzati nel loro insieme.

Sicuramente l'ordine in cui le componenti sono integrate può influenzare lo sforzo richiesto per l'integrazione.

Esistono varie strategie che decidono in che modo vengono scelti i sottoinsiemi di unità.

- **Big bang testing.** Le componenti sono prima testate individualmente e poi testate insieme come un singolo sistema. Sebbene sia semplice da realizzare è costoso: se un test scopre un fallimento è impossibile distinguere i fallimenti nelle interfacce dai fallimenti all'interno delle componenti.
- **Bottom-up testing.** Con questa strategia vengono prima collaudati i moduli di più basso livello nella gerarchia prodotta dalla progettazione, (e cioè unità più piccole ed elementari che compongono il programma).

  Quando questi moduli sono valutati correttamente si passa al livello superiore che viene testato utilizzando le funzionalità del precedente livello, si risale quindi così fino al sistema intero.

  Si tratta di un approccio che comporta alcune complicazioni: generalmente sono necessari dei **driver**, cioè del software ausiliario che simula la chiamata ad un modulo generando dati da passare come parametri. I driver fanno le veci della parte di codice non ancora integrata

nella porzione testata del programma: la parte alta della gerarchia dei moduli.

Sicuramente il vantaggio di questa modalità è che il processo di test ed integrazione delle unità è intuitivamente più semplice da realizzare ma più tardi viene scoperto un errore, più è costoso eliminarlo, in quanto richiede la sua correzione e la ripetizione parziale del test fino ai moduli nei quali è stato trovato l'errore.

- **Top-down testing.** Questo approccio si comporta in modo diametralmente opposto rispetto alla modalità bottom-up. Inizialmente si testa il modulo corrispondente alla radice, senza utilizzare gli altri moduli del sistema. Per fare ciò vengono utilizzati dei simulatori, detti **stub**, si tratta di elementi che possono restituire risultati casuali e richiedere i dati al collaudatore.

  Dopo aver testato il modulo radice della gerarchia si integrano i suoi figli diretti simulando i loro discendenti per mezzo di stub. Si prosegue in questo modo fino a quando non vengono integrate tutte le foglie della gerarchia.

  Il vantaggio principale è che vengono scoperti prima gli errori più costosi da eliminare.

  Ciò risulta particolarmente utile se la progettazione è anch'essa di tipo top-down e le fasi di progettazione, realizzazione e test sono parzialmente sovrapposte: un errore di progetto nella parte alta della gerarchia può essere scoperto e corretto prima che venga progettata la parte bassa, in ogni istante è disponibile un sistema incompleto ma funzionante ma gli stub possono essere costosi da realizzare.

- **Sandwich testing.** Strategia che combina la strategia top-down con quella bottom-up.

  Il sistema è visto come se avesse tre layers: un layer target, un layer sopra il target, un layer sotto il target. In questo modo si può effettuare il testing top-down e bottom-up in parallelo con lo scopo di arrivare ad integrare il target level.

  Un problema di questo testing è che le componenti non vengono testate separatamente prima di integrarle. Esiste infatti una modifica a tale tecnica di testing che testa le singole componenti prima di integrarle.

2.2.3 System testing

Verifica la corretta esecuzione dell'intera applicazione, incluse le interfacce con altre applicazioni. Le attivà di questa fase sono le seguenti:

- **Function testing.** Verifica che tutte le funzioni siano correttamente completate in uno scenario simile a quello degli utenti finali. Le funzioni non vengono eseguite singolarmente, ma in una sequenza di operazioni che completi un tipico task dell'utente finale.
- **Performance testing.** Verifica le prestazioni del sistema, come ad esempio i tempi di risposta e utilizzo delle risorse. Il test è eseguito quando le caratteristiche relative alle prestazioni costituiscono un fattore critico per il successo del prodotto.
- **Pilot testing.** Durante questo testing il sistema viene installato e fatto usare da una selezionata nicchia di utenti. Successivamente gli utenti vengono invitati a dare il loro feedback agli sviluppatori.

2.2.4 Acceptance testing

Nel Test di accettazione il software viene confrontato con i *requisiti dell'utente finale* ed è un test normalmente svolto dal cliente. Una caratteristica del test di accettazione è che viene usualmente svolto senza avere a disposizione il codice sorgente. Per prodotti di largo consumo si utilizzano i concetti di α-test e β-test.

Un *α-test* è un test fatto nell'ambiente di sviluppo. Un *β-test* è un test fatto nell'ambiente di utilizzo.

Nell'*α-test* il software viene già usato all'interno della casa produttrice per verificarne le funzionalità, però ancora non è stato rilasciato all'esterno.

Nel *β-test* il software viene rilasciato a un numero selezionato di utenti che lo usano sapendo che non è una versione stabile, e che possono interagire con chi ha prodotto il software.

2.2.5 Installation testing

Dopo che il sistema è stato accettato viene installato nel suo ambiente. In molti casi il test di installazione ripete i *test case* eseguiti durante il function testing e il performance testing. Quando il cliente è soddisfatto, il sistema viene formalmente rilasciato, ed è pronto per l'uso.

2.3 Progettazione dei test case

La progettazione dei test case è una parte del testing del sistema e dei componenti in cui si progettano gli input e gli output attesi.

L'obiettivo del processo è la creazione di un insieme di test efficace nella scoperta dei difetti del programma e in grado di mostrare che il sistema soddisfa i suoi requisiti.

Per progettare un test case si seleziona una funzione del sistema o del componente che si sta testando e un insieme di input per eseguire quella funzione, si documentano gli output attesi o i loro campi di variazione e, dove applicabile, si progettano controlli automatici che verifichino che gli output previsti e reali siano gli stessi.

Esistono diversi approcci per progettare i test case:

- **test basati su requisiti:** i test case sono progettati per testare i requisiti del sistema. Viene utilizzato per lo più nella fase di progettazione del sistema poichè i requisiti di solito sono implementati da diversi componenti; per ogni requisito si identificano test case che possono dimostrare che il sistema lo soddisfa.

- **test di partizione:** si identificano le partizioni di input e output e si progettano i test in modo che il sistema esegua tutti gli input di tutte le partizioni e generi tutti gli output in tutte le partizioni. Le partizioni sono gruppi di dati che hanno caratteristiche comuni, come tutti i numeri negativi, tutti i nomi inferiori a 30 caratteri, tutti gli eventi che derivano dalla scelta di oggetti in un menù e così via.

- **test stutturale:** si utilizza la conoscenza della struttura del programma per progettare test che esercitano tutte le parti del programma. Essenzialmente quando si testa un programma si dovrebbe provare a eseguire ogni istruzione almeno una volta. Il test strutturale aiuta a identificare i test case che possono rendere possibile questo.

Vedremo esempi di test case nel progetto realizzato.

2.4 Strumenti di supporto al testing

È ben noto come la validazione e la verifica di un programma rappresentino una fase molto importante nel ciclo di sviluppo di qualsiasi prodotto software. È stato più volte stimato come il costo di tale fase si aggira facilmente attorno al 50% dell'intero costo di sviluppo del prodotto.

Per cercare di aumentare l'efficacia, tagliare i costi e ridurre i tempi di sviluppo, sarebbe estremamente utile poter disporre di tool in grado di generare automaticamente grandi quantità di casi di test, a partire dal codice o dalle specifiche del programa sotto osservazione.

Purtroppo però il numero di tool a disposizione per generare automaticamente casi di test è alquanto esiguo e la maggior parte sono a pagamento.

Tra quelli open-source, l'unico degno di nota è JDocletUnit che però si limita a creare lo scheletro dei casi di test di JUnit per una data classe da testare.

Più evoluto è invece CodeProTools il prodotto fornito (a pagamento) da Instantiations che, sfruttando tecniche di esecuzione simbolica genera una serie di casi di test cercando di massimizzare la copertura (che viene rilevata da un apposito tool fornito nel pacchetto).

Purtroppo, risente dei limiti degli approcci statici, in quanto non offre alcuno strumento per gestire chiamate a funzioni o puntatori ed inoltre richiede comunque al programmatore di inserire manualmente i risultati attesi per ciascun caso di test.

Decisamente più interessante, sebbene ancora prematuro, è UTJML. La sua peculiarità è quella di combinare JML e JUnit per la generazione e la valutazione dei casi di test.

Infatti, i casi di test, dopo essere stati generati, vengono eseguiti con JUnit e verificati tramite un ″oracolo″ ricavato (automaticamente) dalle asserzioni JML.

Al momento la generazione di casi di test è basata su tecniche random, ma l'obiettivo è di utilizzare gli algoritmi genetici per ricavare casi di test efficienti che vengono poi controllati tramite JML.

In generale, per ottenere dal testing i benefici sperati, è richiesto un rigoroso senso di disciplina durante tutto il processo di sviluppo. È essenziale mantenere traccia non solo dei test che sono stati sviluppati ed eseguiti, ma anche di tutte le modifiche effettuate al codice funzionale dell'unità in esame.

Per progetti molto grandi e complessi il ruolo di gestore di test è ricoperto da un manager (Test Manager). In progetti meno complessi e di dimensioni minori il ruolo è ricoperto da un coordinatore (Test Team Leader).

Il responsabile dei test coordina tutte le attività di testing (pianificazione, progettazione e preparazione, esecuzione, monitoraggio e reporting) all'interno del progetto; inoltre è responsabile di completare le attività di testing pianificate secondo i tempi ed i costi previsti e raggiungendo gli obiettivi di

qualità attesi.

Un aspetto di vitale importanza della fase di testing sono i costi sostenuti dall'azienda per effettuarlo. Il costo delle attività di test è direttamente legato al livello di rischio identificato per il progetto. Una delle cause di difficoltà di molti progetti è la sottostima del dimensionamento delle attività di test. La maggior parte dei progetti non pianifica più del 10-15%, salvo poi eseguire una serie di attività non pianificate che portano il progetto a superare il budget.

Capitolo 3

JUnit

JUnit è uno dei più famosi framework open-source per i test unitari, creato da Kent Beck ed Erich Gamma nel 1997. Il successo di JUnit è stato tale che la stessa filosofia è andata oltre la sola versione Java. La famiglia di framework che comprede tutti i diversi porting di JUnit si chiama xUnit.

3.1 Motivazioni

Un'alternativa all'uso dei test unitari presentati nel capitolo precedente è sicuramente l'inserimento nel codice da testare di comandi il cui unico scopo è di effettuare debugging.

La soluzione più semplice per questo scopo è sicuramente l'inserimento di comandi tipo `System.out.println` atti a stampare variabili critiche per poterle analizzare.

Questa soluzione presenta però almeno due grossi inconvenienti: è necessario andare a modificare il codice sorgente ogniqualvolta si voglia tracciare una nuova variabile; è necessario analizzare l'output alla ricerca di eventuali errori manualmente e ad ogni esecuzione. Inoltre va sottolineato che per distribuire il progetto è necessario eliminare tutti i comandi inseriti nel codice per il testing e, soprattutto, qualora si debba apportare ulteriori modifiche ai sorgenti e si voglia testarli nuovamente è necessario reinserire tutti i comandi precedentemente cancellati.

Se per programmi brevi e semplici l'inserimento di codice ad-hoc può anche essere una soluzione valida, sicuramente per progetti di grosse dimensioni diviene difficile e laboriosa.

L'utilizzo di uno strumento per il test unitario, come JUnit, invece, permette una maggiore strutturazione dei test, rende molto più semplice

l'inclusione di nuovi test o l'eliminazione di altri e soprattutto permette un'immediata lettura dei risultati. Infatti, come vedremo, la risposta dopo l'esecuzione di una serie di test in caso di successo è un semplice OK.

Questo velocizza l'esecuzione e la verifica dei test. Inoltre, strutturando correttamente le classi adibite al testing, sarà possibile distribuire il codice creato privato del codice di testing senza apportare alcuna modifica e, se in un secondo momento si dovranno ripetere i test o aggiungerne altri, non sarà necessario riscriverli ma semplicemente rieseguirli.

3.2 Test di prova

Per capire meglio il funzionamento di JUnit consideriamo la classe qui riportata, `Operazioni_Base`, che contiene tra l'altro un semplice metodo `Add` che prende due argomenti interi e ne restituisce la somma e proponiamoci di effettuare il testing tramite JUnit (consideriamo che la classe `Operazioni_Base` sia contenuta all'interno di un package `Math`).

```
package Math;

public class Operazioni_Base {
static public int Add(int a, int b){
        return a+b;
}
static public int Sub(int a, int b){
        return a-b;
}
static public int Mul(int a, int b){
        return a*b;
}
static public int Div(int a, int b){
        return a/b;
}
}
```

Come possiamo notare si sta applicando una strategia *bottom-up* in quanto si sta collaudando una piccola unità (la classe `Operazioni_Base`) che comporrà un programma più ampio.

Ora è necessario creare una classe preposta all'esecuzione dei test (la classe `TestOperazioni_Base`, all'interno del package `Math_Test`).

```
public class TestOperazioni_Base extends TestCase{
```

Tale classe estende la superclasse `TestCase` definita all'interno del framework di JUnit e quindi è anche necessario effettuare l'import del framework stesso: `import junit.framework.TestCase`.
Inoltre è necessario importare tutte le classi di cui si vuole fare il test; nel nostro esempio: `import Math.Operazioni_Base`.

A questo punto ci si focalizza sul corpo vero e proprio del test da creare.

Per prima cosa vanno creati gli eventuali oggetti che saranno utilizzati durante il test. Queste inizializzazioni vanno eseguite all'interno del metodo `setUp()`, definito dalla classe `TestCase`, la cui implementazione va inserita nella classe `TestOperazioni_Base`.

```
@Before
public void setUp() throws Exception {
    }
```

L'unico scopo di tale metodo è quindi di inizializzare tutte le variabili che serviranno per il testing. Nel nostro semplice esempio è un metodo con corpo vuoto.

Esiste anche un metodo analogo che ha l'obiettivo di chiudere tutte le connessioni aperte e distruggere tutti gli oggetti eventualmente creati per il testing. Tale metodo si chiama `tearDown()` e la sua implementazione va inserita nella classe `TestOperazioni_Base`.
Nel nostro esempio anche il corpo di questo metodo sarà vuoto.

```
@After
    public void tearDown() throws Exception {
    }
```

Ora non rimane che definire il **test** vero e proprio, definendo un metodo di nome `testAdd` all'interno della classe `TestOperazioni_Base`.

```
@Test
    public void testAdd() {
              int num1 = 4;
              int num2 = 2;
              int sum = 0;
              sum = Operazioni_Base.Add(num1, num2);
              assertEquals(sum, 6);
}
```

È sicuramente importante notare che, per convenzione, il nome del metodo inizia con la stringa **test** seguito dal nome del metodo da testare.

