

Progettazione e implementazione di un
programma per la generazione stocastica
di profili di rilevanza
Relazione sull'attività di laboratorio

Filippo Cussigh

Settembre 2020

Indice

Indice	1
1 Introduzione	3
1.1 Ambito IR	3
1.2 Ambito Object-Oriented	4
1.3 Obiettivi	4
2 Framework jMetal e algoritmo NSGA-II	6
2.1 Architettura di jMetal	6
2.1.1 Interfaccia Solution	7
2.1.2 Interfaccia Problem	7
2.1.3 Interfacce Algorithm e Operator	8
2.1.4 Implementazioni astratte	8
2.2 NSGA-II	9
3 Relevance List: il caso binario	12
3.1 RLBinarySolution	15
3.2 RLBinarySolutionFactory	18
3.3 MetricEvaluator	18
3.4 RLBinaryProblem	19
3.5 RLNSGAI e RLNSGAIIBuilder	21
3.6 Crossover Operator e Mutation Operator	22
3.7 Altre Classi	22
4 Il programma e l'esecuzione	24
4.1 Il flusso del programma	24
4.2 I parametri	24
4.3 L'esecuzione da linea di comando	25
4.4 Dettagli implementativi	26

5	Relevance List Generics: il caso generico	27
5.1	RLAbstractSolution<S, V>	27
5.2	RLAbstractSolutionFactory<T, V>	28
5.3	MetricEvaluator	29
5.4	RLAbstractProblem<T, V>	29
5.5	RLNSGAI, Builder e operatori	31
5.6	La classe Program e l'esecuzione	31
5.7	Esempi di estensione del framework	34
5.7.1	Aggiunta di una nuova metrica di valutazione	34
5.7.2	Aggiunta di un nuovo operatore di mutazione	35
6	Conclusioni e sviluppi futuri	37
6.1	Risultati ottenuti	37
6.2	Sviluppi futuri	38
	Bibliografia	40

Capitolo 1

Introduzione

L'attività di laboratorio descritta in questa relazione tratta la realizzazione di un programma per la generazione stocastica di profili di rilevanza, che rispetti i principi della programmazione object-oriented.

1.1 Ambito IR

Il problema risolto dal programma si colloca nell'ambito della misura di efficacia dei sistemi di *Information Retrieval*. Un sistema di IR ha il compito di recuperare materiale di natura non strutturata all'interno di una collezione più grande, al fine di soddisfare un bisogno informativo dell'utente: l'utente sottopone una interrogazione al sistema, che restituisce un insieme di documenti. Come profilo di rilevanza si intende la lista ordinata dei singoli valori di rilevanza dei documenti, definibile intuitivamente come quanto il documento è pertinente rispetto al bisogno informativo dell'utente. L'ordinamento della lista è determinato dall'ordine di recupero dei documenti, dunque in prima posizione è contenuta l'informazione "quanto è rilevante il primo documento recuperato" e così via per le successive posizioni. Il profilo di rilevanza può venire quindi valutato secondo diverse metriche per dare una misura della qualità del risultato ottenuto. In particolare il programma affronta il problema di generare in modo stocastico dei profili di rilevanza simulati, dunque non ottenuti a partire dal risultato di una reale interrogazione, che producano un determinato valore target secondo le metriche stabilite dall'utente. Il problema viene discusso estensivamente nell'articolo di Roitero et al. [1].

1.2 Ambito Object-Oriented

L'ambito su cui verte principalmente l'attività è tuttavia la progettazione e la programmazione Object-Oriented. Il programma deve rispettare infatti i principi di buona programmazione OO, in particolare in relazione all'utilizzo di framework OO e come andare ad utilizzare le sue classi. In particolare viene posto il focus su:

- *estendere, non modificare*, le classi del framework non devono essere in alcun modo modificate, bensì estese/implementate dalle classi del nuovo programma. Le classi principali realizzate dunque ereditano dal *core* principale del framework;
- *coesione di classe*, oltre alla già presente organizzazione delle classi base del framework, ne sono state introdotte di nuove dove è stata ritenuta un'opzione migliore delegare determinate funzionalità a classi dedicate, anziché mantenere tutto il codice all'interno delle classi originali complicando la comprensione e la modificabilità del progetto;
- *estensibilità*, il programma mira alla possibilità di essere esteso aggiungendo funzionalità e modalità d'uso senza stravolgerne la struttura;
- *confini di incapsulamento e dipendenze*, è stato strutturato il progetto evitando dipendenze che violino i confini dell'incapsulamento, fornendo un'interfaccia di metodi sufficiente a fornire un appropriato scambio di messaggi tra le classi.

È stato inoltre fatto uso di design pattern per la risoluzione di determinate dinamiche tra le classi.

1.3 Obiettivi

L'obiettivo del progetto è utilizzare il framework orientato agli oggetti jMetal [2, 3] per la creazione di un programma che risolva il problema proposto, rispettando i principi dell'object-oriented design. L'algoritmo risolutivo è stato basato sull'algoritmo genetico NSGA-II, originariamente pubblicato da Deb et al. [4], che fa uso di tecniche metaeuristiche per la risoluzione di problemi di ottimizzazione multi-obiettivo. Il framework jMetal mette a disposizione un'architettura, descritta in seguito, per l'impostazione e l'esecuzione di tali algoritmi, di cui è già fornita una implementazione di base. In particolare è stato utilizzato il framework nella sua ultima versione disponibile 5.10, implementata in linguaggio Java.

Il programma è stato realizzato per risolvere l'istanza del problema in cui il valore di rilevanza degli elementi del profilo è espresso come binario, 0 o 1. La versione interamente funzionante è stata poi estesa realizzando un'insieme di classi astratte su cui si adatta la precedente soluzione e che permettano l'implementazione anche su diversi tipi di profilo. È stata inoltre fornita una implementazione parziale per il caso di valori di rilevanza espressi tramite interi.

Capitolo 2

Framework jMetal e algoritmo NSGA-II

2.1 Architettura di jMetal

Il framework jMetal fornisce un'ambiente per la risoluzione di problemi di ottimizzazione basati su tecniche meta-euristiche. Nel caso specifico preso in esame, il problema di ottimizzazione consiste nel minimizzare una certa misurazione sul profilo di rilevanza rispetto a un determinato valore target, entrambi forniti in input. Ad esempio, data la funzione *AveragePrecision* e un valore target x , viene cercata la soluzione la cui precisione media si avvicina più a x .

L'architettura di base del framework jMetal è costituita da quattro interfacce visibili nel seguente diagramma uml: Solution, Problem, Algorithm e Operator (Figura 2.1]). Intuitivamente il funzionamento dell'insieme di classi è che un Algoritmo risolve un Problema, che ha il compito di creare e valutare un insieme di Soluzioni, facendo uso di Operatori che manipolano tali soluzioni. Successivamente verrà descritto come è stata ampliata questa struttura, in particolare per riorganizzare le funzioni svolte dal problema.

A seguire una breve descrizione delle quattro interfacce basate sulla documentazione ufficiale.

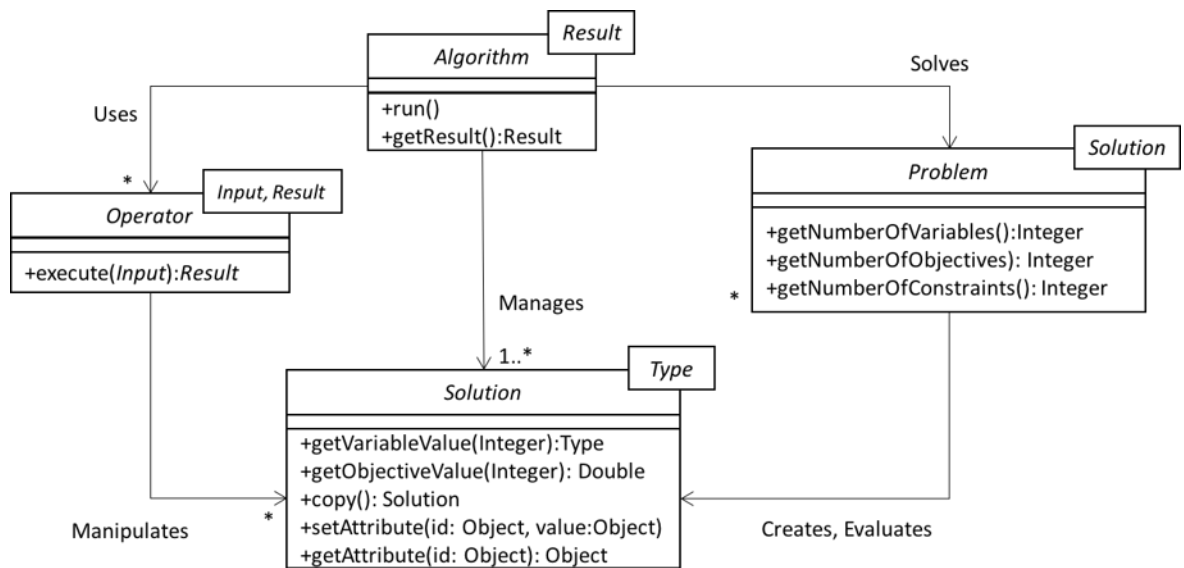


Figura 2.1: Diagramma UML delle interfacce base di jMetal

2.1.1 Interfaccia Solution

L'interfaccia **Solution** rappresenta il tipo di soluzione che l'algoritmo va a computare. Dalla rappresentazione della soluzione scelta dipendono poi le funzioni delle altre classi che le vanno a manipolare. La soluzione è in particolare caratterizzata dagli insiemi dei suoi valori di variabile e di obiettivo, che rispecchiano i corrispettivi elementi di un problema multi-obiettivo.

JMetal mette già a disposizione delle possibili estensioni dell'interfaccia per soluzioni con variabili di tipo **Integer**, **Double** e **BinarySet**, come visibile nel diagramma UML in figura 2.2, e delle implementazioni di base per ciascuna.

All'interno della classe è inoltre presente la gestione degli attributi, campi aggiuntivi utilizzati dagli algoritmi per la codifica di informazioni necessarie all'esecuzione.

2.1.2 Interfaccia Problem

La classe **Problem**, visibile nel diagramma UML in figura 2.3, ha lo scopo di gestire la generazione delle nuove soluzioni e successivamente la valutazione, dopo che esse sono state manipolate dall'algoritmo e i suoi operatori.