Analizzando le operazioni svolte si nota la chiamata al metodo `assertEquals`. Tale metodo è definito sempre nel framework JUnit e verifica se i due argomenti che riceve sono *uguali*.
Il concetto di uguaglianza è però ampio; infatti nel caso si testi l'uguaglianza di due valori verrà verificata l'uguaglianza stretta (a==b), nel caso invece si testi l'uguaglianza di due oggetti verrà verificata la loro uguaglianza così come definita dalla `equals()`.

Nel caso in cui l'uguaglianza sia verificata, il test darà esito positivo *(test con successo)*; mentre se l'uguaglianza non è verificata il test darà esito negativo *(test con fallimento)*.

Riportiamo di seguito il codice completo per il testing del nostro esempio di prova:

```
package Math_Test;

import Math.Operazioni_Base;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import junit.framework.TestCase;
public class TestOperazioni_Base extends TestCase{

@Before
   public void setUp() throws Exception {
   }

@After
   public void tearDown() throws Exception {
   }

@Test
    public void testAdd() {
              int num1 = 4;
              int num2 = 2;
              int sum = 0;
              sum = Operazioni_Base.Add(num1, num2);
              assertEquals(sum, 6);
    }
}
```

3.3 Esecuzione del test di prova con successo

JUnit mette a disposizione due modalità per eseguire il test, una testuale e una grafica.

Nella **modalità testuale** è necessario aprire l'editor dei comandi e richiamare la classe *TestRunner* nel seguente modo:

```
java junit.textui.TestRunner Math_Test.TestOperazioni_Base
```

L'output di tale chiamata sarà:

```
.
Time: 0.02

OK (1 test)
```

Facilmente si può capire che il test non ha rilevato nessun errore e che il tempo impiegato è stato di pochi millisecondi.
Il punto rappresentato sta a significare che JUnit ha eseguito un solo test.

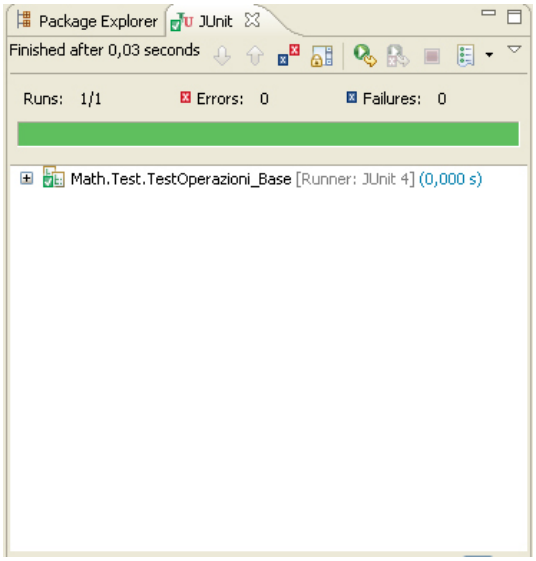

Figura 3.1: Esecuzione grafica con successo

Per eseguire il test in **modalità grafica** invece è sufficiente cliccare sul pulsante *TestRunner*.

In questo modo JUnit esegue il test selezionato e al termine si aprirà la finestra mostrata in figura 3.1.

Nella finestra vengono riportati tutti gli stessi dati visti nella modalità testuale. In particolare nella parte alta è presente una **barra verde** che indica la corretta esecuzione di tutti i test e che svolge la stessa funzione dei punti a livello testuale.

Un vantaggio dell'utilizzo di questa modalità è sicuramente la velocità con cui è possibile verificare la correttezza dei test effettuati.

3.4 Esecuzione del test di prova con fallimento

Supponiamo di effettuare una piccola modifica al nostro test in questo modo:

```
@Test
    public void testAdd() {
              int num1 = 4;
              int num2 = 2;
              int sum = 0;
              sum = Operazioni_Base.Add(num1, num2);
              assertEquals(sum, 7);
}
```

Ovviamente il test porta ad un fallimento in quanto sommare 4 a 2 non dà come risultato 7. Procedendo come descritto nel paragrafo precedente eseguiamo il test in modalità grafica.

Si può subito notare (figura 3.2) che il test è fallito e la barra di scorrimento è di colore rosso. Nella parte inferiore si nota un messaggio che riporta il **motivo del fallimento**.

In questo test il framework ha rilevato un fallimento *(failure)* cioè il test di uguaglianza da noi impostato non è verificato, ovvero una o più delle asserzioni specificate risulta falsa.

Nel caso in cui si verifichi un'eccezione imprevista durante l'esecuzione del test, JUnit termina segnalando un errore *(error)*, ovvero evidenzia il verificarsi di un'eccezione non gestita. Se invece l'eccezione è una di quelle previste allora è opportuno che il metodo di test preveda la gestione.

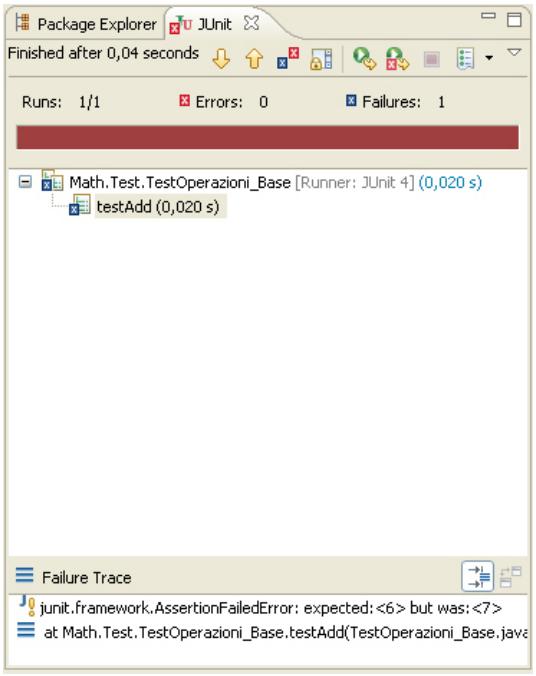

Figura 3.2: Esecuzione grafica con fallimento

3.5 La gestione delle eccezioni

Proviamo a modificare il codice del semplice metodo di prova preso in considerazione nei paragrafi precedenti (metodo `Add`), in modo tale da sollevare un'eccezione.

```
package Math;

public class Operazioni_Base {
     static public int Add(int a, int b){
               if (a>=0 && b>=0){
                    return a + b;
                }
                else
                    throw new NumberFormatException();
     }

[...]

}
```

Se vengono passati dei valori negativi al metodo `Add` verrà generata un'eccezione di tipo `NumberFormatException`.

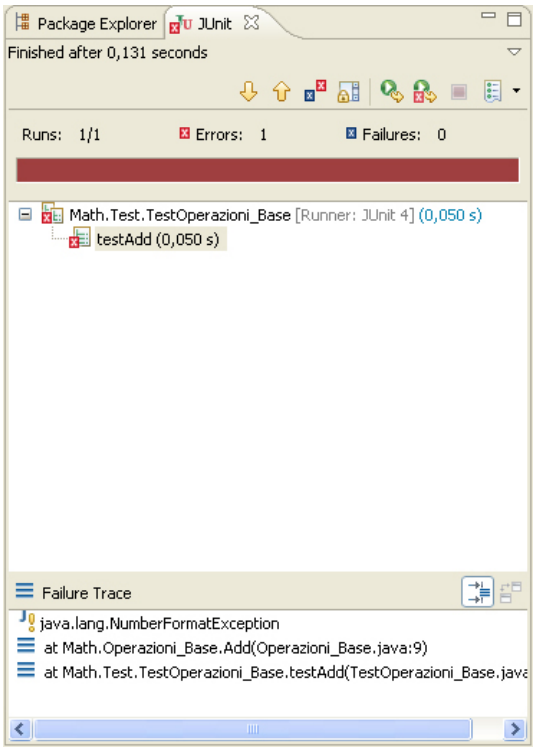

Figura 3.3: Esecuzione grafica con eccezione

Se modifichiamo il metodo `testAdd` in modo che richiami la `Add` con valori negativi senza aggiungere alcuna gestione dell'eccezione, JUnit segnala la generazione di una eccezione imprevista avvertendo la presenza di un *error* (Figura 3.3).

Leggendo nella parte inferiore si nota il nome dell'eccezione sollevata e il punto in cui è stata generata, sia nel metodo principale (`Add`), sia nel metodo di test (`testAdd`).

Per gestire al meglio l'eccezione è disponibile in JUnit il metodo `fail(string)` il quale forza il fallimento del test proponendo in output la stringa passatagli come argomento.

Vediamo quindi come deve essere modificata la porzione di codice:

```
@Test
    public void testAdd() {
```

```
                int num1 = -4;
                int num2 = 2;
                int sum = 0;
                try{
                    sum = Operazioni_Base.Add(num1, num2);
                    fail("Eccezione non sollevata!!!");
                     }
                 catch(NumberFormatException e){}
}
```

In questo metodo, viene invocato il metodo `Add` con valori negativi. A differenza di quanto fatto in precedenza, in questo caso si è racchiusa la chiamata di tale metodo all'interno di un blocco `try/catch` al fine di catturare l'eccezione generata.

Da notare che dopo la creazione dell'oggetto è stata aggiunta la chiamata: `fail(″L'eccezione non è stata sollevata!!!″)`. Tale chiamata è necessaria nel caso in cui il metodo `Add` non sollevi un'eccezione di tipo `NumberFormatException` se gli vengono passati argomenti negativi.

Infatti se l'eccezione non venisse sollevata il metodo non funzionerebbe correttamente e sarebbe necessario far fallire il test, quindi grazie al metodo `fail()` se ciò accadesse verrebbe visualizzato, nella parte bassa della finestra, un messaggio d'errore.

3.6 Test Suite

Di solito nei progetti si hanno decine di classi di test e può essere molto comodo eseguirle con un solo click dal nostro IDE (ad esempio *Eclipse*).

A tale scopo JUnit mette a disposizione la classe `TestSuite` che consente di invocare in modo automatico tutti i metodi di ogni classe che estende `TestCase`.

Eclipse permette di creare la classe `TestSuite` in modo semplice: ci si posiziona nel package desiderato, si sceglie dal menù *New JUnit Test Suite* e nella finestra di dialogo che viene visualizzata si sceglie il nome della classe (di default di solito è `AllTests`) e quindi si scelgono le classi da includere nella suite.

Al termine di questa operazione verrà generato automaticamente il seguente codice:

```
package Math_Test;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
    public static Test suite() {
       TestSuite suite = new TestSuite(AllTests.class.getName());
        //$JUnit-BEGIN$
        suite.addTestSuite(TestOperazioni\Base.class);
        //eventuali altre classi da testare come ad esempio:
        // TestOperazioni_Logaritmiche.class
        // TestOperazioni_Trigonometriche.class
        //etc.
        //$JUnit-END$
        return suite;
    }
}
```

Come prima cosa viene creato un oggetto vuoto di nome `suite` e tipo `TestSuite` a cui verranno aggiunti tramite il metodo `addTestSuite` tutte le classi test che si vorranno testare.

Quando verrà eseguita questa classe, i metodi test verranno eseguiti senza un ordine preciso, quindi non necessariamente vengono eseguiti nell'ordine definito. Nel caso in cui ci siano degli errori, nella finestra grafica apparirà una barra di colore rosso e nella parte sottostante un messaggio ci indicherà in che classe e in che punto trovare l'errore.

3.7 Asserzioni di JUnit

Un metodo di test può contenere una o più invocazioni di metodi **assert**.

I metodi assert sono metodi statici contenuti nella classe *org.junit.Assert* che effettuano una semplice comparazione tra il risultato atteso ed il risultato dell'esecuzione.
Un test fallisce se fallisce almeno uno dei metodi assert contenuti in esso; viceversa il test ha successo se *tutti* i metodi assert sono eseguiti con successo.

Il metodo assert più utilizzato è sicuramente
assertEquals() (valore previsto, valore effettivo)

il quale ha il compito di verificare l'uguaglianza tra il valore effettivo e il valore previsto.

È possibile utilizzare tale assert anche in altre due versioni:

- **assertEquals(messaggio, valore previsto, valore effettivo)** e
- **assertEquals(valore previsto, valore effettivo, delta)**.

Il primo oltre a controllare l'uguaglianza visualizza un messaggio di testo nel caso in cui il test fallisca; il secondo invece è utilizzato per i valori float e double, li confronta tenendo conto di un possibile delta di differenza.

Possono essere utilizzati anche altri tipi di assert come ad esempio:

- **assertFalse(condizione):** controlla che la condizione sia falsa;
- **assertFalse(messaggio, condizione):** controlla che la condizione sia falsa e in caso contrario visualizza un messaggio d'errore;
- **assertTrue(condizione):** controlla che la condizione sia vera;
- **assertTrue(messaggio, condizione):** controlla che la condizione siavera e in caso contrario visualizza un messaggio d'errore;
- **assertNotNull(oggetto):** controlla che l'oggetto non sia null;
- **assertNotNull(messaggio, oggetto):** controlla che l'oggetto non sia null e in caso contrario visualizza un messaggio d'errore;
- **assertNull(oggetto):** controlla che l'oggetto sia null;
- **assertNull(messaggio, oggetto):** controlla che l'oggetto sia null e in caso contrario visualizza un messaggio d'errore;
- **assertSame(valore previsto, valore effettivo):** controlla che i due oggetti si riferiscano allo stesso oggetto;
- **assertSame(messaggio,valore previsto, valore effettivo):** controlla che i due oggetti si riferiscano allo stesso oggetto e in caso contrario visualizza un messaggio d'errore;
- **assertNotSame(valore previsto, valore effettivo):** controlla che i due oggetti non si riferiscano allo stesso oggetto;
- **assertNotSame(messaggio,valore previsto, valore effettivo):** controlla che i due oggetti non si riferiscano allo stesso oggetto e in caso contrario visualizza un messaggio d'errore.

Oltre ai metodi assert esistono anche i seguenti metodi:

- **fail():** interrompe l'esecuzione dei test generando una failure;
- **fail(messaggio):** interrompe l'esecuzione dei test generando una failure e riporta in output la stringa passatagli come parametro;
- **failNotEquals(messaggio, valore atteso, valore reale):** controlla che due oggetti siano uguali e se questo non avviene l'esecuzione del test è interrotta visualizzando un messaggio d'errore;
- **failNotSame(messaggio, valore atteso, valore reale):** controlla che due oggetti si riferiscano allo stesso oggetto e se questo non avviene l'esecuzione del test è interrotta visualizzando un messaggio d'errore.

Capitolo 4

JSetL

JSetL è una libreria Java che combina la programmazione OO di Java con i concetti fondamentali del CLP($\mathcal{SET}$) come:

- variabili logiche,
- unificazione,
- strutture dati ricorsive,
- liste e insiemi anche parzialmente specificate,
- unificazione,
- risoluzione di vincoli, in particolare su insiemi,
- non-determinismo.

4.1 Caratteristiche principali

JSetL è pensato principalmente come strumento di supporto alla programmazione dichiarativa. Questo paradigma di programmazione consiste nello specificare che *cosa* il programma deve fare, piuttosto che *come* farlo, tipico invece della programmazione imperativa.

JSetL è stato sviluppato presso il Dipartimento di Matematica dell'Università di Parma. Si tratta di un package completamente scritto in codice Java, importabile con la sola istruzione `import JSetL.*`.

La libreria è un software libero, re-distribuibile e modifcabile sotto i termini della licenza GNU.

La versione corrente è JSetL 2.0.

Vediamo brevemente quali sono le principali caratteristiche della libreria JSetL:

- **Variabili logiche:** hanno lo stesso significato che possiedono nei linguaggi di programmazione logica e funzionale. Possono essere inizializzate o non inizializzate, il loro valore può essere di ogni tipo e può essere determinato come risultato di vincoli che coinvolgono le variabili stesse.
- **Liste e insiemi:** sono strutture dati la cui principale differenza consiste nel fatto che nelle *liste* è importante l'ordine e la ripetizione degli elementi, mentre negli *insiemi* l'ordine e la ripetizione non hanno importanza. Entrambe le strutture dati possono essere parzialmente specificate, cioè possono contenere variabili logiche non inizializzate sia come elementi che come parte della struttura dati.
- **Unificazione:** l'unificazione tra due oggetti può essere usata per testare l'equivalenza tra essi o per assegnare valori alle variabili logiche non inizializzate in essi eventualmente contenute.
- **Vincoli:** uguaglianza, disuguaglianza ed operazioni insiemistiche di base, come differenza, unione, intersezione, sono trattati come vincoli in JSetL. I vincoli vengono in primo luogo aggiunti al *constraint store* e poi risolti tramite il *constraint solver*.
- **Non determinismo:** i vincoli in JSetL vengono risolti in maniera non deterministica, sia perchè l'ordine in cui sono risolti non è importante, sia perchè nella loro risoluzione si utilizzano i punti di scelta e il backtracking.

4.2 Definizione e uso di `LVar`, `IntLVar`, `IntLSet` e `LList`

4.2.1 Variabile logica `LVar`

Una **variabile logica** è un oggetto logico che è un'istanza della classe `LVar`, creato in particolare con la dichiarazione:

```
LVar nomeLVar = new LVar(NomeLVarExt, ValoreVar);
```

dove `nomeLVar` è il nome della variabile, `NomeLVarExt` è un nome esterno (opzionale), `ValoreVar` è un valore (opzionale) di inizializzazione della variabile stessa.

Una variabile logica che non ha un valore unico associato con essa è detta **non inizializzata** (o incognita o unbound). Altrimenti la variabile è detta **inizializzata** (bound).

Il valore di una variabile logica può essere specificato al momento della costruzione oppure determinato dalla risoluzione di vincoli che coinvolgono la variabile; in ogni caso in seguito non può essere modificato direttamente con metodi della classe `LVar`.

Oltre ai vincoli, la classe `LVar` fornisce metodi che permettono di stampare l'eventuale valore della variabile logica, di conoscere se la variabile è inizializzata o no, di ottenere il suo nome esterno, e cosi via.

4.2.2 Variabile logica intera `IntLVar`

Una **variabile logica intera** è un caso particolare della variabile logica precedentemente descritta, nella quale i valori si limitano a essere numeri interi. Inoltre una variabile logica intera può disporre di un dominio finito e un vincolo aritmetico ad essa associato.

Può essere creata in particolare con la seguente dichiarazione:

```
IntLVar nomeIntLVar = new IntLVar(NomeVarExt, Valore_a,
                      Valore_b);
```

dove `nomeIntLVar` è il nome della variabile, `NomeVarExt` è un nome esterno (opzionale), `Valore_a` e `Valore_b` sono gli estremi inferiore e superiore del dominio della variabile.

Se $b<a$ allora è sollevata l'eccezione `NotValidDomainException`. Questi due valori sono opzionali e dipendono dal tipo di dominio della variabile.

Il dominio è rappresentato, nel caso generale, come l'unione di n (n>1) intervalli disgiunti $[l_i,h_i]$, $1 \leq i \leq n$, dove l_i e h_i sono rispettivamente il limite inferiore e il limite superiore dell'intervallo i-esimo, con $l \leq h$.

Un dominio per una variabile logica intera può essere specificato quando la variabile è creata ed è aggiornato automaticamente quando i vincoli sono risolti, in modo da mantenere la consistenza del vincolo.

Ad esempio se `x` e `y` sono due `IntLVar` con dominio [1,10] e si ha il constraint `x>y` allora il dominio di `x` è aggiornato a [2,10] e il dominio di `y` è aggiornato a [1,9].

Quando il dominio di una variabile è limitato ad un unico valore allora la variabile è legata a questo valore; viceversa se il dominio si riduce ad un

insieme vuoto, significa che i vincoli che coinvolgono le variabili non sono soddisfatti.

I vincoli aritmetici associati alle `IntLVar` sono rappresentati come congiunzione di vincoli atomici. I vincoli aritmetici sono generati attraverso una valutazione di espressioni logiche intere cioè espressioni che utilizzano i soliti operatori aritmetici quali `sum, sub, mul, div` e `mod` applicati a variabili logiche intere o a numeri interi. Per esempio se

```
x.sum(y.sub(1))
```

dove `x` e `y` sono `IntLVar`, la valutazione di questa espressione restituisce una nuova variabile logica intera X_1 a cui è associato il vincolo aritmetico

$$X_1\texttt{= x+}X_2 \wedge X_2\texttt{=y-1}$$

4.2.3 Insieme logico `LSet`

Un **insieme logico** è un oggetto logico che è un'istanza della classe `LSet`, creato dalla dichiarazione:

```
LSet nomeLSet = new Set(NomeLSetExt);
```

dove `nomeLSet` è il nome dell'insieme, `NomeLSetExt` è un nome esterno (opzionale).

L'insieme vuoto è creato con `LSet.empty()`. Gli insiemi possono essere **inizializzati** con elementi di qualsiasi tipo o **non inizializzati**.
Il valore di un insieme può essere specificato elencando i suoi elementi tramite i metodi `ins` e `insAll`.

Un insieme che contiene alcuni elementi che non sono inizializzati è detto **insieme parzialmente specificato**.

Un insieme è detto **illimitato** se contiene un certo numero di elementi (noti o meno) $e_1.....e_n$ e un resto r, rappresentato da un insieme non inizializzato.

La notazione astratta usata usata in questo caso è la seguente:

$$\{e_1.....e_n \ |\text{r}\}$$

Un insieme illimitato viene costruito tramite i metodi di inserimento `ins` e `insAll` applicati a un insieme non inizializzato.

4.2.4 Insieme logico intero `IntLSet`

Un **insieme logico intero** è un particolare caso dell'insieme logico precedentemente descritto, nel quale i valori degli elementi si limitano a essere numeri interi. È creato dalla dichiarazione:

```
IntLSet nomeIntLSet = new IntSet(NomeIntLSetExt, Valore_1,
                                 Valore_2);
```

dove `nomeIntLSet` è il nome dell'insieme logico intero, `NomeIntLSetExt` è un nome esterno (opzionale), `Valore_1` e `Valore_2` sono due numeri interi che indicano gli elementi contenuti nell'insieme, (opzionali).

4.2.5 Lista logica `LList`

Una **lista logica** è un oggetto logico che è un'istanza della classe `LList`, creato con la dichiarazione

```
LList nomeLList = new LList(nomeExt);
```

dove `nomeLList` è il nome della lista, `nomeExt` (opzionale) è il nome esterno della lista.

La **lista vuota** è creata con il metodo `LList.empty()` e verrà indicata con $[\,]$.
Se una lista contiene n elementi $c_1, \ldots, c_n$ verrà indicata con $[c_1, \ldots, c_n]$.
Inoltre, $[c_1, \ldots, c_n|r]$ con r lista, denota la **concatenazione** delle liste $[c_1, \ldots, c_n]$ ed r.

Se in particolare r è unknown, $[c_1, \ldots, c_n|r]$ è una **lista illimitata (unbounded)** in quanto i primi n elementi $c_1, \ldots, c_n$ sono noti, ma gli elementi di r non lo sono (si noti che potrebbe essere anche che $r = [\,]$).

4.2.6 Esempi di utilizzo

Per meglio comprendere questa e le precedenti stutture dati vengono riportati alcuni esempi:

Esempi di dichiarazione e utilizzo degli oggetti logici:

1. `LVar x = new LVar();`
2. `LVar y = new LVar("vary",'a');`
3. `LVar z = new LVar(x);`

4. `LList l = new LList("l");`

5. `LList m = LList.empty().ins(3).ins(2);`

6. `IntLVar n = new IntLVar("n", 2,4);`

7. `IntLSet s = new IntLSet(2,5);`

8. ```
   Object[] v = {1,x};
   LList o = LList.empty().insAll(v);
   ```

Vediamo ora di commentare queste istruzioni:

1. `x` è una variabile logica non inizializzata e senza nome esterno.

2. `y` è una variabile logica inizializzata con il carattere 'a' e nome esterno `vary`.

3. `z` è una variabile logica inizializzata senza nome esterno che ha come valore associato la variabile logica `x`.

4. `l` è una lista logica non inizializzata con nome esterno `l`.

5. `m` è una lista logica inizializzata senza nome esterno, i valori sono [3,2].

6. `n` è una variabile logica intera con nome esterno `n` e dominio compreso nell'intervallo [2,4].

7. `s` è un insieme logico intero senza nome esterno i cui valori sono {2,3,4,5}.

8. `o` è una lista logica parzialmente specificata ma limitata [1,x] (ha 2 elementi).

4.3 I vincoli in `JSetL`

I vincoli in JSetL sono particolari condizioni su `LVar` e `LSet` gestiti da un **Constraint Solver** (o risolutore di vincoli, di fatto un'istanza della classe di JSetL `SolverClass`) che ne implementa la strategia di risoluzione.

Tali vincoli sono passati al constraint solver S mediante l'inserimento in un **Constraint Store**, che contiene la collezione corrente Γ dei vincoli attivi in S.

La risoluzione di Γ è effettuata mediante la chiamata di uno dei **metodi di constraint solving** forniti dal solver, come ad esempio `solve()`. Il solver
   ```

ricerca (in modo non-deterministico) una *soluzione* che soddisfi tutti i vincoli di Γ, riscrivendoli se possibile in una **forma semplificata**.

4.3.1 Definizione di vincoli

Formalmente, un vincolo è una particolare *relazione su un certo dominio* $\mathcal{D}$. Un **vincolo atomico** in JSetL è un'espressione di una delle seguenti forme:

- `x.op();`
- `x.op(y);`
- `x.op(y,z);`

dove `op` è uno dei **metodi predefiniti** per la gestione dei vincoli (`eq`, `neq`, `in`, `nin`, `lt`, `le`, `gt`, `ge`, ecc...) e `x`,`y`,`z` sono espressioni che dipendono dal vincolo `op`.

Il significato di questi metodi è quello intuitivamente suggerito dal loro nome: `eq` e `neq` sono vincoli di uguaglianza e disuguaglianza; `in` e `nin` sono vincoli di appartenenza e non appartenenza insiemistica; `lt`, `le`, `gt`, `ge` sono vincoli su interi corrispondenti rispettivamente alle relazioni $<, \leq, >, \geq$ e cosi via.

Un **vincolo composto** in JSetL è una congiunzione o disgiunzione di uno o più vincoli atomici, cioè un'espressione delle forme:

- c_1`.and(`c_2`)....and(`c_n`)`,

oppure

- c_1`.or(`c_2`)....or(`c_n`)`.

dove $c_1, c_2, \ldots, c_n$ sono vincoli atomici o composti e n$\geq$0.

Il significato di c_1`.and(`c_2`)....and(`c_n`)` è la **congiunzione logica** $\hat{c_1} \wedge \hat{c_2} \wedge \ldots \wedge \hat{c_n}$ dove $\hat{c_1}, \ldots, \hat{c_n}$ rappresentano il ″significato logico″ delle $c_1, \ldots, c_n$ espressioni (ad esempio, se c è il vincolo `X.subset(Y)`, allora $\hat{c}$ è X $\subseteq$ Y). Analogo il significato di c_1`.and(`c_2`)....and(`c_n`)` che è la disgiunzione logica $\hat{c_1} \vee \hat{c_2} \vee \ldots \vee \hat{c_n}$.

Un vincolo in JSetL è un vincolo atomico o un vincolo comporto ed è realizzato con un'istanza della classe `Constraint`. Vediamo ora alcuni esempi di vincoli in JSetL.

Esempio
Siano `x,y,z` tre `LVar`, `r,s,t` tre `LSet` e `n` un intero

1. `r.eq(s);`

2. `x.neq(y.sum(n)).and(x.eq(3)).and(y.neq(z));`

3. `z.ge(-6);`

Il vincolo 1 rappresenta l'uguaglianza insiemistica `r=s`.
Si noti che ciò è ben diverso da un assegnamento ed implica il concetto di **unificazione** tra r ed s. Se r e s sono LSet questo significa che il vincolo atomico `r=s` impone la ricerca di un assegnamento agli elementi di r ed s tale che i due insiemi risultino uguali secondo la teoria degli insiemi.
In generale, un problema di unificazione insiemistica può ammettere più di una soluzione: ad esempio, il vincolo {x,y}={z,2} con x,y,z variabili logiche unknown risulta soddisfatto con x=z $\wedge$ y=2 ma anche con y=z $\wedge$ x=2.

La gestione di queste soluzione multiple è affidata, al constraint solver.

L'esempio 3 rappresenta il vincolo $x \neq (y + n) \wedge x = 3 \wedge y \neq z$.

L'esempio 4 corrisponde al vincolo atomico $z \geq -6$.

4.3.2 Constraint store e constraint solving

Il **constraint store** (o più sinteticamente *c.store*) di un constraint solver S contiene la collezione di tutti i vincoli attualmente attivi in S.

JSetL fornisce metodi attraverso i quali è possibile aggiungere nuovi vincoli al constraint store, visualizzarne il contenuto, rimuovere tutti i vincoli in esso presenti, ecc...