Gli altri metodi principali dell'interfaccia sono i getter per il numero di valori di variabile, obiettivo e vincolo con cui vengono definite le soluzioni. I valori di vincolo, i *Constraints*, vengono utilizzati per la valutazione delle

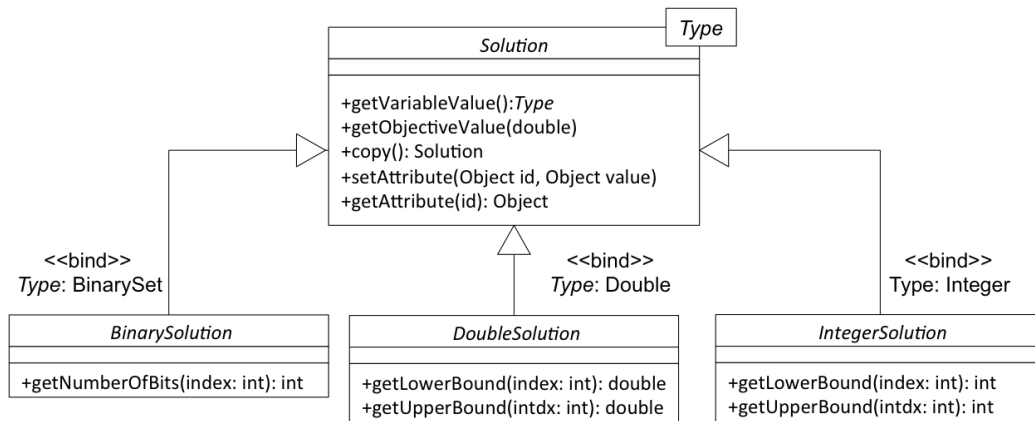


Figura 2.2: Diagramma UML dell'interfaccia Solution

soluzioni nel caso esistano condizioni per cui una soluzione perda il confronto con un'altra. Verrà mostrato in seguito un esempio pratico dell'utilizzo dei vincoli, in quanto sono stati utilizzati per limitare il numero di documenti rilevanti di un profilo.

2.1.3 Interfacce Algorithm e Operator

Le interfacce **Algorithm** e **Operator** sono molto semplici e presentano ben pochi elementi. **Algorithm** richiede di implementare un metodo di esecuzione *run* e un metodo per la restituzione di un risultato di tipo **Result**. JMetal mette a disposizione implementazioni per una vasta gamma di algoritmi multi-obiettivo e a obiettivo singolo. In questo caso verrà utilizzata la implementazione dell'algoritmo NSGA-II, estesa per aggiungere alcuni casi di terminazione necessari per la risoluzione del problema preso in esame.

L'interfaccia **Operator** comprende invece un solo metodo *execute* in cui viene implementata la funzione dell'operatore. Gli algoritmi utilizzano degli operatori per la definizione e l'esecuzione dei determinati sotto-processi delle tecniche meta-euristiche.

2.1.4 Implementazioni astratte

All'interno del framework vengono già fornite delle implementazioni astratte di base delle interfacce, che introducono gli attributi necessari per la codifica delle informazioni e i getter/setter aggiuntivi. È stato scelto di estendere tali classi astratte, in particolare **AbstractSolution** e **AbstractGenericProblem**, per la realizzazione delle classi del progetto.

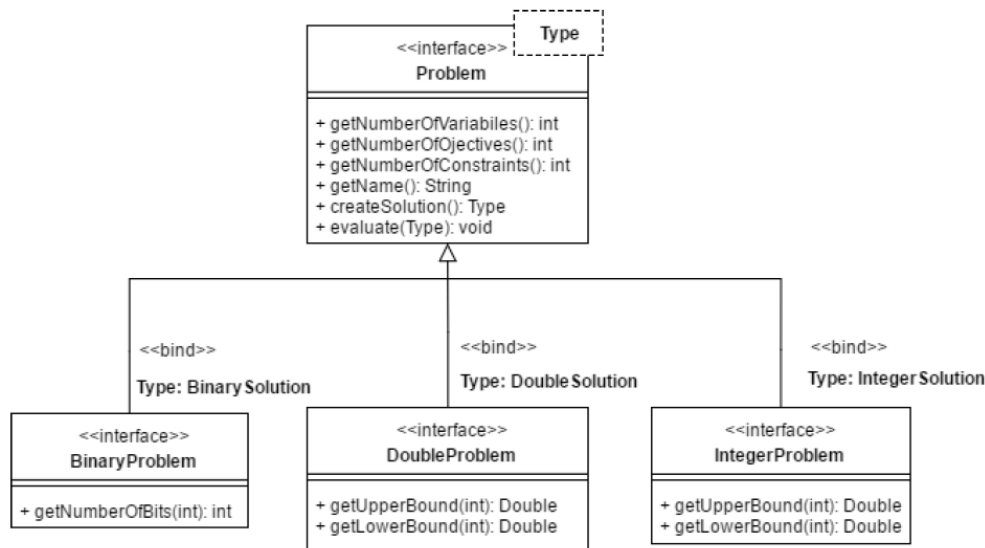


Figura 2.3: Diagramma UML dell'interfaccia Problem

2.2 NSGA-II

NSGA-II è l'algoritmo genetico utilizzato all'interno del programma. Un algoritmo genetico è un algoritmo euristico che sfrutta principi ispirati alla selezione naturale e all'evoluzione per la risoluzione dei problemi di ottimizzazione.

Il normale flusso di esecuzione di un algoritmo genetico si svolge secondo i seguenti passi:

1. generazione di un primo insieme di soluzioni, chiamato popolazione
2. valutazione delle soluzioni secondo una funzione di fitness
3. selezione delle soluzioni migliori
4. operazione di crossover per la generazione delle soluzioni figlie
5. operazione di mutazione sulle nuove soluzioni
6. creazione della nuova popolazione, la generazione successiva, a partire dalle nuove soluzioni
7. ripetizione a partire dal punto 2 sulla nuova popolazione fino al raggiungimento della condizione di terminazione

Con operazione di crossover viene intesa la procedura di generazione di nuove soluzioni a partire da più soluzioni esistenti, mentre la procedura di mutazione consiste nella variazione casuale dei valori di una soluzione per evitare che la popolazione si stagni su ottimi locali. Già dalla descrizione di un generico algoritmo genetico si può osservare la correlazione con l'architettura di base di jMetal: l'*algoritmo* manipola una popolazione di *soluzioni*, facendo uso di *operatori* quali selezione(3), crossover(4) e mutazione(5).

L'algoritmo NSGA-II estende la struttura di base descritta, introducendo l'ordinamento per non dominazione e il concetto di elitismo. La popolazione prima di passare attraverso l'operazione di selezione viene ordinata per non dominazione e a ciascuna soluzione viene assegnato il rango di non dominazione, ovvero l'indice del fronte di pareto di soluzioni non dominate. L'elitismo invece consiste nel preservare una parte delle soluzioni migliori, ovvero quelle con il miglior risultato di fitness, della precedente generazione, copiandole senza mutazioni all'interno della nuova generazione; ciò previene da possibilità di perdere soluzioni ottime.

Il processo dunque a partire da una popolazione di dimensione N , genera altrettanti N figli formando una popolazione complessiva di $2N$ elementi; la popolazione viene dunque ordinata per non dominazione e, a partire dal fronte di Pareto di indice minimo, vengono prese le N soluzioni migliori per formare la nuova generazione. Poiché tutti i genitori vengono inclusi nel calcolo dell'ordinamento, viene soddisfatto il concetto di elitismo. Nella figura 2.4 è visibile il flusso d'esecuzione dell'algoritmo NSGA-II.

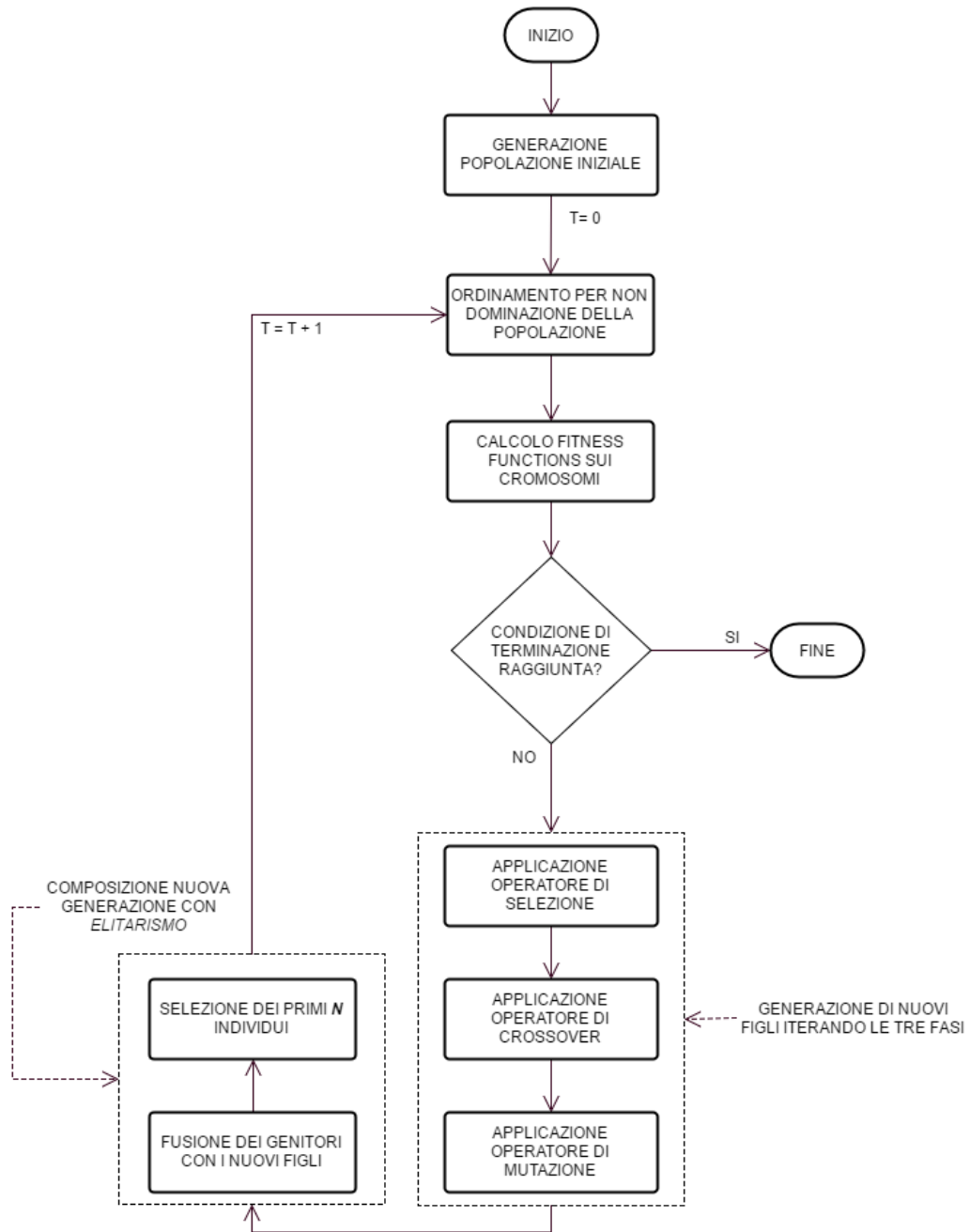


Figura 2.4: Diagramma d'esecuzione dell'algoritmo NSGA-II, tratto da Soprano [5]

Capitolo 3

Relevance List: il caso binario

In questo capitolo verrà presentato il programma **Relevance List** che affronta il problema binario, ovvero quello in cui il valore di rilevanza è esprimibile come 0 o come 1. Il programma dunque prenderà in input una serie di parametri che descrivono le caratteristiche del profilo da calcolare e restituirà il miglior profilo ottenuto sotto forma di vettore binario.

La soluzione proposta estende le implementazioni astratte fornite da jMetal per le quattro classi base, e introduce due nuove classi che verranno successivamente descritte: il **Metric Evaluator** per la delega della valutazione delle soluzioni e la **Solution Factory** per delegare la generazione di soluzioni. Idealmente queste funzioni sarebbero potute essere svolte direttamente dalla classe che rappresenta il problema, come visibile dall'interfaccia **Problem** che contiene i metodi *createSolution* e *evaluateSolution*, tuttavia ciò comporta un'eccessiva quantità di codice all'interno della classe, numerosi valori da passare al problema e soprattutto scarsa adattabilità nel caso si vogliano aggiungere nuove funzioni di valutazione.

Nel diagramma UML in figura 3.1 è possibile vedere le relazioni tra le principali classi introdotte nel progetto e le corrispondenze con il core dell'architettura jMetal precedentemente discusso; nei diagrammi delle sezioni seguenti verrà mostrato più specificatamente da quali classi del framework ereditano, e che sono state omesse in questo diagramma per facilitarne la leggibilità e l'organizzazione. In figura 3.2 è visibile anche la struttura dei package.

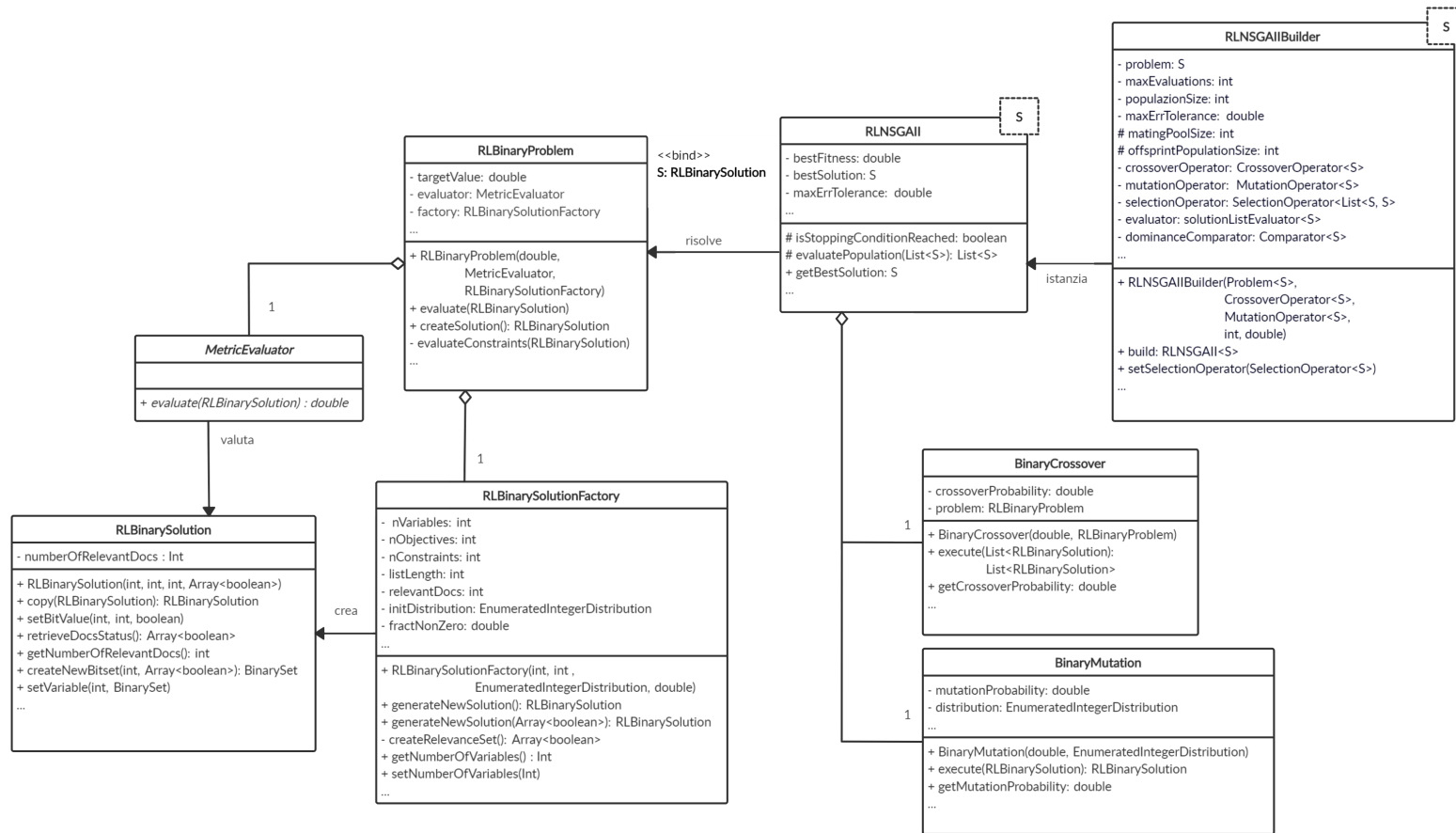


Figura 3.1: Diagramma UML delle classi principali di Relevance List

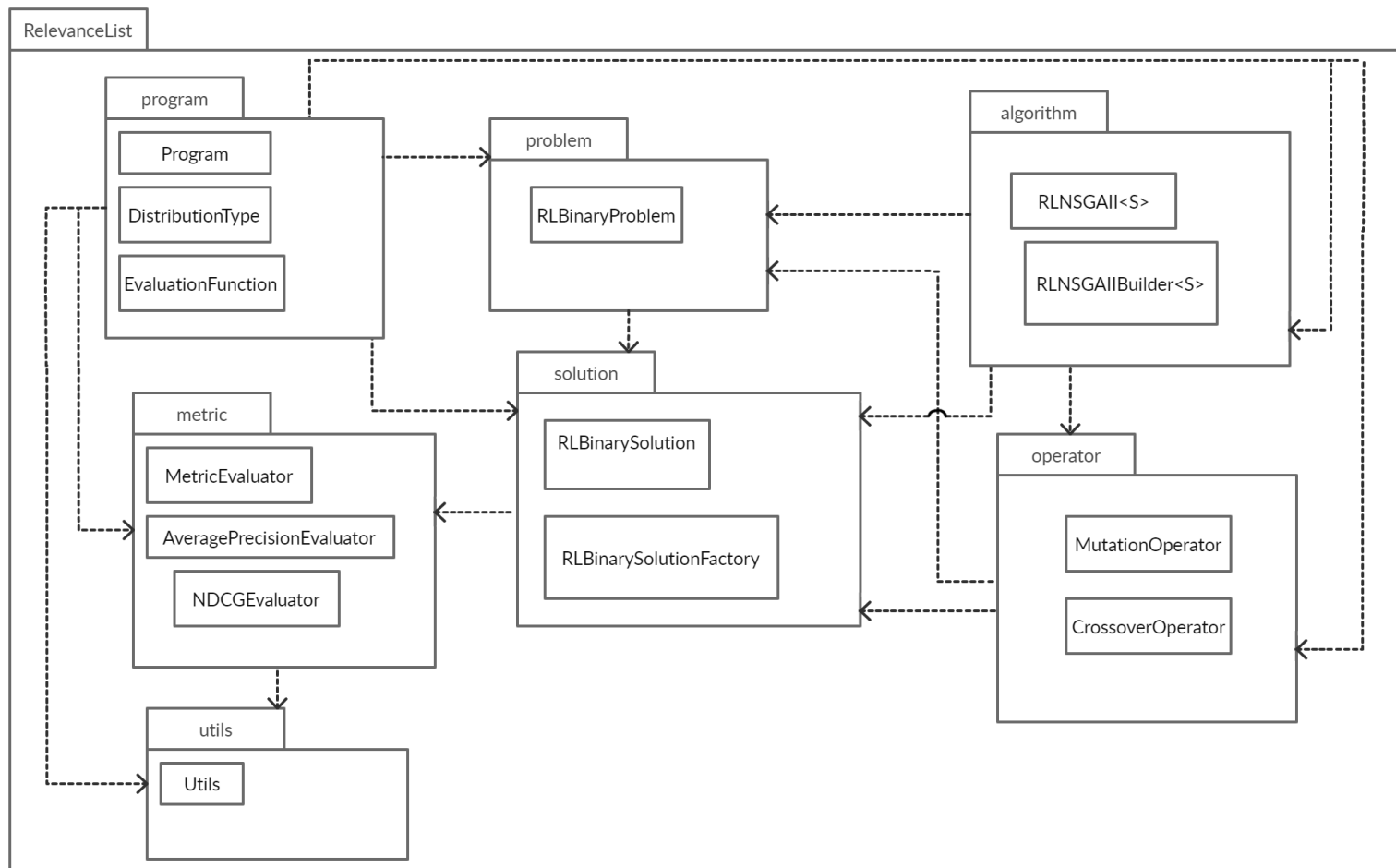


Figura 3.2: Diagramma di Package di Relevance List

Si possono osservare in figura le seguenti classi:

- **RLBinaryProblem**: classe che costituisce il problema e che intuitivamente corrisponde a un'implementazione dell'interfaccia **Problem** di **Jmetal**;
- **RLNSGAI**: l'algoritmo risolutivo;
- **BinaryCrossover**: operatore di crossover;
- **BinaryMutation**: operatore di mutazione;
- **MetricEvaluator**: classe astratta che rappresenta il valutatore delle soluzioni;
- **RLBinarySolution**: classe che rappresenta una soluzione del problema;
- **RLBinarySolutionFactory**: classe delegata alla generazione di soluzioni;
- **RLNSGAIBuilder**: classe che crea un'istanza dell'algoritmo **RLNSGAI**.