Se il c.store contiene n vincoli (atomici e non) $c_1, c_2, \ldots, c_n$ allora indicheremo con Γ il vincolo corrispondente alla congiunzione dei vincoli $c_1, \ldots, c_n$, cioè $\Gamma = c_1 \wedge c_2 \wedge \ldots \wedge c_n$.

L'**inserimento** di un nuovo vincolo `C` nel c.store effettuato con la chiamata al metodo `and` della classe `SolverClass`: l'invocazione

```
S.add(C)
```

comporta l'aggiunta del constraint `C` al c.store del contraint solver `S`.

Il significato di tale istruzione è l'aggiornamento del vincolo Γ, cio $\Gamma \leftarrow \Gamma \wedge$ `C`.

A tale metodo può essere passato un oggetto di tipo `Constraint`, cioè un vincolo singolo, oppure un oggetto di tipo `ConstraintsConjuction`, cioè

una congiunzione di vincoli. Nel primo caso il metodo aggiunge il vincolo passato come paramentro in cosa allo store, nel secondo caso aggiunge in cosa tutti i vincoli della congiunzione.

Una volta aggiunti nuovi vincoli al c.store, si può chiedere al solver S di *risolverli* invocando uno degli appositi metodi. La classe `Solver` si occupa dell'introduzione dei vincoli nel constraint store e della loro risoluzione. È costituita da sei attributi i quali sono accessibili solo dalla classe `SolverClass` stessa.

Il contraint store è realizzato mediante l'attributo `store` della classe `ConstraintStore` che estende la classe `ConstrantinsConjunction`, i cui elementi sono riferimenti a istanze della classe `Constraint`.

Il **constraint solving** (o più sinteticamente *c.solving*) in JSetL basato sulla riduzione di ogni vincolo atomico in una forma semplificata, chiamata *solved form*, che è dimostrato essere **soddisfacibile**. Il successo di questa riduzione su tutti i vincoli atomici di Γ permette quindi di concludere che la collezione dei vincoli del c.store sicuramente è **soddisfacibile**. In caso contrario, l'individuazione di un *fallimento* (in termini logici, una riduzione a `false`) implica la non-soddisfacibilità di Γ.

Capitolo 5

Progettazione e realizzazione dei test per JSetL

Nel capitolo seguente verrà illustrata la progettazione e la realizzazione delle classi di test necessarie a svolgere il testing di JSetL.

Si cercherà di creare dei test il più possibile completi ed efficienti, sviluppando una casistica completa sulla creazione e utilizzo degli oggetti trattati in JSetL e prendendo in considerazione tutti i metodi public della maggior parte delle classi che compongono JSetL.

Per quanto riguarda l'implementazione del testing si utilizzerà JUnit.

Per ognuna delle classi di JSetL da testare si realizza una corrispondente classe di test, utilizzando gli strumenti messi a disposizione da JUnit. Le classi di test realizzate sono:

1. `LvarTest`
2. `LSetTest`
3. `LListTest`
4. `IntLSetTest`
5. `IntLvarTest`
6. `ConstraintTest`

A queste si aggiunge una classe `testSuite` chiamata `AllTest` la quale, come abbiamo visto nei capitoli precedenti, permette di eseguire tutti i test creati e contenuti nello stesso package: uno per ogni classe di JSetL presa in considerazione.

Nei paragrafi successivi le classi di JSetL prese in considerazioni, vengono descritte in modo semplice e schematico. Inizialmente viene fatta una veloce carrellata sugli attributi di ogni classe, dopodichè si descrivono i casi presi in considerazione per testare i singoli metodi.

5.1 Test della classe `LVar`

Come già detto precedentemente una `LVar` è una variabile logica che possiede i seguenti attributi:

- **name**: contiene una stringa che corrisponde al nome esterno della `LVar`, se è stato specificato, altrimenti viene restituisce il simbolo `?`;
- **init**: flag che indica l'inizializzazione della variabile. Se è settato a `true` significa che la variabile logica è inizializzata, in caso contrario sarà `false`;
- **val**: contiene un eventuale valore della `LVar` nel caso in cui sia inizializzata oppure `null` in caso contrario.

I casi che prendiamo in considerazione per effettuare i vari test della classe `LVar` sono stati suddivisi in quattro gruppi principali:

1. **Casistica 1:**
    1. una `LVar` con nome.
       Ad esempio:
       `LVar a = new LVar("x")`
    2. una `LVar` senza nome.
       Ad esempio:
       `LVar a = new LVar(1)`
2. **Casistica 2:**
    1. una `LVar` non inizializzata.
       Ad esempio:
       `LVar x = new LVar()`
    2. una `LVar` inizializzata:
        1. con un tipo primitivo e precisamente con un intero.
           Ad esempio:
           `LVar x = new LVar("x",1)`

2. con un oggetto e precisamente con una stringa.
   Ad esempio:
   `String x1 = "Stringa";`
   `LVar x = new LVar("x",x1)`
3. con un'altra `LVar`:
   1. inizializzzata.
      Ad esempio:
      `LVar y = new LVar("y",3);`
      `LVar x = new LVar(y)`
   2. non inizializzata.
      Ad esempio:
      `LVar y = new LVar("y");`
      `LVar x = new LVar(y);`

3. **Casistica 3:**

   1. `LVar` inizializzata.
      Ad esempio:
      `LVar y = new LVar("y",3);`
   2. `LVar` non inizializzata.
      Ad esempio:
      `LVar y = new LVar("y");`
   3. `LVar` clonata da una `LVar` generica.
      Ad esempio:
      `LVar x1 = new LVar("x1",1);`
      `LVar x = x1.clone();`

Utilizziamo la tecnica dell'*equivalence testing* vista nei precedenti capitoli secondo la quale gli input vengono divisi in classi di equivalenza per cui tutti gli altri casi non sono presi in considerazione dato che appartengono già alle classi di equivalenza descritte.

Per testare tutti i metodi presenti nella classe `LVar` si è utilizzata la classe `LVarTest`.

Nella classe **LVarTest** i metodi di `LVar` che vogliamo testarei sono i seguenti:

- **getName()**: ha il compito di restituire una stringa contenente il nome del `LVar` se specificato, in caso contrario restituirà il simbolo `?`.

  I casi che prendiamo in considerazione per testare il corretto funzionamento di questo metodo sono quelli descritti nella **casistica 1**.

Di seguito è riportato un test che controlla il corretto funzionamento del `getName()` per una variabile senza nome come visto nel caso *1.2*:

```
@Test
       public void testGetNameWithNoName() {
                 LVar a = new LVar();
                 assertTrue("?".equals(a.getName()));
}
```

Come si può notare attraverso la `assertTrue` viene confrontata la stringa restituita dal metodo con una stringa che dovrebbe restituire il `getName()` nel caso in cui funzionasse correttamente.

- **setName(String)**: prende come argomento una stringa, la quale sarà il nome esterno della `LVar`.

  Anche in questi test i casi che prendiamo in esame sono quelli descritti nella **casistica 1**.

- **getValue()**: restituisce il valore presente nella `LVar` se è inizializzata altrimenti restituisce `null`.

  I casi che consideriamo per testare il corretto funzionamento di tale metodo sono quelli descritti nella **casistica 2**.

  Vediamo un test che serve a testare il metodo per il caso *2.3.1*:

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

  Come si può notare viene creata una `LVar` inizializzata con una `LVar` inizializzata e si confronta il risultato del metodo `getValue()` con un intero che si suppone essere quello corretto nel caso in cui il metodo funzioni in modo corretto.

- **toString()**: restituisce una stringa contenente il valore della `LVar`, se è stata inizializzata, altrimenti `_nomeLVar` se la variabile logica ha nome oppure `_?` se non lo ha.

I casi che consideriamo per testare il corretto funzionamento di questo metodo sono quelli descritti nella **casistica 2**.

Di seguito è ripostato un test per il caso *2.2.2*:

```
@Test
public void testToStringInitializedWithString() {
          String x1 = "Stringa";
          LVar x = new LVar("x",x1);
          assertEquals(x1,(x.toString()));
}
```

È facile intuire che il metodo `toString()` viene testato su una `LVar` inizializzata tramite una variabile di tipo `String`. Successivamente viene confrontato il risultato del metodo con il `LVar` che si suppone essere l'esito della `toString()` nel caso in cui funzioni con successo.

- **isKnown()**: restituisce un booleano, `true` se la `LVar` è inizializzata, `false` altrimenti. Per effettuare i test vengonoi usate le `assertTrue` o la `assertFalse` in base al booleano restituito dal metodo.

  I casi che prendiamo in considerazione per testare il corretto funzionamento del metodo sono quelli descritti nella **casistica 3**.

  Vediamo un semplice esempio per questo test:

  ```
  @Test
     public void testIsKnownInitialized() {
     LVar x = new LVar("x",1);
     assertTrue(x.isKnown());
     }
  ```

  Come si può notare attraverso la `assertTrue` si controlla che la `LVar` sia effettivamente inizializzata come da definizione.

- **equals(LVar)**, mette a confronto due `LVar` e restituisce `true` se sono uguali oppure `false` in caso contrario.

  I casi che prendiamo in considerazione sono quelli ottenuti combinando tra loro i casi elencati precedentemente nella **casistica 2** per la due `LVar` coinvolte nel metodo.

I casi così ottenuti sono riportati nella tabella seguente:

L_1	L_2	$L_1.equals(_2)$
non iniz x **(2.1)**	non iniz x **(2.1)**	true
	non iniz y **(2.1)**	false
	iniz con intero **(2.2.1)**	false
	iniz con stringa **(2.2.2)**	false
	iniz con LVar iniz **(2.2.3.1)**	false
	iniz con LVar non iniz **(2.2.3.2)**	false
iniz con intero **(2.2.1)**	non iniz **(2.1)**	false
	iniz con intero **(2.2.1)**	true
	iniz con stringa **(2.2.2)**	false
	iniz con LVar iniz **(2.2.3.1)**	true
	iniz con LVar non iniz **(2.2.3.2)**	false
iniz con stringa **(2.2.2)**	non iniz **(2.1)**	false
	iniz con intero**(2.2.1)**	false
	iniz con stringa **(2.2.2)**	true
	iniz con LVar iniz **(2.2.3.1)**	false
	iniz con LVar non iniz **(2.2.3.2)**	false
iniz con LVar iniz **(2.2.3.1)**	non iniz **(2.1)**	false
	iniz con intero **(2.2.1)**	true
	iniz con stringa **(2.2.2)**	false
	LVar iniz con LVar iniz **(2.2.3.1)**	true
	iniz con LVar non iniz **(2.2.3.2)**	false
iniz con LVar non iniz **(2.2.3.2)**	non iniz **(2.1)**	false
	iniz con intero **(2.2.1)**	false
	iniz con stringa **(2.2.2)**	false
	iniz con LVar iniz **(2.2.3.1)**	false
	iniz con LVar non iniz **(2.2.3.2)**	false

La tabella è di facile comprensione ma di seguito vengono fornite alcune spiegazioni in modo da essere più chiari:

- nelle prime due colonne sono specificate le `LVar` chiamate in gioco, mentre nella terza colonna è possibile trovare il risultato del metodo `equals`
- in grassetto per ogni caso è indicato il numero corrispondente alla casistica descritta precedentemente in modo da avere un'idea più limpida
- la `LVar` non inizializzata `x` è diversa dalla `LVar` non inizializzata `y`

– quando si confrontano due `LVar` inizializzate con intero si prende in esame il caso in cui l'intero sia uguale per entrambe.

Ovviamente per effettuare questi test utilizziamo la `assertTrue` o la `assertFalse` in base ai casi ottenuti.

- **equals(Object)**: confronta una `LVar` con un oggetto qualsiasi controllando se sono uguali. Le `LVar` prese in considerazione sono quelle presentate nella **casistica 2** e sono state confrontate con un intero, con un `LSet` inizializzato e non inizializzato.
  Ad esempio:
  ```
  LVar x1 = new LVar("x1",2);
  LSet x = LSet.empty().ins(x1)
  ```

5.2 Test della classe `LSet`

Come già detto un `LSet` è un insieme di elementi di tipo qualsiasi, costituito da quattro attributi:

- **name**: contiene una stringa che corrisponde al nome esterno del `LSet`, se è specificato, un simbolo `?` nel caso in cui non abbia il nome, oppure `_emptyLSet` nel caso in cui `LSet` sia vuoto
- **init**: flag che indica l'inizializzazione del `LSet`. Inizialmente a `false`, diventa `true` se `LSet` è inizializzato
- **lista** e **resto** conterranno gli elementi che possono essere inseriti sia durante la creazione del `LSet` (attraverso i costruttori di default) oppure attraverso i metodi `ins` e `inAll`.

Anche negli `LSet` come negli `LVar` i casi che prendiamo in considerazione per testare il corretto funzionamento dei metodi sono suddivisi in più gruppi.