Come precedentemente accennato, le classi indicate non implementano direttamente le interfacce di **jMetal**, ma fanno uso delle implementazioni di base già fornite ereditando da esse.

3.1 **RLBinarySolution**

RLBinarySolution è la classe dedicata alla rappresentazione delle soluzioni. Come visibile dalla figura 3.3, la classe estende l'implementazione di soluzione astratta di tipo generico già presente nel framework **AbstractSolution<T>** e implementa l'interfaccia per le soluzioni binarie **BinarySolution**. Il tipo di dato con cui vengono espresse le variabili è il **BinarySet**, una classe già implementata all'interno del package con cui vengono rappresentati vettori di bit e che fornisce i metodi per la loro corretta manipolazione. Come si può osservare dalla figura e dai dati della classe **AbstractSolution<T>**, **BinarySet** è il tipo di dato che corrisponde al generico **T** della classe astratta.

La soluzione per il problema che si vuole risolvere comprende:

- una variabile, che rappresenta il profilo di rilevanza come **BinarySet**;
- un obiettivo, che rappresenta l'errore relativo calcolato come differenza tra il valore *double* della funzione di valutazione calcolata sul profilo e il valore target definito dal problema;

- un vincolo (constraint), che è un valore *double* che viene valutato come la differenza tra il numero di valori con rilevanza 1 che ci sono nel profilo e il numero di documenti rilevanti presenti nel topic considerato.

Il profilo di rilevanza che si sta considerando rappresenta idealmente il risultato in termini di rilevanza di una query su un topic, per il quale esistono un numero n di documenti rilevanti nell'insieme di ricerca. Il profilo dunque non può contenere più di n valori 1, in quanto significherebbe che sono stati recuperati più documenti rilevanti di quanti ne esistano effettivamente per quel topic. Durante le operazioni svolte dagli algoritmi genetici non viene tenuto conto di questo fatto, bensì vengono generati profili che non sono considerabili validi. Al momento della valutazione di una soluzione viene aggiornato il valore di constraint, se tal valore risulta negativo si considera violato il vincolo. Quando l'algoritmo genetico confronta due soluzioni, la soluzione con meno vincoli violati domina l'altra; siccome la popolazione di partenza viene formata interamente da soluzioni valide, tutte le soluzioni generate dalla prima iterazione che violano il vincolo perderanno il confronto con le soluzioni iniziali e verranno dunque scartate.

Il costruttore della classe comprende come parametri il numero di variabili, obiettivi, vincoli e un array di booleani per la costruzione del **BinarySet** dell'unica variabile. Siccome sono già state fissate per il problema le "dimensioni" della soluzione, si potrebbero eliminare tali parametri dal costruttore e passare direttamente 1 per ciascuno al costruttore della sopraclasse, tuttavia si è scelto di lasciare il costruttore flessibile lasciando alla **Solution Factory** il compito di passare queste informazioni.

All'interno della classe è presente anche un valore privato *numberOfRelevantDocs* che indica il numero di documenti rilevanti recuperati rappresentati dal profilo. Modificando i valori delle variabili tramite i setter presenti, il valore viene mantenuto correttamente. Il metodo *getNumberOfRelevantDocs* restituisce tale numero, anziché ricalcolarlo in base alla variabile relativa al profilo; sebbene l'opzione di ricalcolare il valore sia la più sicura in termini di consistenza, è stata scelta l'altra implementazione per questioni di efficienza.

Si noti inoltre che la dimensione del **BinarySet** dell'unica variabile determina già la lunghezza del profilo, dunque non è necessario inserire esplicitamente questo attributo.

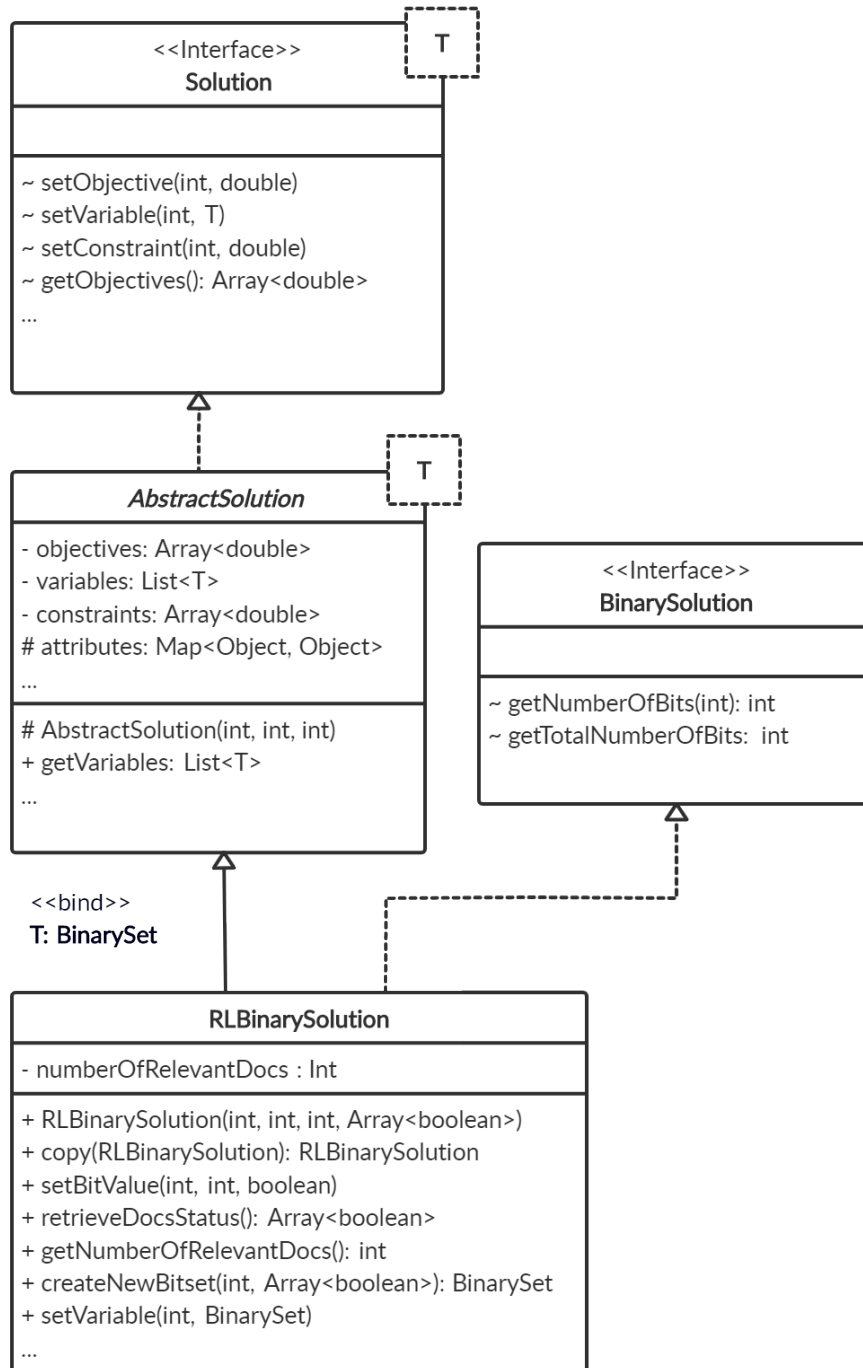


Figura 3.3: Diagramma UML di classe di **RLBinarySolution**

3.2 RLBinarySolutionFactory

La classe **RLBinarySolutionFactory**, visibile nel diagramma UML in figura 3.1, ha lo scopo unico di generare oggetti di tipo **RLBinarySolution**. I parametri necessari per creare le soluzioni sono i seguenti:

- `listLength`, la lunghezza del profilo di rilevanza della soluzione;
- `relevantDocs`, il numero di documenti rilevanti presenti per il topic considerato;
- `initDistribution`, la distribuzione di probabilità con cui determinare come distribuire i valori del profilo di una nuova soluzione;
- `fractNonZero`, il rapporto tra il numero di documenti rilevanti delle nuove soluzioni da generare e *relevantDocs*. Se posto uguale a 1 significa che il profilo delle soluzioni create conterrà *relevantDocs* elementi diversi da 0 (dunque in questo caso binario, uguali a 1).

Inoltre sono presenti i valori per quanto riguarda il numero di variabili, obiettivi e vincoli delle soluzioni da generare, che sono già stati discussi precedentemente.

La classe mette a disposizione due metodi per la generazione delle soluzioni. Il primo, senza parametri, utilizza la distribuzione di probabilità per determinare quali valori del profilo inizializzare a 1 o a 0, avvalendosi della sottoprocedura *createRelevanceSet* che restituisce un array di booleani con cui chiamare il costruttore di **RLBinarySolution**. L'altro costruttore riceve come parametro già l'array, dunque non ha bisogno dell'altro metodo.

Il metodo *createRelevanceSet* utilizza il valore *fractNonZero* e *relevantDocs* per determinare quanti valori 1 inserire. Il valore *fractNonZero* può venire (al 50% dei casi) precedentemente perturbato di una percentuale casuale per introdurre varietà nelle soluzioni. Dopodiché viene usata la distribuzione di probabilità per determinare quali valori del profilo inizializzare a 1.

I restanti metodi della classe sono *getter* e *setter* delle dimensioni della soluzione e *getter* degli altri valori.

3.3 MetricEvaluator

MetricEvaluator è una classe astratta che rappresenta una potenziale metrica con cui valutare una soluzione. In questo caso sono presenti due specializzazioni, la *Average Precision* e la *Normalized Discounted Cumulative Gain*,

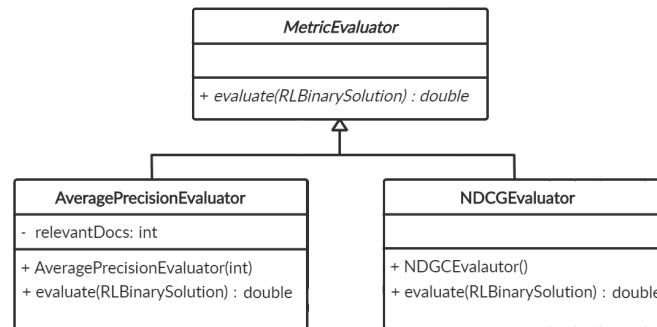


Figura 3.4: Diagramma UML di classe di MetricEvaluator

come mostrato nel diagramma UML in figura 3.4. L'unico metodo presente all'infuori del costruttore è *evaluate*, che restituisce un valore di tipo **double** e che le sotto-classi devono implementare. Si noti che tale valore non è l'*obiettivo* di un oggetto di tipo **RLBinarySolution**, bensì la effettiva valutazione della metrica. L'obiettivo della soluzione è la minimizzazione della differenza tra il valore della metrica e il valore di target contenuto nel problema. Il metodo *evaluate* dunque non va ad alterare lo stato della soluzione, ma ne valuta semplicemente il profilo di rilevanza; è compito dell'istanza di **RLBinaryProblem** utilizzare il valore ottenuto per il calcolo del valore obiettivo della soluzione e aggiornarlo debitamente.

3.4 RLBinaryProblem

La classe **RLBinaryProblem**, visibile nel diagramma UML in figura 3.5, contiene i metodi di creazione di soluzioni e valutazione delle soluzioni, che la classe dedicata all'algoritmo chiama durante l'esecuzione. Precedentemente è stato descritto come entrambe le operazioni facciano parte del flusso d'esecuzione degli algoritmi genetici. In questo caso le procedure vengono delegate in parte ai due oggetti di tipo **MetricEvaluator** e **RLBinarySolutionFactory**, che vengono passati tramite costruttore e utilizzati all'interno dei metodi. L'altro parametro passato nel costruttore è il valore *targetValue*, ovvero la valutazione che devono puntare a raggiungere le soluzioni. Il problema di ottimizzazione consiste nella minimizzazione della differenza tra il *targetValue* e la valutazione che l'oggetto *evaluator* compie sulla soluzione, tale valore viene conservato all'interno dell'unico obiettivo della soluzione. Il metodo *evaluate* chiama l'omonimo metodo di *evaluator* sulla soluzione che deve valutare, ricevendo un valore di tipo *double* che rappresenta il valore

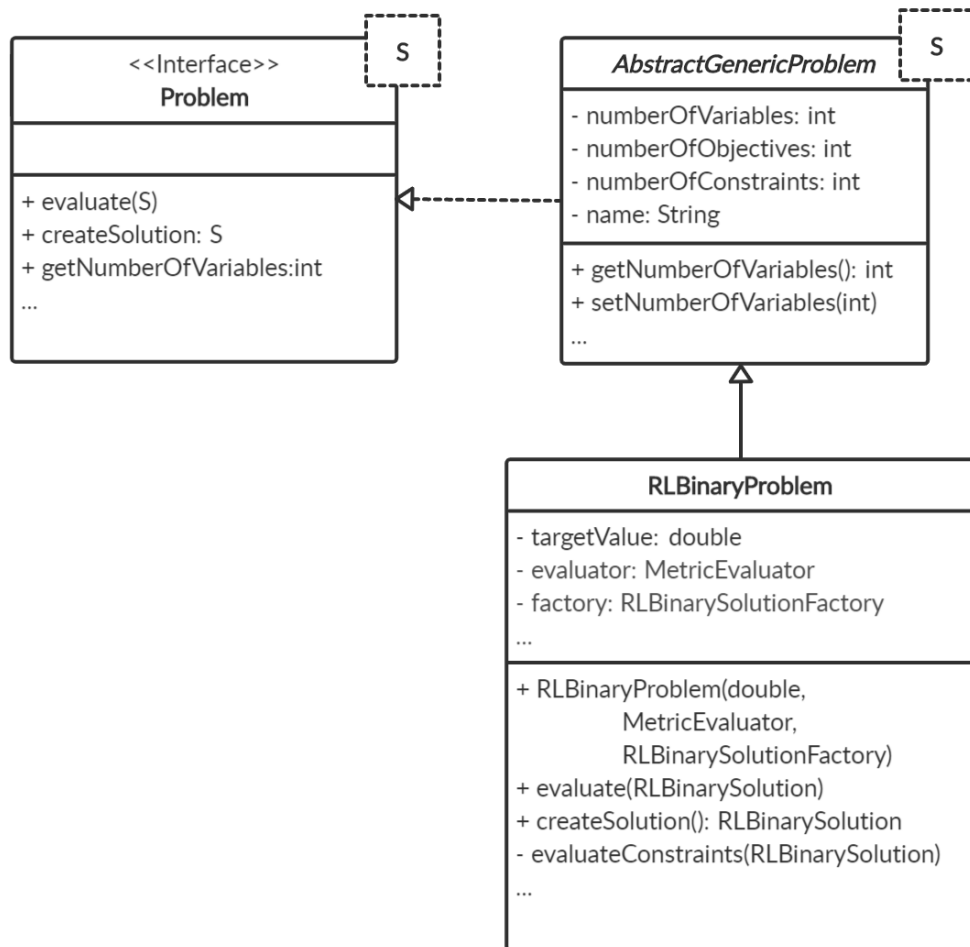


Figura 3.5: Diagramma UML di classe di RLBinaryProblem

della metrica da calcolare; dopodiché calcola la differenza con il valore target, calcola le violazioni dei vincoli tramite il sotto-metodo *evaluateConstraints* e aggiorna variabile e obiettivo della soluzione. Si noti che *evaluator* è un oggetto di tipo astratto **MetricEvaluator**, dunque l'istanza di **RLBinaryProblem** non conosce quale metrica stia calcolando sulla soluzione, ma è interessato solo alla differenza rispetto al *targetValue*; dunque comunica solamente con la classe base usufruendo del suo metodo. Questa struttura permette di aggiungere nuove metriche con estrema semplicità estendendo la classe **MetricEvaluator** con una nuova specializzazione, senza modificare nulla all'interno del problema.

3.5 RLNSGAII e RLNSGAIIBuilder

La classe **RLNSGAII** è una sotto-classe di **NSGAII**, che è già presente nel framework e implementa l'algoritmo di risoluzione necessario. L'unica modifica effettuata consiste nell'*override* del metodo che determina la condizione di terminazione dell'algoritmo: oltre a basarsi su un numero massimo di valutazioni, l'algoritmo si interrompe anche se raggiunge una soluzione sufficientemente buona; se il valore di target da raggiungere ha un certo numero di cifre decimali, non ha senso cercare una soluzione con valore obiettivo con più cifre decimali.

Una volta istanziato l'algoritmo è sufficiente chiamare il comando *run*, presente nell'interfaccia di **Algorithm** e già implementato nelle classe **NSGAII**, e chiamare poi il metodo *getResult* per ottenere il fronte pareto di soluzioni ottime. È stato aggiunto un metodo *getBestSolution* che restituisce una singola soluzione, in quanto per via della natura del problema non multi-obiettivo esiste una singola soluzione migliore di tutte le altre.

Per semplificare l'istanziatura della classe è stata introdotta anche la classe **RLNSGAIIBuilder**. È stato scelto per convenzione di seguire il modello di builder per NSGA-II già presente nel framework, preservando quali elementi inserire all'interno del costruttore direttamente e quali modificare tramite setter; tuttavia siccome sono state rimosse delle funzionalità non più permesse, la classe **RLNSGAIIBuilder** non eredita da altri builder ma implementa direttamente l'interfaccia **AlgorithmBuilder** (di cui è stato omesso il diagramma in quanto contiene solo la segnatura di un metodo *build* già descritto).

Una volta istanziata la classe **RLNSGAIIBuilder**, è sufficiente chiamare il suo metodo *build* per ottenere un'istanza completa di **RLNSGAII** pronta per essere lanciata. I parametri che vengono passati nel costruttore di **RLNSGAIIBuilder** sono:

- l'istanza del problema;
- l'operatore di crossover;
- l'operatore di mutazione;
- la grandezza della popolazione;
- la tolleranza massima dell'errore.

3.6 Crossover Operator e Mutation Operator

Gli operatori di crossover e mutation definiti per questo problema implementano le rispettive interfacce, che a loro volta implementano l'interfaccia **Operator** come visibile in figura 3.6. L'operatore di crossover deve definire quanti sono i genitori per operazione e quanti sono i figli generati, in questo caso sono entrambi due. Le due soluzioni figlie vengono computate la prima facendo l'*and* dei vettori di rilevanza (che si ricorda sono contenuti nella prima variabile della soluzione), mentre la seconda eseguendo l'*or*. Il valore di probabilità determina se l'operazione di crossover viene effettuata o meno; in caso non venga eseguita vengono restituiti i due genitori. Il costruttore richiede l'istanza del problema con cui generare le nuove soluzioni e la probabilità di crossover.

L'operatore di mutation riceve anch'esso dal costruttore una probabilità, per determinare se eseguire o meno l'operazione, e una distribuzione di probabilità. Nel caso venga sorteggiato di eseguire l'operazione, viene scelta casualmente un'operazione di *swap* o di *sum*. L'operazione di *swap* sceglie tramite la distribuzione di probabilità due indici del profilo di rilevanza e ne scambia i valori. L'operazione di *sum* sceglie allo stesso modo l'indice di un valore e ci aggiunge un valore; siccome si tratta di un caso binario viene semplicemente impostato a 1 quel bit.

Le operazioni degli operatori vengono implementate all'interno del metodo *execute*, che viene richiesto dall'originale interfaccia *Operator*.