1. **Casistica 1:**
   1. `LSet` con nome.
      Ad esempio:
      ```
      LSet x = LSet.empty().ins(1).setName("x")
      ```
   2. `LSet` senza nome.
      Ad esempio:
      ```
      LSet x = new LSet()
      ```

3. `LSet` vuoto.
   Ad esempio:
   `LSet x = LSet.empty()`

2. **Casistica 2:**

   1. `LSet` vuoto.
   2. `LSet` non inizializzato.
      Ad esempio:
      `LSet x = new LSet()`
   3. `LSet` inizializzato costruito con:
      1. metodo `ins()` a partire da:
         1. `LSet` vuoto:
            1. con intero.
               Ad esempio:
               `LSet x = LSet.empty().ins(3)`
            2. con `LVar` non inizializzata.
               Ad esempio:
               `LVar x0 = new LVar("x0");`
               `LSet x = LSet.empty().ins(x0)`
            3. con `LVar` inizializzata.
               Ad esempio:
               `LVar x1 = new LVar("x1",2);`
               `LSet x = LSet.empty().ins(x1)`
            4. con intero, più `LVar` non inizializzata, più `LVar` inizializzata.
               Ad esempio:
               `LSet x = LSet.empty().ins(1).ins(x0).ins(x1)`
         2. `LSet` non inizializzato:
            1. con intero.
               Ad esempio:
               `LSet r = new LSet("r")`
               `LSet x = r.ins(1)`
            2. con `LVar` non inizializzata.
               Ad esempio:
               `LSet x = r.ins(x0)`
            3. con `LVar` inizializzata.
               Ad esempio:
               `LSet x = r.ins(x1)`

4. con intero, più `LVar` non inizializzata, più `LVar` inizializzata.
   Ad esempio:
   ```
   LSet x = r.ins(x1).ins(x0).ins(1)
   ```

2. metodo `insAll()` a partire da:
   1. `LSet` vuoto:
      1. con array.
         Ad esempio:
         ```
         Integer [] a1={1,2,3};
         LSet x = LSet.empty().insAll(a1);
         ```
      2. con vector.
         Ad esempio:
         ```
         Vector<Integer> x1 = new Vector();
         v1.add(1); v1.add(2); v1.add(3);
         LSet x = LSet.empty().insAll(v1);
         ```
   2. `LSet` non inizializzato:
      1. con array.
         Ad esempio:
         ```
         LSet x = r.insAll(a1);
         ```
      2. con vector.
         Ad esempio:
         ```
         LSet x = r.insAll(v1);
         ```
   3. `LSet` inizializzato:
      1. con array.
         Ad esempio:
         ```
         LSet t = LSet.empty().ins(5);
         LSet x= t.insAll(a1);
         ```
      2. con vector.
         Ad esempio:
         ```
         LSet x = t.insAll(v1);
         ```

4. `LSet` inizializzato, costruito con costruttori a cui è passato un:
   1. `LSet` vuoto.
      Ad esempio:
      ```
      LSet x = LSet.empty();
      LSet x = new LSet(x1);
      ```
   2. `LSet` con elementi inseriti con metodo `ins()`.
      Ad esempio:
      ```
      LSet t = LSet.empty().ins(5);
      LSet x = new LSet(t);
      ```

3. `LSet` non inizializzato.
   Ad esempio:
   ```
   LSet r= new LSet(″r″);
   LSet x= new LSet(x1);
   ```

5. `LSet` inizializzato, costruito con costruttori a cui è passato un:
   1. `Set` vuoto.
      Ad esempio:
      ```
      Set s = new HashSet();
      LSet x= new LSet(s);
      ```
   2. `Set` con elementi inseriti con il metodo `add()`.
      Ad esempio:
      ```
      s.add(1); s.add(2);
      LSet x= new LSet(s);
      ```

3. **Casistica 3:**
   1. `LSet` vuoto.
   2. `LSet` non inizializzato.
   3. `LSet` inizializzato costruito con:
      1. metodo `ins()` a partire da:
         1. `LSet` closed.
            Ad esempio:
            ```
            LSet t = LSet.empty().ins(5);
            LSet x = t.ins(3);
            ```
         2. `LSet` open.
            Ad esempio:
            ```
            LSet r= new LSet(″r″);
            LSet x = r.ins(1).ins(2);
            ```
      2. costruttori a cui è passato un:
         1. `LSet` closed.
            Ad esempio:
            ```
            LSet x = new LSet(t);
            ```
         2. `LSet` open.
            Ad esempio:

            ```
            LSet y = new LSet(x)
            ```
         3. `Set`.
            Ad esempio:
            ```
            Set s = new HashSet();
            ```

```
s.add(1); s.add(2);
LSet x = new LSet();
```

Anche in questo caso si utilizza la tecnica dell'*equivalence testing* vista nei precedenti capitoli nel quale gli input vengono divisi in classi di equivalenza per cui tutti gli altri casi non vengono presi in considerazione dato che appartengono già alle classi di equivalenza descritte.

Per testare tutti i metodi presenti nella classe `LSet` si è utilizzata la classe `LSetTest`.

Nella classe **LSetTest** i metodi testati sono i seguenti:

- **getName()**: ha il compito di restituire una stringa contenente il nome del `LSet` se specificato, in caso contrario restituirà `?` e se l'insieme è vuoto `emptyLSet`.

  I casi presi in considerazione per testare tale metodo sono quelli descritti nella **casistica 1**.

  Vediamo ora un esempio per il caso *1.1*

  ```
  @Test
  public void testGetNameWithName() {
            LSet x = LSet.empty().ins(1).setName("x");
           assertTrue("x".equals(x.getName()));
  }
  ```

  Come si può notare viene creato un `LSet` il cui nome esterno è `x`; attraverso un `assertTrue` si confronta la stringa ottenuta attraverso il metodo `getName()` e una stringa inserita che si presume sia il risultato del metodo nel caso in cui funzioni correttamente.

- **setName(s)**: prende come argomento una stringa `s` che viene assegnata al campo `name` diventando così il nome del `LSet`.

  I casi presi in considerazione per testare questo metodo sono quelli elencati nella **casistica 1**.

- **getValue()**: restituisce i valori presenti nel `LSet` come `Set` di `Java.util` se è stato inizializzato altrimenti restituisce `null`.

  Per testare questo metodo ci focalizziamo sugli elementi contenuti negli `LSet` piuttosto che preoccuparci di come gli `LSet` vengono creati; quindi i casi presi in considerazione sono quelli descritti nella **casistica 2**.

  Vediamo ora un esempio di test creato per il caso *2.3.1.1.3*

```
@ Test
public void testGetValueInitializedWithInsLVarInitialized() {
          LVar x1 = new LVar("x1",1);
          LSet x = LSet.empty().ins(x1);
          Set y = new HashSet();
          y.add(1);
          assertEquals(y,(x.getValue()));
}
```

Si può notare che viene creata una `LVar` di nome `x1` inizializzata ad 1 e si dà vita ad un `LSet` inizialmente vuoto in cui si inserisce la `LVar` appena creata.

Per effettuare il test si utilizza un `assertEquals` che mette a confronto il risultato della `getValue()` con un `Set` di Java che si suppone essere l'esito della `getValue()` nel caso in cui il metodo funzioni correttamente.

- **toString()**: restituisce una stringa che rappresenta il valore del `LSet` se è inizializzato, altrimenti `_nomeLSet` se l'insieme ha nome oppure `_?` se non ha nome.

  Come per il metodo precedente si vuole dare importanza agli elementi contenuti nel `LSet`, per questo i casi presi in considerazione sono quelli della **casistica 2**.

  Riportiamo di seguito un esempio per il caso *2.4.2*:

```
@Test
public void testToStringInitializedWithNewLSetIns() {
          LSet x1 = LSet.empty().ins(1);
          LSet x= new LSet(x1);
          assertEquals("{1}",(x.toString()));
}
```

  Come si può semplicemente notare è stato creato un `LSet` attraverso il costruttore a cui è passato un `LSet` inizializzato. Attraverso l'`assertEquals` si confronta la stringa ottenuta tramite il metodo con la stringa fornita che si suppone essere quella restituita dal metodo `toString()` nel caso questo funzioni correttamente.

- **normalize()**: utile per eliminare gli eventuali elementi duplicati presenti nel LSet.

Come per il metodo precedente si è deciso di dare importanza agli elementi contenuti nel `LSet`. Per questo i casi presi in considerazione sono quelli della **casistica 2**.

Un chiaro esempio è quello sotto descritto che testa il metodo per il caso *2.3.1.1.1*:

```
@Test
   public void testNormalizedInitializedWithInsEmpty() {
     LSet x = LSet.empty().ins(1).ins(2).ins(1);
     LSet y = LSet.empty().ins(2).ins(1).setName("y");
     assertEquals(y.toVector(),(x.normalizeSet().toVector()));
  }
```

In questo metodo viene creato un `LSet`, nel quale ci sono tre elementi di cui due ripetuti e un `LSet` definito nello stesso modo ma senza ripetizioni. Tramite l'`assertEquals` si mettono a confronto i due `LSet` a uno dei quali è stato applicato il metodo preso in esame.

- **isKnown()**: restituisce un booleano, `true` se l'insieme è stato inizializzato, `false` altrimenti.

  Il metodo viene testato basandosi sui casi descritti nella **casistica 3** che prende in considerazione la modalità di creazione degli `LSet`.

  Ovviamente per effettuare questi test si è utilizzata la `assertTrue` o la `assertFalse` in base alle esigenze.

  Un esempio di tale test è quello riportato qui di seguito che controlla il corretto funzionamento per il caso *3.3.1.1*:

```
@Test
public void testIsKnownIninializedWithInsClosed() {
          LSet x = LSet.empty().ins(1).ins(2);
          assertTrue(x.isKnown());
}
```

  In questo metodo viene creato un `LSet` inizializzato con due elementi e attraverso una `assertTrue` si controlla che il metodo funzioni correttamente.

- **getRest()**: in alcuni casi è possibile che si creino degli insiemi in cui non tutti gli elementi sono specificati; questo metodo restituisce la parte non definita del `LSet`, se esiste, altrimenti restituiscire l'insieme vuoto.

I casi presi in considerazione sono quelli che descrivono le modalità di creazione degli `LSet` quindi quelli elencati nella **casistica 3**.

Vediamo ora un semplice test per capire meglio come si opera, con riferimento il caso *3.3.2*:

```
@Test
public void testgetRestIninializedWithConstructLSetOpen() {
     LSet r = new LSet("r");
     LSet x1 = r.ins(1).ins(2);
     LSet x = new LSet(x1);
     assertEquals(r,(x.getRest()));
}
```

Come si può notare il `LSet` creato è {`2,1|_r`}. Si mette quindi a confronto la parte non specificata dell'insieme (rappresentata dal `LSet` non inizializato r) con il risultato restituito dal metodo `getRest()` richiamato sul `LSet`.

- **isClosed()**: restituisce un booleano, `true` se l'insieme ha resto vuoto, `false` altrimenti.

  Come nel caso precendente i casi presi in considerazione sono quelli descritti nella **casistica 3**.

  Vediamo un esempio per il caso *3.3.2*:

```
@Test
   public void testIsClosedIninializedWithInsOpen() {
      LSet r= new LSet();
      LSet x = r.ins(1).ins(2);
      assertFalse(x.isClosed());
   }
```

  Come si nota l'insieme non è del tutto specificato quindi applicando il metodo al `LSet` il risultato sarà sicuramente `false`.

- **isEmpty()**: restituisce un booleano, `true` se l'insieme è vuoto, `false` altrimenti.

  Come per i metodi precedenti, i casi presi in considerazione sono quelli che descrivono le modalità di creazione degli `LSet` quindi quelli elencati nella **casistica 3**.

Un particolare caso degno della nostra attenzione è sicuramente il test eseguito per il caso *3.2* in cui viene testato il metodo `isEmpty()` per un LSet non inizializzato.

In questo caso è necessario fare attenzione in quanto verrà sollevata un'eccezione di tipo `NotInitVarException`.

Il codice del metodo di test è il seguente:

```
@Test
  public void testIsEmptyUnitialized() {
  LSet x = new LSet();
  try{
    x.isEmpty();
    fail("Dovrebbe sollevare un'eccezione");
  }
  catch(NotInitVarException e){}
}
```

L'eccezione viene gestita nel blocco `try/catch` il quale forza il fallimento e si preoccupa di visualizzare un messaggio, tramite il metodo `fail()`, nel caso in cui l'eccezione non sia lanciata .

- **isGround()**: restituisce un booleano, `true` se l'insieme è vuoto, completamente specificato e non contiene variabili non inizializzate; `false` altrimenti.

  Anche per questo metodo i casi presi in considerazione sono quelli che si focalizzano sulle modalità di creazione degli `LSet` e quindi elencati nella **casistica 3**.

  Come per il metodo `isEmpty()` quando si applica il metodo `isGround()` ad un `LSet` non inizializzato, come ad esempio per il caso *3.2* sarà sollevata un'eccezione di tipo `NotInitVarException`.
  Vediamo la porzione di codice che testa questa situazione:

```
@Test
  public void testIsGroundUnitialized() {
    LSet x = new LSet();
    try{
      x.isGround();
      fail("Dovrebbe sollevare un'eccezione");
    }
```

```
    catch(NotInitVarException e){}
}
```

L'eccezione è catturata nel blocco `try/catch` il quale forza il fallimento e si proccupa di visualizzare un messaggio attraverso il metodo `fail` nel caso in cui l'eccezione non sia lanciata.

- **equals(LSet)**: il quale mette a confronto due `LSet` e restituisce `true` se sono uguali oppure `false` in caso contrario.

  Ovviamente per effettuare questi test si è utilizzata la `assertTrue` o la `assertFalse` in base ai casi ottenuti.

  Come si può notare dalla tabella riportata di seguito, vengono combinati tra loro i casi elencati nella prima parte della **casistica 2**. Ci si è fermati a studiare la casistica fino agli `LSet` inizializzati costruiti con il metodo `ins` a partire da un `LSet` non inizializzato quindi fino al caso **2.3.2.1** compreso.

  Tutti gli altri casi vengono ignorati in quanto si ritiene che rientrino nelle stessa classe di equivalenza di quelli scelti.

LS_1	LS_2	$1.equals(2)$
Ø **(2.1)**	Ø **(2.1)**	true
	non iniz **(2.2)**	false
	Ø.ins(int) **(2.3.1.1.1)**	false
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	false
	Ø.(LVar non iniz).(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false

LS_1	LS_2	$1.equals(2)$
non inizializzato x **(2.2)**	Ø **(2.1)**	false
	non iniz x **(2.2)**	true
	non iniz y **(2.2)**	false
	Ø.ins(int) **(2.3.1.1.1)**	false
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	false
	Ø.(LVar non iniz). .(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false
Ø.ins(int) **(2.3.1.1.1)**	Ø**(2.1)**	false
	non iniz **(2.2)**	false
	Ø.ins(int) **(2.3.1.1.1)**	true
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	true
	Ø.(LVar non iniz). .(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false
Ø.ins(LVar non iniz) **(2.3.1.1.2)**	Ø **(2.1)**	false
	non iniz **(2.2)**	false
	Ø.ins(int)**(2.3.1.1.1)**	false
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	false
	Ø.(LVar non iniz). .(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false
Ø.ins(LVar iniz) **(2.3.1.1.3)**	Ø **(2.1)**	false
	non iniz **(2.2)**	false
	Ø.ins(int) **(2.3.1.1.1)**	true
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	true
	Ø.(LVar non iniz). .(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false

LS_1	LS_2	1.$equals$(2)
Ø.(Lv N Un). .(Lv iniz).(int) **(2.3.1.1.4)**	Ø **(2.1)**	false
	non iniz **(2.2)**	false
	Ø.ins(int) **(2.3.1.1.1)**	false
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	false
	Ø.(LVar non iniz). .(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false
non iniz.ins(int) **(2.3.1.2.1)**	Ø **(2.1)**	false
	non iniz **(2.2)**	false
	Ø.ins(int) **(2.3.1.1.1)**	false
	Ø.ins(LVar non iniz) **(2.3.1.1.2)**	false
	Ø.ins(LVar iniz) **(2.3.1.1.3)**	false
	Ø.(LVar non iniz). .(LVar iniz).(int) **(2.3.1.1.4)**	false
	non iniz.ins(int) **(2.3.1.2.1)**	false

La tabella è di facile comprensione, ma diamo alcune spiegazioni in modo da essere più chiari:

- in grassetto per ogni caso è indicato il numero corrispondente alla casistica descritta precedentemente in modo da avere un'idea più limpida;
- `LSet` non inizializzato `x` è diversa dal `LSet` non inizializzato `y` in quanto hanno riferimenti diversi.

- **equals(Object)**: confronta un `LSet` con un oggetto e restituisce `true` se sono uguali, `false` altrimenti.

  Per effettuare tutti i test si sono combinati i casi elencati nella **casistica 2** con oggetti diversi come `Set` inizilizzati, non inizializzati, interi e array.

5.3 Test della classe `LList`

Come già detto una `LList` è una lista di elementi di qualsiasi tipo, costituita da quattro attributi del tutto simili a quelli di un `LSet`: **name**, **init**, **lista** e **resto**.

La casistica presa in considerazione per testare tali metodi è esattamente quella usata per testare i metodi degli `LSet` con le opportune modifiche.

Per testare tutti i metodi presenti nella classe `LList` si è utilizzata la classe `LListTest`.

Su questa base sono stati testati i seguenti metodi della classe `LList`:

- **getName()**: restituisce una stringa contenente il nome della lista se specificato, in caso contrario restituirà `?` e se la LList è vuota verrà restituita la stringa contentente `emptyLList`.

  I casi considerati per testare tale metodo sono quelli descritti nella **casistica 1**.

  Vediamo ora un esempio di test utile per capire se il metodo funziona correttamente per una lista vuota, come per il caso *1.3*.

  ```
  @Test
  public void testGetNameEmpty() {
            LList x = LList.empty();
            assertTrue("emptyLList".equals(x.getName()));
  }
  ```

  Come si può notare viene creata una lista vuota e tramite una `assertTrue` si controlla che il risultato del metodo sia uguale alla stringa passata che si suppone essere quella restituita dal `getName()` nel caso funzioni correttamente.

- **setName(s)**: prende come argomento una stringa `s`, che viene assegnata al campo `name` diventando così il nome della lista.

  Anche per questo metodo i casi utilizzati sono quelli descritti nella **casistica 1**.

- **getValue()**: restituisce i valori presenti nella `LList` (come `List` di `java.util`) se è stata inizializzata altrimenti restituisce null.

  I casi presi in considerazioni dai vari test sono quelli descritti nella **casistica 2**.

Vediamo ora un semplice esempio di test per il caso *2.2*.

```
@Test
public void testGetValueUninitialized() {
           LList x = new LList();
           assertEquals(null,(x.getValue()));
}
```

È facile capire che viene creata una lista non inizializzata e tramite una `assertEquals` si confronta il risultato del metodo con `null` che si suppone essere ciò che restituisce la `getValue()` nel caso funzioni correttamente.

- **toString()**: restituisce una stringa che rappresenta il valore della `LList` se è inizializzata, altrimenti `_nomeLList` se la lista ha nome oppure `_?` se non ha nome.

  I casi presi in considerazione per testare questo metodo sono quelli descritti nella **casistica 2**.

  Vediamo un semplice esempio di test per il caso *2.3.1.2.1*:

```
@Test
public void testToStringUninitialized_insInteger() {
     LList r = new LList("r");
    LList x = r.ins(1);
    assertEquals("[1|_r]",(x.toString()));
}
```

  Come si può notare si crea una lista di nome `r` non inizializzata ed attraverso il metodo `ins()` si crea una `LList` non completamente specificata.

  Tramite la `assertEquals` è possibile fare il confronto con una stringa creata come dovrebbe essere il risultato della `toString()` nel caso in cui il metodo funzioni correttamente.

- **isKnown()**: si comporta esattamente come negli LSet restituendo un booleano, `true` se la lista è stata inizializzata, `false` altrimenti.

  Si è testato il metodo utilizzando i casi descritti nella **casistica 3**.

  Vediamo un semplice esempio per il caso *3.3.1.1*:

```
@Test
public void testIsKnownIninializedWithInsClosed() {
          LList x = LList.empty().ins(1).ins(2);
          assertTrue(x.isKnown());
}
```

Si intuisce con una certa facilità che si è creata una lista inizializzata con tutti gli elementi specificati. Tramite una `assertTrue` si controlla il corretto funzionamento del metodo.

- **getRest()**: restituisce la parte della `LList` non inizializzata, se esiste, altrimenti restituisce l'insieme vuoto.

  I casi utilizzati per testare il corretto funzionamento del metodo sono quelli descritti nella **casistica 3**.

  Vediamo un esempio di tale test in cui si fa riferimento al caso *3.3.2.1*:

```
@Test
 public void testgetRestIninializedWithConstructLListClosed() {
   LList x1 = LList.empty().ins(1).ins(2);
   LList x = new LList(x1);
   LList y = LList.empty();
   assertEquals(y,(x.getRest()));
}
```

  Si nota facilmente che la lista logica creata è completamente inizializzata quindi il metodo che si sta testando restituirà l'insieme vuoto.

- **isClosed()**: restituisce un booleano, `true` se la lista ha la parte del resto vuota, `false` altrimenti.

  I casi utilizzati per testare il corretto funzionamento del metodo sono quelli descritti nella **casistica 3**.

  Vediamo ora un esempio di tale test relativo al caso *3.1*:

```
@Test
   public void testIsClosedInitializedEmpty() {
      LList x = LList.empty().setName("x");
      assertTrue(x.isClosed());
 }
```

Si nota che la lista logica non è inizializzata quindi il metodo restituirà il booleano `true` in quanto `x` ha resto vuoto.

- **isEmpty()**: restituisce un booleano, `true` se la lista è vuota, `false` altrimenti.

  I casi testati sono quelli presenti nella casistica 3.

  Anche per le liste come per gli `LSet` quando si testa il metodo per una `LList` non inizializzata (*3.2*) verrà sollevata un'eccezione del tipo `NotInitVarException`. Vediamo la porzione di codice che testa questa situazione:

```
@Test
public void testIsEmptyUnitialized() {
  LList x = new LList();
  try{
     x.isEmpty();
     fail("Eccezione non sollevata");
    }
   catch(NotInitVarException e){}
}
```

  Come si può notare nel caso in cui l'eccezione sia sollevata il blocco `try/catch` forza il fallimento, mentre se per errore l'eccezione non fosse sollevata il metodo `fail()` visualizza un messaggio che ci fa capire che qualcosa è andato storto.

- **isGround()**: restituisce un booleano, `true` se la lista è vuota, completamente specificata e non contiene variabili non inizializzate; `false` altrimenti.

  I casi testati sono quelli descritti nella **casistica 3**.

  Anche in questo metodo, come nel precedente, applicando `isGround()` ad una lista non inizializzata verrà sollevata un'eccezione che verrà gestita esattamente come per il metodo precedente.

  Vediamo di seguito un esempio per il caso *3.3.2.3*

```
@Test
 public void testIsGroundIninializedWithConstructListClosed() {
    List x1 = new ArrayList();
```

```
    x1.add(1);x1.add(2);
    LList x = new LList(x1);
    LList y = LList.empty();
    assertTrue(x.isGround());
}
```

Come si intuisce la lista logica è completamente specificata quindi il metodo restituirà un booleano a `true`.

- **equals(LList)**: mette a confronto due `LList` e restituisce `true` se sono uguali oppure `false` in caso contrario.

  Da sottolineare il fatto che in una lista, al contrario degli insiemi, l'ordine degli elementi è importante.
  Quindi liste con elementi uguali ma in ordine diverso sono due liste completamente diverse.

  Ovviamente per effettuare questi test si è utilizzata la `assertTrue` o la `assertFalse` in base ai casi considerati.

5.4 Test della classe `IntLSet`

Come già detto un `IntLSet` è un'estensione della classe `LSet` in cui elementi possono essere soltanto numeri interi o `IntLVar`.

I metodi testati e la casistica proposta sono esattamente uguali a quelli descritti per gli `LSet`.

Per testare tutti i metodi presenti nella classe `IntLSet` si è utilizzata la classe `IntLSetTest`.

I metodi testati in questo caso non verranno descritti in quanto è stato già fatto in precedenza, ma verrà riportato un semplice elenco:

- **getName()**.
- **setName(string)**.
- **getValue()**.
- **normalize()**.
- **toString()**.
- **getRest()**.

- **isEmpty()**.
- **isClosed()**.
- **isGround()**.
- **isKnown()**.
- **equals(IntLSet)**.
- **equals(Object)**.

5.5 Test della classe `IntLVar`

Le varibili logiche intere sono un particolare caso delle variabile logiche, già descritte, nelle quali i valori si limitano ad essere dei numeri interi.

Sono composte dai seguenti attributi:

- **name**: contiene una stringa che corrisponde al nome esterno della `IntLVar`, se è stato specificato, altrimenti viene settato con il simbolo `?`;
- **init**: flag che indica l'inizializzazione della variabile intera. Se è settato a `true` significa che la varibile logica è inizializzata, in caso contrario sarà `false`;
- **val**: contiene un eventuale valore della `IntLVar` nel caso in cui sia inizializzata oppure `null` in caso contrario.
- **domain**: inizialmente a `null`, conterrà il dominio della variabile. Tale dominio può essere definito nel momento della creazione con l'utilizzo dei costruttori oppure quando i vincoli vengono risolti.

  Quando si dà vita ad una `IntLVar` se al costruttore gli si passano due numeri interi `a` e `b` il dominio sarà `[a,b]` e ovviamente `a` dovrà essere *minore* di `b`, in caso contrario sarà sollevata una eccezione di nome `NotValidDomainException()`; se al costruttore si passa invece un solo numero intero `a` il dominio sarà `[a,a]` e per finire se l'`IntLVar` è creata ma non viene inizializzata il dominio sarà `[-∞, +∞ ]`.
- **Constraints**: contiene un vincolo aritmetico associato alla variabile logica intera come già descritto nei precedenti paragrafi.

Anche in questa circostanza i casi utilizzati per controllare il corretto funzionamento dei metodi sono stati divisi in gruppi.

- **Casistica 1:**

  1. `IntLVar` con nome.
     Ad esempio:
     `IntLVar a = new IntLVar("a");`
  2. `IntLVar` senza nome.
     Ad esempio:
     `IntLVar a = new IntLVar();`

- **Casistica 2:**

  1. `IntLVar` non inizializzata.
     Ad esempio:
     `IntLVar x = new IntLVar();`
  2. `IntLVar` inizializzata con costruttore a cui è passata:
     1. `IntLVar` non inizializzata.
        Ad esempio:
        `IntLVar x1 = new IntLVar("x1");`
        `IntLVar x = new IntLVar(x1);`
     2. `IntLVar` inizializzata con intero.
        Ad esempio:
        `IntLVar x1 = new IntLVar("x1",1);`
        `IntLVar x = new IntLVar(x1);`
     3. `IntLVar` inizializzata con intervallo.
        Ad esempio:
        `IntLVar x1 = new IntLVar("x1",1,10);`
        `IntLVar x = new IntLVar(x1);`

- **Casistica 3:**

  1. `IntLVar` non inizializzata.
     Ad esempio:
     `IntLVar a = new IntLVar();`
  2. `IntLVar` inizializzata:
     1. con intero.
        Ad esempio:
        `IntLVar x1 = new IntLVar(3);`

2. con `IntLVar`.
   Ad esempio:
   `IntLVar x= new IntLVar(x1);`

3. `IntLVar` non inizializzata, con constraint.
   Ad esempio:
   ```
   IntLVar x= new IntLVar();
   IntLVar y= new IntLVar();
   y=x.mul(1);
   ```

Si è utilizzata la tecnica dell'*equivalence testing* vista nei precedenti capitoli nel quale gli input vengono divisi in classi di equivalenza per cui tutti gli altri casi non sono stati presi in considerazione dato che appartengono già alle classi di equivalenza descritte.

Per testare tutti i metodi presenti nella classe `IntLVar` si è utilizzata la classe `IntLVarTest`.

I metodi sottoposti ai test sono i seguenti.

- **getName()**: ha il compito di restituire una stringa contenente il nome della `IntLVar` se specificato, in caso contrario restituirà il simbolo `?`.

  I casi che utilizziamoi per testare questo metodo sono quelli descritti nella **casistica 1**.

  Vediamo un esempio di questi test nel caso in cui la `IntLVar` non abbia nome, come nel caso *1.2*:

  ```
  @Test
     public void testGetNameWithNoName() {
          IntLVar a = new IntLVar();
          assertTrue("?".equals(a.getName()));
  }
  ```

  Come si può notare viene messo a confronto la stringa "?" (che si suppone essere il risultato del metodo nel caso in cui funzioni correttamente) e ciò che restituisce il `getName()` applicato alla `IntLVar`.

- **setName(String)**: prende come argomento una stringa, la quale verrà assegnata al campo `name` diventando così il nome della `IntLVar`.

  I casi presi in considerazione per testare il metodo sono quelli descritti nella **casistica 1**.

- **getValue()**: restituisce `null` nel caso in cui la variabile logica intera non sia stata inizializzata e un `Integer` in caso contrario.

  I casi utilizzati per testare questo metodo sono quelli descritti nella **casistica 2**.

  Vediamo ora un semplice esempio che controlla il corretto funzionamento del metodo per il caso *2.2.2*:

```
@Test
    public void testGetValueNewIntLVarInitializedWithInteger() {
         IntLVar x1 = new IntLVar("x1",1);
         IntLVar x = new IntLVar(x1);
         Integer y = 1;
         assertEquals(y,(x.getValue()));
    }
```

  Come si può facilmente notare è stato creato una `IntLVar` chiamata `x1` e inizializzata con il valore `1`, dopodichè si da vita ad una seconda `IntLVar` attraverso il costruttore a cui è passata la precedente `IntLVar`. In un secondo momento si confronta un intero (che si suppone essere quello corretto nel caso in cui il metodo funzioni correttamente) e il risultato della `getValue()`.

- **toString()**: restituisce una stringa rappresentante il valore contenuti nella `IntLVar` oppure la stringa `_nomeIntLVar` se la `IntLVar` ha nome oppure `_?` se non lo ha.

  I casi presi in considerazione per testare tale metodo sono quelli descritti nella **casistica 2**.

  Vediamo ora un semplice esempio che controlla il corretto funzionamento del metodo per il caso *2.1*:

```
@Test
    public void testToStringUninitialized() {
        IntLVar x = new IntLVar("x");
        assertEquals("_x",(x.toString()));
     }
```

  La `IntLVar` presa in considerazione è non inizializzata quindi applicando il metodo `toString()` verrà restituita una stringa contentente il nome della variabile logica intera.

- **isKnown()**: restituisce un booleano, `true` se la `IntLVar` è inizializzata, `false` altrimenti.
  Per effettuare i test usiamo le `assertTrue` o la `assertFalse` in base al booleano restituito dal metodo.

  I casi presi in considerazione per testare tale metodo sono quelli descritti nella **casistica 2**.

  Di seguito viene riportato un esempio per il caso *2.2.3*:

```
@Test
public void testisKnownNewIntLVarInitializedWithIntervall() {
          IntLVar x1 = new IntLVar("x1",1,10);
          IntLVar x = new IntLVar(x1);
          assertFalse(x.isKnown());
}
```

  Si controlla che la `IntLVar` creata tramite un'altra `IntLVar` sia inizializzata, in questo caso la risposta è falsa.

- **getDom()**: restituisce il dominio della `IntLVar` come visto precedentemente.

  I casi presi in considerazione per testare tale metodo sono quelli della **casistica 2**.

  Vediamo un esempio per il caso *2.1*:

```
@Test
 public void testGetDomNewIntLVarUninitialized() {
   IntLVar x1 = new IntLVar("x1");
   IntLVar x = new IntLVar(x1);
   MultiInterval a= new MultiInterval
              (Interval.MIN_VALUE, Interval.MAX_VALUE);
   assertEquals(a,(x.getDom()));
 }
```

  In questo caso si vuole capire qual è il dominio di una `IntLVar` definita tramite una `IntLVar` non inizializzata e senza un dominio esplicito. Si crea un oggetto di tipo `MultiInterval a` che è proprio l'intervallo che dovrebbe risultare. Ovviamente l'intervallo sarà compreso tra il minimo valore macchina e il massimo valore macchina.

- **sum()**: è un metodo che prende due argomenti e ne restituisce la somma tramite una `IntLVar` nel cui campo `Constraint` sarà inserito il vincolo creato.

  Per testare il metodo sono state utilizzate le `IntLVar` descritte nella **casistica 3** a cui sono state sommati interi, altre `IntLVar` inizializzate con interi e altre `IntLVar` non inizializzate costruite con altri vincoli precedentemente creati.

  Per capire meglio come vengono affrontati i test mostriamo un esempio relativo al caso *3.2.1* in cui alla `IntLVar` si somma un intero:

```
@Test
    public void testSUMIntLVarUninitialized_Integer() {
         IntLVar x= new IntLVar();
         IntLVar e= x.sum(1);
         Constraint c= new Constraint();
         c.and(new Constraint
              (e,Environment.name_to_code("sum"),x,1));
         assertEquals(c,e.getConstraint());
    }
```

  Dopo aver costruito il vincolo $e = x+1$ implicitamente con il metodo `sum`, lo stesso vincolo viene costruito esplicitamente tramite il costruttore di default di constraint a cui sono passati i due argomenti di `sum`, il codice corrispondente a `sum` e la `IntLVar` risultante.

  Quindi si confronta il vincolo creato tramite la `sum` e recuperato attraverso il metodo `getConstraint()` con quello creato con il costruttore di Constraint.

- **mul()**: è un metodo che prende due argomenti e ne restituisce il prodotto tramite una `IntLVar` nel cui campo `Constraint` sarà inserito il vincolo creato.

  Per testare il metodo sono state utilizzate le `IntLVar` descritte nella **casistica 3** a cui sono state sommati interi, altre `IntLVar` inizializzate con interi e altre `IntLVar` non inizializzate costruite con altri vincoli precedentemente creati.

  Per capire meglio come vengono affrontati i test mostriamo un esempio dove si effettua la moltiplicazione del caso *3.1* con un `IntLVar` inizializzata tramite un intero.

```
@Test
    public void testMULIntLVarUninitialized_
                                IntLVarInitializedInteger() {
    IntLVar x= new IntLVar();
    IntLVar y= new IntLVar(3);
    IntLVar e= x.mul(y);
    ConstraintsConjunction c= new ConstraintsConjunction();
    c.and(new Constraint (e,Environment.name_to_code("mul"),x,y));
    assertEquals(c,e.getConstraint());
}
```

Dopo aver costruito il vincolo $e = x*3$ implicitamente con il metodo `mul`, lo stesso vincolo viene costruito esplicitamente tramite il costruttore di default di constraint a cui sono passati i due argomenti di `mul`, il codice corrispondente a `mul` e la `IntLVar` risultante.

Quindi si confronta il vincolo creato tramite la `mul` e recuperato attraverso il metodo `getConstraint()` con quello creato con il costruttore di Constraint.

5.6 Test del constraint solving

Come visto nel capitolo 4 per risolvere un vincolo `C` bisogna creare un `solver` `S` definendo un'istanza di `SolverClass`, inserire il nuovo vincolo nel C.store di S effettuando una chiamata al metodo `S.add(C)`.

Una volta aggiunti nuovi vincoli al C.store, si può chiedere al solver S di *risolverli* invocando uno degli appositi metodi. Oppure più semplicemente basta risolvere il vincolo tramite `S.solve(C)`

La classe `SolverClass` si occupa dell'introduzione dei vincoli nel constraint store e della loro risoluzione.

La risoluzione di un constraint `C` di fatto genera un nuovo constraint store che può essere vuoto se il constraint `C` è stato risolto completamente oppure può essere una congiunzione di tipo $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ che è una versione logicamente equivalente di `C` così come prodotta dalle regole di riscrittura del solver.

Quindi in generale per testare la risoluzione di un constraint `C`, sapendo che `C` viene riscritto nell congiunzione $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ con $n \geq 0$, è necessario operare nel seguente metodo:

```
SolverClass solver = new SolverClass();
Constraints c;
```

```
solver.solve(C);
c_new = solver.getConstraint();
assertEquals("nuovo_constraint",c_new.toString());
```

dove `nuovo_constraint` è la stringa che rappresenta la congiunzione $c_1 \wedge c_2 \wedge \ldots \wedge c_n$.

Il metodo `getConstraint()` applicato al `solver` permette di estrarre dal `constraint store` di `solver` il vincolo corrente.

Quando un vincolo è completamente risolto il constraint store sarà rappresentato tramite il simbolo *[ ]*.

I constraint controllati in questi test sono quelli delle classi `LVar, LSet, LList` e `IntLVar`.

5.6.1 Constraint per `LVar`

I vincoli testati per la classe `LVar` sono della forma

$$v_1\texttt{.op(}v_2\texttt{)}$$

dove:

- v_1 è una `LVar`,
- `op` è uno degli operatori `eq, neq, in, nin`,
- v_2 cambia in base al constraint considerato.

Di seguito riportiamo la casistica relativa a v_1 utilizzata per testare i constraint di `LVar`.

- **Casistica 1:**
  1. una `LVar` non inizializzata.
     Ad esempio:
     `LVar x = new LVar();`
  2. una `LVar` inizializzata
     1. con intero.
        Ad esempio:
        `LVar x = new LVar(2);`
     2. con oggetto (in particolare `LList`).
        Ad esempio:
        `LList x1 = LList.empty().ins(1);`
        `LVar x = new LVar(x1);`

I constraint sottoposti ai test sono i seguenti.

- **eq(v_2)**: prende due valori v_1 e v_2 e li unifica.

  Per questo vincolo v_2 può essere un object quindi si utilizzano gli stessi casi della **casistica 1** a cui si aggiunge il caso in cui v_2 è un intero.

  Vediamo un esempio di questi test nel caso in cui si confronti una `LVar` non inizializzata (caso *1.1*) con un intero:

```
@Test
    public void testEQLVarUninitializedx_Integer()
         throws NotDefConstraintException, Failure {
        SolverClass solver = new SolverClass();
        Constraint c;
        LVar x = new LVar("x1");
        solver.solve(x.eq(1)); // x=1
        c = solver.getConstraint();
        assertEquals("[]",c.toString());
     }
```

  Questo vincolo viene risolto totalmente quindi tramite la `assertEquals` viene confrontata la stringa ottenuta tramite il metodo `toString()` applicato al vincolo risultante dalla `solve` con la stringa ″[ ]″ (ovvero il constraint vuoto).

- **neq(v_2)**: prende due valori v_1 e v_2 verifica che non siano unificabili.

  Anche per testare questo vincolo, come nel precedente sono stati utilizzati i casi elencati nella **casistica 1** a cui si è aggiunto il caso in cui l'oggetto passato sia un intero.

  Vediamo un esempio in cui v_1 è una `LVar` inizializzata tramite una `LList` (caso *1.2.2*) mentre v_2 è una `LVar` non inizializzata:

```
@Test
    public void testNEQLVarInitializedWithLlist_
                LVarUninitialized() throws
          NotDefConstraintException, Failure {
    SolverClass solver = new SolverClass();
    Constraints c;
    LVar x0 = new LVar(2);
    LList x1 = LList.empty().ins(x0);
```

```
    LVar x = new LVar(x1);
    LVar y = new LVar();
    solver.solve(x.neq(y));
    //x=[2], x neq y
    c = solver.getConstraint();
    assertEquals("[2] neq _?",c.toString());
    }
```

Come si può notare il constraint è in una forma irriducibile e viene lasciato inalterato nel `c.store`. Quindi si confronta una stringa che si suppone essere il vincolo creato con il risultato del metodo `toString`.

- **in**(v_2): prende due valori e crea il vincolo di appartenenza insiemistica, $v_1 \in v_2$.

  In questo caso il valore di v_2 potrà essere una collezione di oggetti. Il test viene effettuato utilizzando per v_2 gli `LSet` inizializzati, non inizializzati (cfr. **casistica 2** dei constraint per `LSet`).

  Vediamo ora un semplice esempio che controlla il corretto funzionamento del metodo per il caso *1.1*, in cui v_2 sia un `LSet` inizializzato:

```
@Test
    public void testINLVarUninitializedx_LList()
         throws NotDefConstraintException, Failure {
       SolverClass solver = new SolverClass();
       ConstraintsConjunction c;
       LVar x = new LVar("x1");
       LSet y = LSet.empty().ins(1).ins(2);
       solver.solve(x.in(y)); // x in {1,2}
       c = solver.getConstraint();
       assertEquals("[] ",c.toString());
}
```

  Il constraint viene riscritto nel constraint `x.eq(4)` e quindi, come visto sopra, completamente risolto.

  Dunque confrontiamo il constraint generato con la stringa ″[ ]″.

- **nin**(v_2): prende due valori e crea il vincolo di non appartenenza insiemistica, $v_1 \notin v_2$.

Anche in questo caso come nel precedente il valore di v_2 potrà essere una collezione di oggetti; in particolare utilizziamo gli `LSet` inizializzati e non inizializzati e i `Set`.

Vediamo ora un semplice esempio che controlla il corretto funzionamento del metodo per il caso *1.1* in cui v_2 sia un `LSet` non inizializzato:

```
@Test
  public void testNINLVarUninitializedx_Integer()
         throws NotDefConstraintException, Failure {
     SolverClass solver = new SolverClass();
     ConstraintsConjunction c;
     LVar x = new LVar("x1");
     LSet y = new LSet();
     solver.solve(x.nin(y));
     c = solver.getConstraint();
     assertEquals("_x1 nin _?",c.toString());
}
```

Anche in questo caso si tratta di un vincolo in forma irriducibile per perciò viene lasciato inalterato nel `C.store`.

5.6.2 Constraint per `LSet`

I vincoli testati per la classe `LSet` sono della forma

$$v_1\texttt{.op(}v_2\texttt{)}$$

dove:

- v_1 è un `LSet`,
- `op` è uno deglio operatori `eq, neq, contains, ncontains`
- v_2 cambia in base al constraint considerato.

Di seguito riportiamo la casistica relativa a v_1 utilizzata per testare i constraint.

- **Casistica 2:**

  1. un `LSet` non inizializzato.
     Ad esempio:
     `LSet x = new LSet();`

2. un `LSet` inizializzato:
    1. con un insieme con elementi noti, in particolare interi.
       Ad esempio:
       `LSet x = LSet.empty().ins(2).ins(1);`
    2. con un insieme con resto non noto.
       Ad esempio:
       ```
       LSet r = new LSet();
       r.ins(2);
       ```
    3. con un insieme con elementi non noti.
       Ad esempio:
       ```
       LVar x0 = new LVar();
       LVar x1 = new LVar();
       LSet x =LSet.empty().ins(x0).ins(x1);
       ```

I vincoli sottoposti ai test sono i seguenti:

- **eq(v_2)**: prende due valori v_1 e v_2 e li unifica.

  Per questo metodo il valore di v_2 può essere tutti i casi presi in cosiderazione nella **casistica 2** a cui è stato aggiunto anche un `Set`.

  Un semplice caso di test viene riportato di seguito:

```
@Test
    public void testEQLSetUninitializedx_LSet()
            throws NotDefConstraintException, Failure {
       SolverClass solver = new SolverClass();
       Constraint c;
       LSet x = new LSet("x1");
       LSet y = LSet.empty().ins(3).ins(x);
       try{
           solver.solve(x.eq(y));
           //y={x,3}, x=y
           fail("FALLIMENTO!!!!");
       }
       catch(Failure e) {}
   }
```

  Come si può notare questo è un caso in cui la risoluzione fallisce quindi attraverso un blocco `try/catch` viene gestito il fallimento e attraverso il metodo `fail()` viene stampato un messaggio nel caso in cui la solve non fallisse.

- **neq(v_2)**: prende due valori v_1 e v_2 e li unifica creando dei vincoli di disuguaglianza.

  Per testare questo vicolo i valori di v_1 e v_2 sono esattamente quelli presi in considerazione per il precedente vincolo.

  Di seguito è riportato un semplice esempio di test in cui si utilizza il caso *1.2.3* come valore di v_1, mentre come valore di v_2 si utilizza un `LSet` non inizializzato:

```
@Test
public void testNEQLSetInitializedWithLVarUninitialized
                _LSetUninitializedx()
            throws NotDefConstraintException, Failure {
    SolverClass solver = new SolverClass();
    ConstraintsConjunction c;
    LVar x1 = new LVar();
    LVar x2 = new LVar();
    LSet x = LSet.empty().ins(x1).ins(x2);
    LSet y= new LSet();
    solver.solve(x.neq(y));
    //x={x1,x2} , x neq y
    c = solver.getConstraint();
    assertEquals("{_?,_?} neq _?",c.toString());
}
```

  Come si può notare il vincolo è in forma irriducibile e quindi rimane inalterato nel `c.store`.

- **contains(v_2)**: prende due valori v_1, v_2 e crea il vincolo di non appartenenza insiemistica, $v_1 \in v_2$

  In questo caso il valore di v_2 potrà essere una `LVar` inizializzata, non inizializzata e un object in particolare un intero.

  Vediamo ora un esempio che prende il caso *1.1* come valore di v_1 e una `LVar` inizializzata come valore di v_2:

```
@Test
    public void testNCONTAINSLSetUninitializedx_
               LVarInitializedx()
            throws NotDefConstraintException, Failure {
```

```
        solverClass solver = new SolverClass();
        ConstraintsConjunction c;
        LSet x = new LSet("x1");
        LVar y = new LVar(3);
        solver.solve(x.contains(y));
        // y=3, y in x
        c = solver.getConstraint();
        assertEquals("[]",c.toString());
 }
```

Come si può notare si è risolto totalmente il vincolo `contains` quindi il constraint estratto dal solver sarà vuoto.

- **ncontains(v_2)**: prende due valori e crea un vincolo di non appartenenza insiemistica, $v_1 \notin v_2$.

  Anche in questo caso il valore di v_2 potrà essere una `LVar` inizializzata, non inizializzata e un object in particolare un intero.

  Vediamo ora un esempio in cui il valore di v_1 è descritto nel caso *1.2.3* mentre il valore di v_2 è una `LVar` non inizializzata.

```
@Test
public void testNCONTAINSLSetInitializedWithLVarUninitialized_
                LVarUninitialized()
          throws NotDefConstraintException, Failure {
    SolverClass solver = new SolverClass();
    ConstraintsConjunction c;
    LVar x0 = new LVar("x0");
    LVar x1 = new LVar("x1");
    LSet x = LSet.empty().ins(x0).ins(x1);
    LVar y = new LVar("y");
    solver.solve(x.ncontains(y));
    //x={x0,x1}, y ncontains x
    c = solver.getConstraint();
    assertEquals("_y neq _x0 AND _y neq _x1",c.toString());
}
```

Come si può notare risolvendo il vincolo ciò che abbiamo è una congiunzione di vincoli ($y \neq x0 \wedge y \neq x1$).

5.6.3 Constraint per `LList`

Per testare tali constraint i casi presi in considerazione sono esattamente quelli utilizzati per testare i vincoli degli `LSet`.

Per le liste logiche però i vincoli da testare sono soltanto quelli di uguaglianza e disuguaglianza.

- **eq(v_2)**: prende due valori v_1 e v_2 e li unifica.

  Vediamo qui di seguito un semplice esempio di test in cui il valore di v_1 è descritto nel caso *1.1* mentre il valore di v_2 è un `LSet` inizializzato con resto:

```
@Test
public void testEQLListUninitializedx_
           LListInitializedWithRest() throws
               NotDefConstraintException, Failure {
    SolverClass solver = new SolverClass();
    Constraint c;
    LList x = new LList("x1");
    LList y = x.ins(1);
    try{
        solver.solve(x.eq(y));
        // y=[1|x], x=y
        c = solver.getConstraint();
        fail("FALLIMENTO!!!!");
     }
     catch(Failure e) {}
   }
```

  Come si può notare questo è un caso in cui la risoluzione fallisce quindi attraverso un blocco `try/cath` viene gestito il fallimento e attraverso il metodo `fail()` viene stampato un messaggio nel caso in cui la solve non fallisce.

- **neq(v_2)**: prende due valori v_1 e v_2 controlla che non unifichino.

  I casi presi in considerazione sono esattamente quelli descritti per il constraint precedente.

  Vediamo ora un esempio di test in cui il valore di v_1 è descritto nel caso *1.2.1* mentre il valore di v_2 è una lista inizializzata con resto:

```
@Test
public void testNEQLListInitializedWithInteger
          _LListWithRest()
     throws NotDefConstraintException, Failure {
    SolverClass solver = new SolverClass();
    Constraint c;
    LList x = LList.empty().ins(1);
    LList y = x.ins(2);
    solver.solve(x.neq(y));
    // x=[1] , y=[2|x], x neq y
    c = solver.getConstraint();
    assertEquals("[]",c.toString());
  }
```

Come si nota facilmente il vincolo viene completamente risolto e il metodo `toString()` restituisce la stringa ″[ ]″.

5.6.4 Constraint per `IntLVar`

I vincoli testati per la classe `IntLVar` sono della forma

$$v_1\texttt{.op(}v_2\texttt{)}$$

dove:

- v_1 è una `IntLVar`,
- `op` è uno degli operatori `eq, neq, lt, ge, le, ge`,
- v_2 cambia in base al constraint considerato.

Di seguito riportiamo la casistica riferita a $v_1$utilizzata per testare i constraint.

- **Casistica 3:**
  1. una `IntLVar` non inizializzata.
     Ad esempio:
     `IntLVar x = new IntLVar();`
  2. una `IntLVar` inizializzata:
     1. con intero.
        Ad esempio:
        `IntLVar x = new IntLVar(2);`

2. con intervallo.
   Ad esempio:
   `IntLVar x = new IntLVar(2,6);`
3. con `IntLVar` inizializzata con intero.
   Ad esempio:
   `IntLVar x1 = new IntLVar(3);`
   `IntLVar x = new IntLVar(x1);`

Si è scelta questa casistica in quanto è intuitivo capire che tutti gl'altri casi possibili rientrano in uno dei casi proposti.

I vincoli sottoposti ai test sono i seguenti:

- **eq(v_2)**: prende due valori v_1 e v_2 e li unifica.

  Per questo vincolo i valori di v_2 sono esattamente quelli descritti nella **casistica 3** a cui è stato aggiunto il caso in cui l'argomento sia un intero.

  Un esempio per il test di tale metodo può essere descritto dal codice seguente che prende in considerazione il caso *3.2.2* e un intero:

```
@Test
public void testEQIntLVarinitializedWithIntervall_Integer()
         throws NotDefConstraintException, Failure {
      SolverClass solver = new SolverClass();
      Constraint c;
      IntLVar x = new IntLVar(8,12);
      try{
          solver.solve(x.eq(1));
          c = solver.getConstraint();
          fail("FALLIMENTO!!!!");
      }
      catch(Failure e) {}
}
```

  Come si può notare la risoluzione fallisce e viene gestita attravero un blocco `try-catch`. Al suo interno si trova il metodo `fail()` che visualizza una stringa nel caso in cui la solve non fallisse come ci aspettiamo.

- **neq**(v_2): prende due valori v_1 e v_2 e controlla che non unifichino.

  I casi presi in considerazione sia per i valori di v_1 che per i valori di v_2 sono esattamente quelli descritti per il constraint precedente.

  Un esempio per il test di tale metodo può essere descritto dal codice seguente che prende in considerazione il caso *3.1* unificandolo con un intero:

```
@Test
    public void testNEQInLVarUninitializedx_
                IntLVarInitializedWithIntervall()
        throws NotDefConstraintException, Failure {
    SolverClass solver = new SolverClass();
    Constraint c;
    IntLVar x = new IntLVar("x1");
    IntLVar y = new IntLVar(2,6);
    solver.solve(x.neq(y));
    c = solver.getConstraint();
   assertEquals("_x1 neq _?",c.toString());
}
```

  Si nota che risolvendo il constraint non viene modificando essendo in una forma irriducibile.

- **lt**(v_2): prende due valori v_1, v_2 e crea il vincolo di minore su interi cioè $v_1 < v_2$.

  Per questo vincolo i valori di v_2 sono esattamente quelli descritti nella **casistica 3** a cui è stato aggiunto il caso in cui l'argomento sia un intero.

  Un esempio per il test di tale metodo può essere descritto dal codice seguente che prende in considerazione il caso *3.1* che viene confrontato con un'altra `IntLVar` non inizializzata.

```
@Test
      public void testLTIntLVarUninitializedx_
                        IntLVarUninitializedy()
              throws NotDefConstraintException, Failure {
      SolverClass solver = new SolverClass();
      ConstraintsConjunction c;
      IntLVar x = new IntLVar("x1");
```

```
        IntLVar y = new IntLVar();
        solver.solve(x.lt(y));
        c = solver.getConstraint();
        assertEquals("_x1 < _?",c.toString());
}
```

Si nota che il vincolo viene risolto e viene creato un `Constraint` in forma irriducibile.

- **gt(v_2)**: prende due valori v_1, v_2 e crea un vincolo di maggiore su interi, cioè $v_1 > v_2$.

  Per questo vincolo i valori di v_2 sono esattamente quelli descritti nella **casistica 3** a cui è stato aggiunto il caso il cui l'argomento sia un intero.

  Un esempio per il test di tale metodo può essere descritto dal codice seguente che prende in considerazione il caso *3.2.2* che viene confrontato con una `IntLVar` inizializzata con un intervallo diverso dal precedente:

```
@Test
public void testLTIntLVarinitializedWithIntervall_
     IntLVarInitializedWithIntervall()
        throws NotDefConstraintException, Failure {
   SolverClass solver = new SolverClass();
   Constraint c;
   InLVar x = new IntLVar(8,12);
   IntLVar y = new IntLVar(2,6);
   try{
     solver.solve(x.lt(y));
     c = solver.getConstraint();
     fail("FALLIMENTO!!!!");
    }
    catch(Failure e) {}
 }
```

Come si può notare la risoluzione fallisce dato che i due intervalli hanno intersezione vuota e viene gestita attravero un blocco `try-catch`. Al suo interno si trova il metodo `fail()` che visualizza una stringa nel caso in cui la solve non fallisse come ci aspettiamo.

5.7 La classe AllTest

Quando le classi di test diventano numerose può essere molto comodo testarle attraverso l'esecuzione di un'unica classe.

Come abbiamo visto nel capitolo riguardante il testing una volta eseguiti i singoli test si intraprende la fase dell'integration testing utilizzando la strategia *bottom-up* in cui vengono integrati i test unit in sottoinsiemi più grandi per poi essere analizzati nel loro insieme.

Nel nostro caso la prima classe testata è stata quella delle `Lvar`, dato che da lei dipendono gli `IntLvar` e viene usata per esaminare i casi presi in considerazioni sia per gli `LSet` e per le `LList`.

Dopodichè stato controllato il buon funzionamento degli insiemi logici dato che estendono la classe degli `IntLSet` per poi procedere al controllo delle altre classi descritte.

Al termine è stata definita una classe che si occupa di eseguire tutti i test con un solo click.

JUnit mette a disposizione la classe `TestSuite` che consente di invocare in modo automatico tutti i metodi di una classe che estende `TestCase`.

Nel nostro caso l'`AllTest.java` è così definito:

```
package JSetL.test;
import junit.framework.Test;
import junit.framework.TestSuite;
  public class AllTests {
   public static Test suite() {
     TestSuite suite = new TestSuite(AllTests.class.getName());
     //$JUnit-BEGIN$
      suite.addTestSuite(LSetTest.class);
      suite.addTestSuite(IntLvarTest.class);
      suite.addTestSuite(LvarTest.class);
      suite.addTestSuite(LvarExprTest.class);
      suite.addTestSuite(LListTest.class);
      suite.addTestSuite(IntLSetTest.class);
      suite.addTestSuite(ConstraintTest.class);
      //$JUnit-END$
      return suite;
   }
 }
```

Com'è possibile notare facilmente all'interno della Suite, viene creato un oggetto di tipo `TestSuite` al quale attraverso il metodo `addTestSuite` si aggiungono tutti i test precedentemente creati, che si vogliono mandare in esecuzione.

JUnit permette di costruire in modo automatico e veloce la classe `AllTest`.

Per fare ciò è sufficiente posizionarsi sul package principale,e scegliere il nome del test (che di default è `AllTest`) e i singoli test da mandare in esecuzione.

Mandando in esecuzione la classe `AllTest` verranno richiamate tutte le classi aggiunte attraverso il metodo `addTestSuite` e saranno testate.

Se l'esecuzione terminerà con successo verrà visualizzata una barra verde, in caso contrario verranno elencate le classi errate con le specifiche dell'errore trovato.

Capitolo 6

Conclusioni e lavori futuri

Il lavoro di tesi ha avuto come scopo quello di progettare e realizzare un insieme di test per eseguire il testing delle principali classi di JSetL.

Come prima cosa si è creata una classe di test per ognuna delle classi di JSetL da testare; quindi per ogni metodo delle singole classi si è studiata una casistica adatta per eseguire i test.

Per fare ciò si è diviso il dominio di input in classi di equivalenza in modo da coprire tutti i casi possibili e si è effettuato il test di tutti i metodi utilizzando la casistica proposta.

Nel nostro caso la prima classe testata è stata quella delle `LVar`, dato che da essa sono derivate le `IntLvar` ed è utilizzata per esaminare i casi presi in considerazioni sia per gli `LSet`, che per le `LList` e per i `Constraint`.

Dopodichè si è controllato il buon funzionamento degli insiemi logici (classe `LSet`) e della classe derivata degli `IntLSet`, per poi procedere al controllo delle altre classi descritte; quindi si è testata la classe `Constraint` che contiene al suo interno tutti i test per i vincoli delle classi precedenti. Al termine si è definita una classe di nome `AllTest` che si occupa di eseguire tutti i test con un solo click.

In futuro si potrebbero testare i metodi presenti nelle altre classi di JSetL non trattate, ma comunque di fondamentale importanza, come ad esempio le classi `SolverClass` e `Constraint`.

Un altro aspetto che non si è toccato in questo elaborato sono sicuramente i *criteri di copertura* del testing.

Gli strumenti per il **code coverage** permettono di sapere quali parti di codice sono state eseguite e quali sono state saltate e sono quindi molto importanti per avere risultati attendibili durante le sessioni di unit testing.

Infatti se il codice è stato coperto interamente durante i test si può essere sicuri che l'intera applicazione funzioni a dovere, mentre se parte del

codice non fosse stata coperta è possibile che la sua esecuzione in situazioni particolari porti anomalie.

Un lavoro da fare per completare questa tesi sarebbe dunque il controllo della efficacia del testing realizzato. Per fare ciò si possono utilizzare diversi strumenti come ad esempio **Emma** che è un tool free per la valutazione della copertura di codice Java oppure **codecover** che si occupa di controllare anche la ridondanza dei casi di test.

Bibliografia

[1] Ercole Colonese
Collaudo del software
Versione 2.1 Giugno 2007

[2] Fausto Fasano
Testing - Corso di ingegneria del software
`http://serviziweb.unimol.it/unimol/allegati/docenti/2786/materiale/11.testing`

[3] Ian Sommerville
Ingegneria del Software
Addison Wesley, 2005

[4] Cadoli, Lenzerini, Naggar, Schaerd
Fondamenti della progettazione dei programmi. Principi, tecniche e loro applicazioni in C++
CittaStudiEdizioni di UTET Libreria, 1997.

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

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

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

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