3.7 Altre Classi

Le classi rimanenti all'interno del programma, visibili nel diagramma UML in figura 3.2, sono la classe **Program** che contiene il *main* da eseguire al lancio del programma e una classe **Utils** che contiene alcune funzioni matematiche comuni. Inoltre sono presenti due classi **DistributionType**

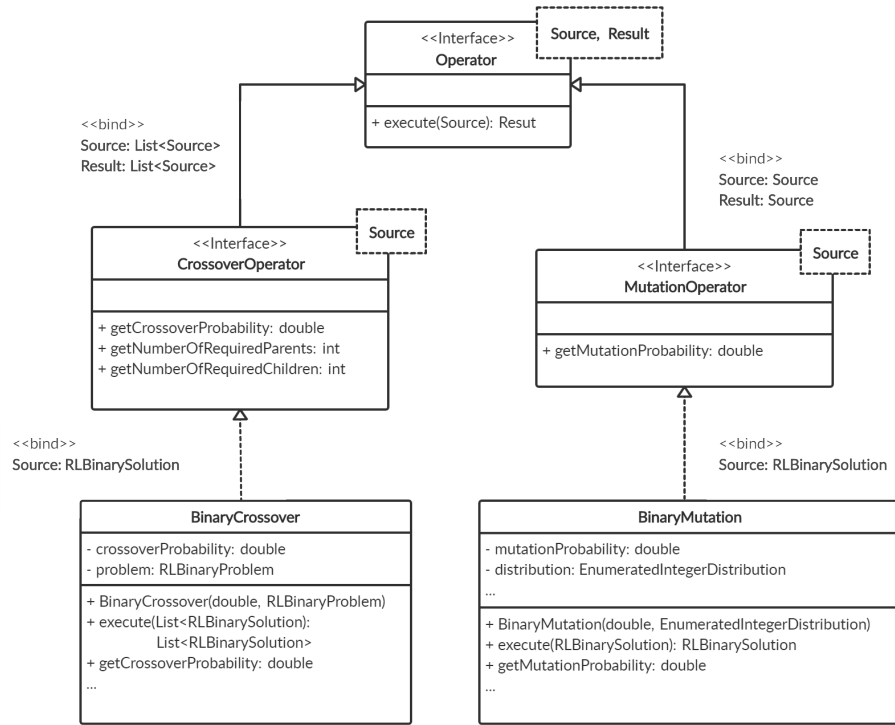


Figura 3.6: Diagrammi UML di classe degli operatori

e **EvaluationFunction**, composte solamente da un'enumerazione per facilitare l'operazione di controllo degli input. Più precisamente i valori delle enumerazioni sono:

- `DistributionType{uniform, geometric}`
- `EvaluationFunction{avgPrecision, ndcg}`

Capitolo 4

Il programma e l'esecuzione

Il programma vero e proprio, in cui viene descritto l'esperimento e che genera il file `.jar`, è contenuto nella classe `Program` dell'omonimo package. Siccome viene utilizzato sia per il precedente caso analizzato `Relevance List`, sia per la espansione descritta nei capitoli successivi con differenze minime, viene discusso in questo capitolo a parte.

4.1 Il flusso del programma

Il flusso di esecuzione del programma è il seguente:

1. vengono acquisiti i parametri su cui eseguire l'esperimento e controllati per evitare valori non validi
2. vengono generati gli oggetti intermedi necessari derivati dai dati, come ad esempio le distribuzioni di probabilità
3. vengono istanziate le classi necessarie a definire l'algoritmo
4. l'algoritmo viene eseguito e ripetuto per un numero di volte prefissato (o interrotto prematuramente in caso di raggiungimento di una soluzione ottima)
5. la soluzione migliore viene stampata su file

4.2 I parametri

I parametri del programma, che sono i valori che deve ricevere da linea di comando, sono i seguenti:

1. La grandezza della popolazione di NSGA-II : tipo **Integer**
2. Il numero massimo di valutazioni che può svolgere un'esecuzione di NSGA-II : tipo **Integer**
3. La probabilità di crossover : tipo **Double**
4. La probabilità di mutazione : tipo **Double**
5. La lunghezza dei profili di rilevanza da calcolare, ovvero da quanti valori sono composti : tipo **Integer**
6. Il massimo valore che può assumere un elemento singolo del profilo : tipo **Double**
7. Il valore target da raggiungere : tipo **Double**
8. Il numero di documenti rilevanti nel pool : tipo **Integer**
9. Il massimo errore di tolleranza, al di sotto del quale non ha senso considerare differenze tra la valutazione delle soluzioni e il valore target : tipo **Double**
10. Il numero di iterazioni dell'esperimento : tipo **Integer**
11. Il nome della funzione di valutazione da utilizzare : tipo **String**
12. Il nome del file in cui salvare i dati : tipo **String**
13. La frazione di elementi del profilo di rilevanza da inizializzare diversi da zero : tipo **Double**
14. Il tipo di distribuzione da utilizzare per l'inizializzazione delle soluzioni : tipo **String**
15. Il tipo di distribuzione da utilizzare per l'operatore di mutazione : tipo **String**

4.3 L'esecuzione da linea di comando

All'interno della cartella del progetto sono presenti le sotto-cartelle per i sorgenti, i compilati e la cartella **Target** dove sono salvati gli eseguibili e dove viene salvato il file di risultato. Il progetto fa uso di Maven per gestire le dipendenze, dunque lanciando il comando `maven install` dalla cartella principale vengono ricompilati i file .jar in quella locazione.

Per eseguire il programma basta lanciare, dalla cartella principale, il comando

```
java -jar .\target\RelevanceList-1.0-SNAPSHOT-jar-with-dependencies.jar
```

seguito dalla lista degli argomenti nell'ordine indicato sopra. Chiaramente il `.jar` porta il nome del progetto, dunque per l'espansione discussa successivamente **Relevance List Generics**, il comando diventa:

```
java -jar .\target\RelevanceListGenerics-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Un esempio di comando completo può essere:

```
java -jar .\target\RelevanceList-1.0-SNAPSHOT-jar-with-dependencies.jar 100 500000 0.8 0.3 100 1 0.8957 21 0.00005 10 avgPrecision risultati.csv 0.1 geometric geometric
```

4.4 Dettagli implementativi

Di seguito vengono discussi alcuni dettagli implementativi. Il *parsing* dei parametri avviene secondo i tipi precedentemente indicati, provocando un errore nel caso si utilizzino tipi non conformi. Se il numero di parametri inserito non è 15, viene stampato un messaggio che sottolinea la discrepanza.

Per quanto riguarda le funzioni di valutazione e i tipi di distribuzione, le stringhe vengono confrontate con delle enumerazioni in cui sono indicate tutte le possibilità. In caso si utilizzi una stringa non corrispondente a un elemento della enumerazione, viene stampato un messaggio d'errore elencando quali siano le scelte possibili. Se la stringa viene riconosciuta vengono generate le distribuzioni necessarie e l'istanza della sottoclasse di **MetricEvaluator** corrispondente.

Il valore di rilevanza massima di una cella (il punto 6 dell'elenco dei parametri) nel caso binario **Relevance List** non viene effettivamente utilizzato, ma solo controllato visualizzando un messaggio d'errore in caso non sia 1, il che vorrebbe dire che si sta cercando di istanziare un problema non binario.

Capitolo 5

Relevance List Generics: il caso generico

Questo capitolo mostra come è stato espanso il progetto **Relevance List**, pensato per trattare solo valori di rilevanza binari, facendo uso di un *layer* di classi astratte che si colloca tra le implementazioni generiche di jMetal (ad esempio **AbstractSolution**<**S**>) e le implementazioni di tipo definito (ad esempio **RLBinarySolution**). Le classi implementate nel precedente progetto sono state adattate semplicemente alle nuove classi astratte e funzionano quasi interamente nel medesimo modo. Le differenze consistono principalmente nel codice che è stato spostato nella sopra-classe, ma il funzionamento dei metodi e la logica intrinseca sono invariati. Segue una raccolta dei diagrammi di classe e eventuale descrizione.

5.1 **RLAbstractSolution**<**S**, **V**>

La scelta effettuata per le soluzioni è di mantenere le medesime dimensioni del caso binario, dunque la prima ed unica variabile contiene il profilo di rilevanza, un obiettivo per l'errore relativo e un vincolo. Il profilo è codificato in questo caso come un insieme generico di tipo **S**, composto da singoli elementi di tipo **V**. Questo permette di poter definire i metodi astratti che fanno uso di singoli valori, come ad esempio *setSingleValue*. Nella figura 5.1 si possono osservare due implementazioni di **RLAbstractSolution**, una basata su vettori di tipo **BinarySet**, che è il tipo di soluzione definita precedentemente per il progetto **RelevanceList**, e una basata su liste di interi chiamata **RLIntegerSolution**.

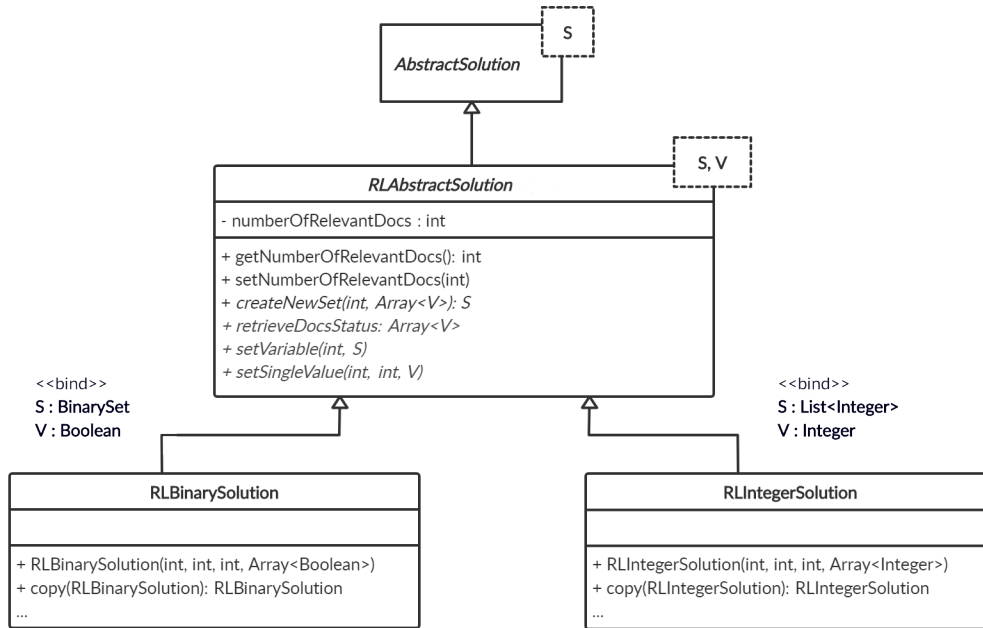


Figura 5.1: Diagramma UML di classe di AbstractSolution

5.2 RLAbstractSolutionFactory<T, V>

La **Factory** astratta, visibile nel diagramma UML in figura 5.2, ha il compito di istanziare soluzioni che sono sottoclassi concrete di **RLAbstractSolution**. A differenza della sua implementazione binaria viene passato come parametro anche *maxValue*, il massimo valore di rilevanza. Il costruttore della soluzione binaria, privo di quel parametro, chiama il costruttore della superclasse passando il valore 1. Siccome **RLAbstractSolution** ha due tipi generici che la caratterizzano, viene utilizzata una *wildcard* per il generico che rappresenta il tipo del vettore, in quanto il secondo generico è sufficiente per la definizione dei metodi. Nel caso si volesse modificare la classe e esplicitare anche il primo generico, esso andrebbe esplicitato tra i generici della classe **Factory**, portandola a tre generici. Per semplificazione è stato preferito utilizzare la *wildcard*.

Le sottoclassi fanno uso del metodo *createDiscreteDistribution* per definire una distribuzione di interi con cui creare il vettore per il profilo di rilevanza. Qualora si volesse implementare una soluzione con vettori di **Double** o tipi più particolari, andrebbe implementato un diverso metodo per il calcolo della distribuzione.

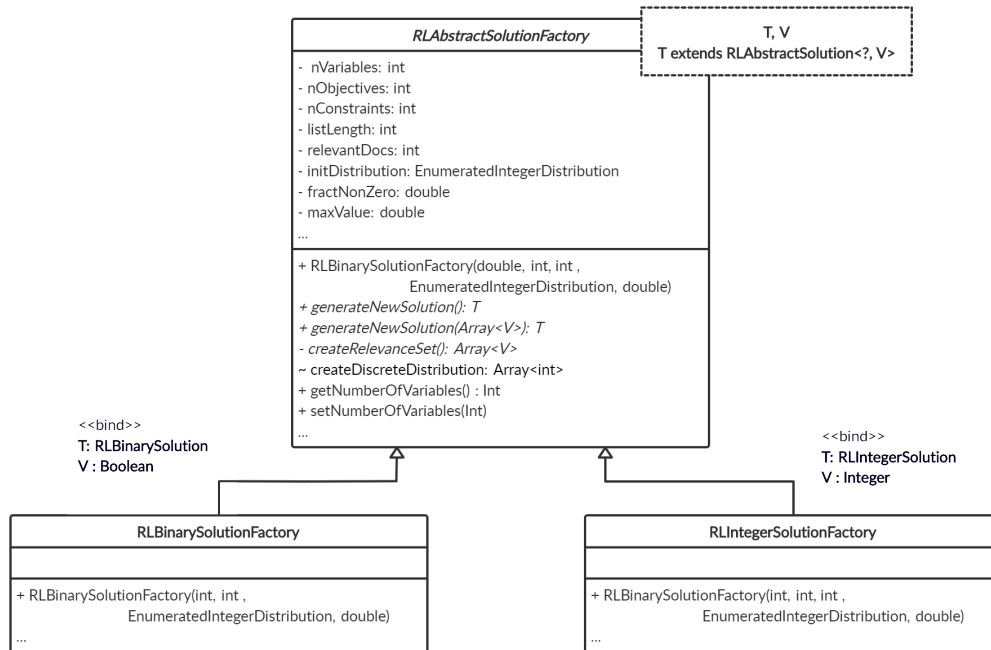


Figura 5.2: Diagramma UML di classe di **RLAbstractSolutionFactory**

5.3 MetricEvaluator

MetricEvaluator (Figura 5.3) era una classe astratta anche nel precedente progetto, tuttavia è interessante notare la relazione tra metriche e tipi di soluzione. In questo caso la classe astratta contiene un metodo *evaluate* per le **RLBinarySolution** e uno per le **RLIntegerSolution**, che devono essere implementati dalle sottoclassi. Questa scelta comporta che l'aggiunta di un nuovo tipo di soluzione abbia un *costo* elevato, in quanto costringe a estendere le classi per inserire il metodo per la nuova soluzione; tuttavia viene estremamente semplificato aggiungere nuove metriche, in quanto si tratta di creare una semplice sottoclasse di **MetricEvaluator** allo stesso livello di quelle già presenti.

5.4 RLAbstractProblem<T, V>

La classe **RLAbstractProblem** (Figura 5.4) non presenta grosse differenze rispetto a quanto mostrato per il caso binario. È presente un utilizzo del pattern *Template Method* per il metodo *evalaute*, la cui implementazione consiste nella chiamata a due metodi astratti *evaluateObjectives* e *evaluateConstraint*.

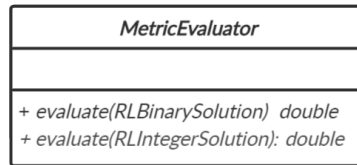


Figura 5.3: Diagramma UML di classe di MetricEvaluator

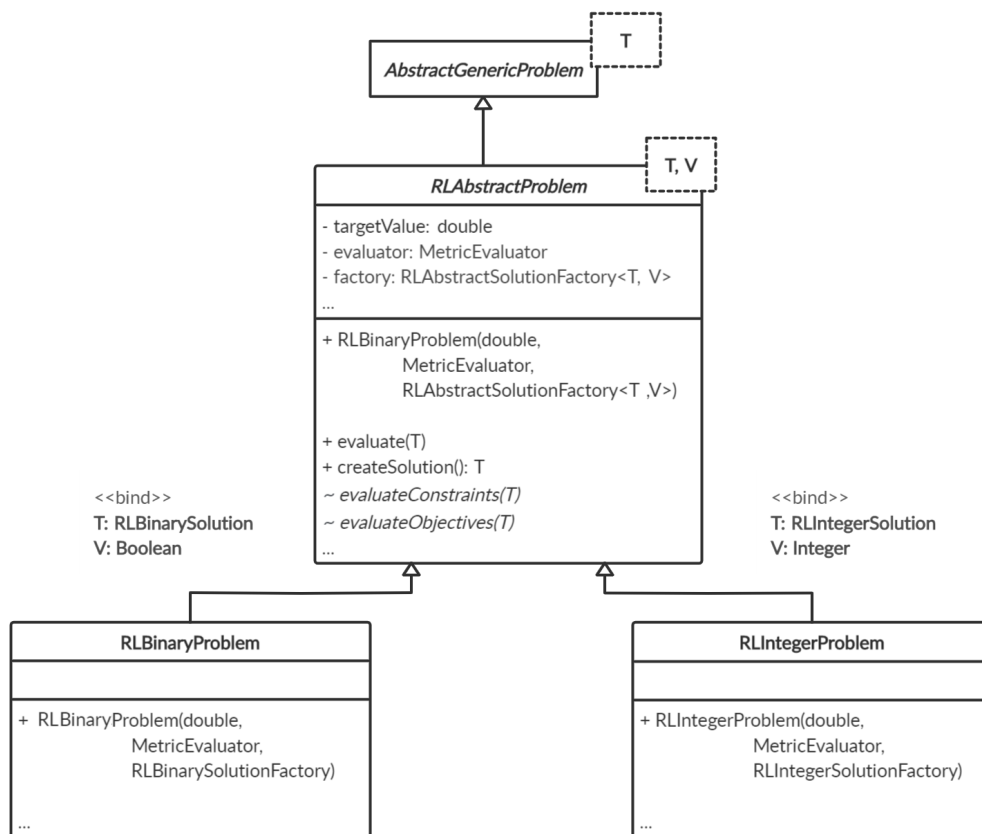


Figura 5.4: Diagramma UML di classe di RLAbstractProblem

5.5 RLNSGAI, Builder e operatori

RLNSGAI e **RLNSGAIBuilder** erano già basate su una soluzione generica nel progetto precedente, dunque vengono riutilizzate senza modifiche necessarie.

Per quanto riguarda gli operatori sono state introdotte due interfacce **RLMutation** e **RLCrossover**, che semplicemente limitano i tipi dei generici utilizzati a sottoclassi di **RLAbstractSolution**. Ciò permette poi di poter eventualmente implementare altri operatori a partire da queste interfacce, anziché da quelle completamente generiche viste in figura 3.6. Siccome si prevede la possibile aggiunta di nuovi operatori, l'operazione di mutazione descritta nel capitolo 3, composta dalla funzioni *swap* e *sum*, è stata rinominata *SumSwapMutation*.

Per ogni operatore di mutazione (o crossover) vengono implementate le nuove interfacce tramite una classe astratta, che implementa già i metodi comuni a tutti i tipi di soluzione. Per il caso di **RLAbstractSumSwapMutation** vengono già implementati i metodi per il numero di genitori, figli, e viene utilizzato il pattern template method per il metodo *execute*. I metodi astratti vengono invece implementati nelle sottoclassi specifiche per il tipo. La gerarchia per gli operatori di crossover funziona nel medesimo modo (Figure 5.5 e 5.6).

5.6 La classe Program e l'esecuzione

La classe **Program** ha il funzionamento descritto nel capitolo 4. La versione di questo progetto effettua un controllo sul valore *maxValue*, tramite cui decidere se istanziare il caso binario o intero. Il builder costruito viene passato a un metodo che genera l'algoritmo ed esegue le valutazioni; la lista di soluzioni generata dagli esperimenti viene passata al metodo di stampa. Viene riportato anche qua un esempio di comando eseguibile da linea di comando:

```
java -jar .\target\RelevanceListGenerics-1.0-SNAPSHOT-jar-with-  
dependencies.jar 100 500000 0.8 0.3 100 1 0.8957 21 0.00005 10  
avgPrecision risultati.csv 0.1 geometric geometric
```

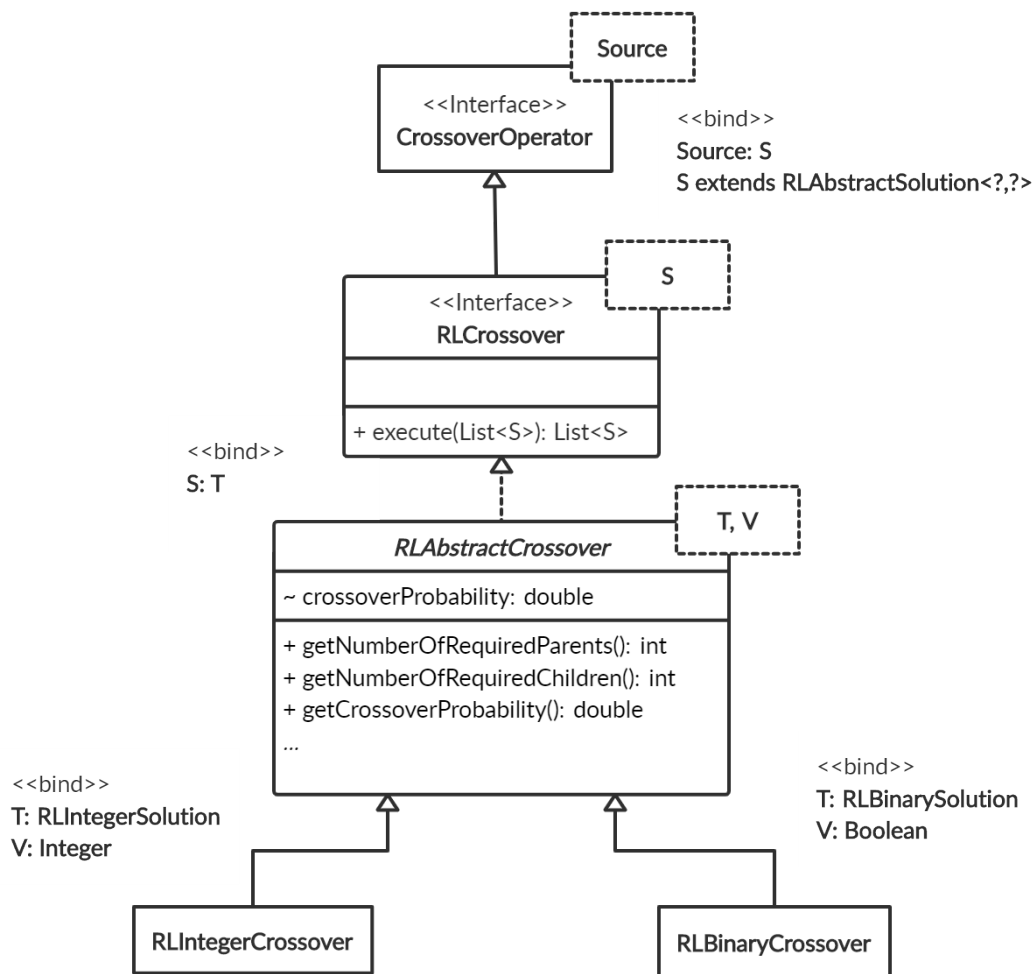



Figura 5.5: Diagramma UML di classe di RLCrossover

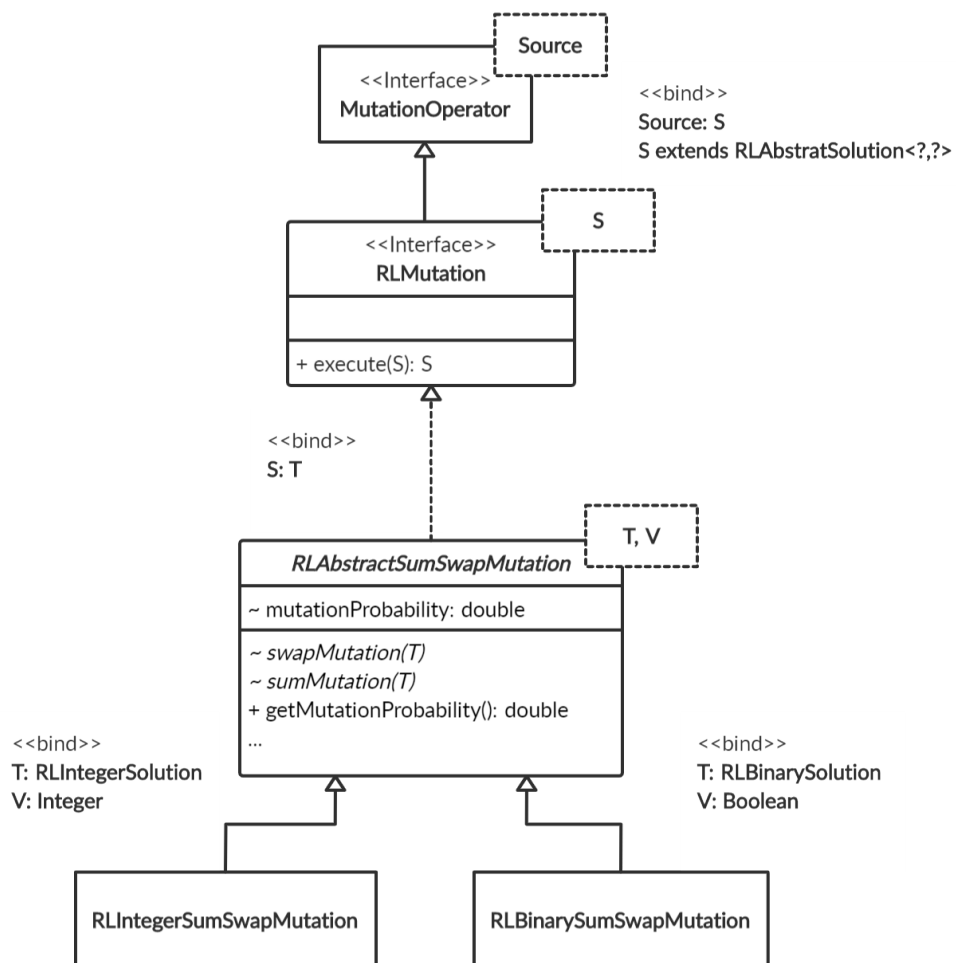


Figura 5.6: Diagramma UML di classe di RLMutation

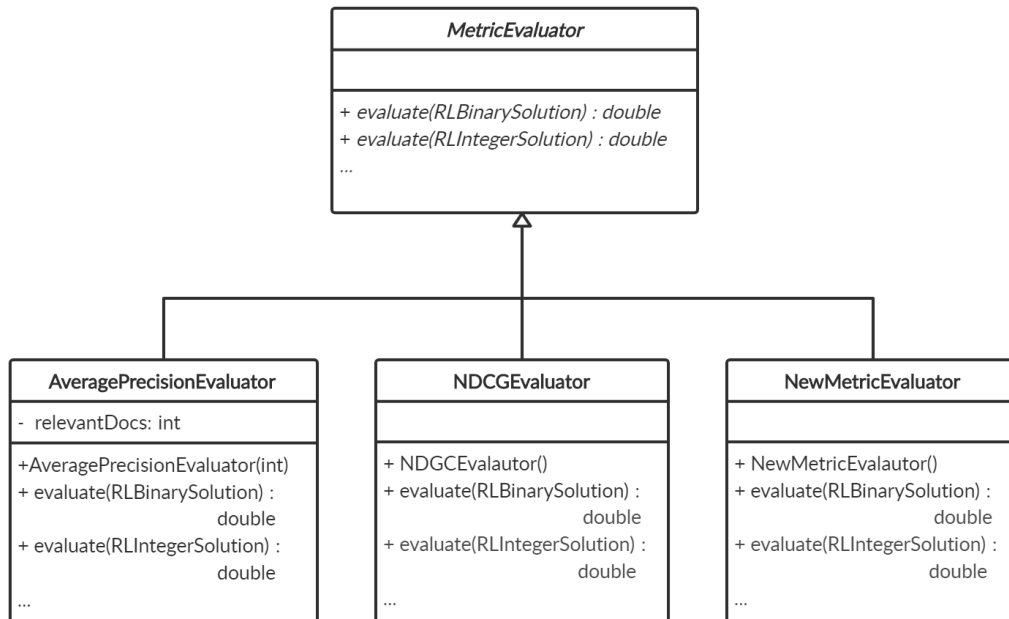


Figura 5.7: Esempio di aggiunta di NewMetricEvaluator alla gerarchia

5.7 Esempi di estensione del framework

In questa sezione verranno mostrati due esempi di estensione del framework: una nuova metrica e un nuovo operatore di mutazione.

5.7.1 Aggiunta di una nuova metrica di valutazione

L'aggiunta di una nuova metrica può essere effettuata estendendo la classe **MetricEvaluator**. La nuova classe deve implementare il metodo *evaluate* per ciascuno dei tipi di soluzione considerati. Nel caso in cui la metrica non sia compatibile con un certo tipo di soluzione, si può mandare un messaggio d'errore nel corpo del metodo. Idealmente un caso del genere andrebbe trattato nel momento in cui si istanzia il framework, riconoscendo che non ha senso passare questo tipo di valutatore al problema che si basa su quel tipo di soluzione. Nella figura 5.7 si può osservare l'aggiunta di una nuova metrica **NewMetricEvaluator**.

Un'aggiunta di operatore di crossover seguirebbe lo stesso procedimento.

5.7.2 Aggiunta di un nuovo operatore di mutazione

Per aggiungere un nuovo operatore di mutazione è sufficiente implementare l'interfaccia **RLMutation** vista nel diagramma UML in figura 5.6. La scelta preferibile è definire una nuova classe astratta che implementi l'interfaccia, in questo esempio chiamata **RLAbstractNewMutation** (Figura 5.8). Dentro questa classe si inserisce il codice comune a tutti i tipi di soluzione per questo tipo di operatore di mutazione. In questo caso è stato aggiunto il metodo per ottenere la probabilità di mutazione e un nuovo ipotetico metodo astratto *newMethod*, che deve essere implementato dalle sottoclassi. Per ogni tipo soluzione va poi creata una sottoclasse concreta che implementi i metodi rimasti astratti.

Un'alternativa a questa soluzione potrebbe essere far sì che **RLIntegerNewMutation** e **RLBinaryNewMutation** implementino direttamente l'interfaccia **RLMutation** senza passare per una classe astratta, tuttavia si ritiene preferibile mantenere la simmetria della gerarchia in quanto il codice risulta meglio organizzato.

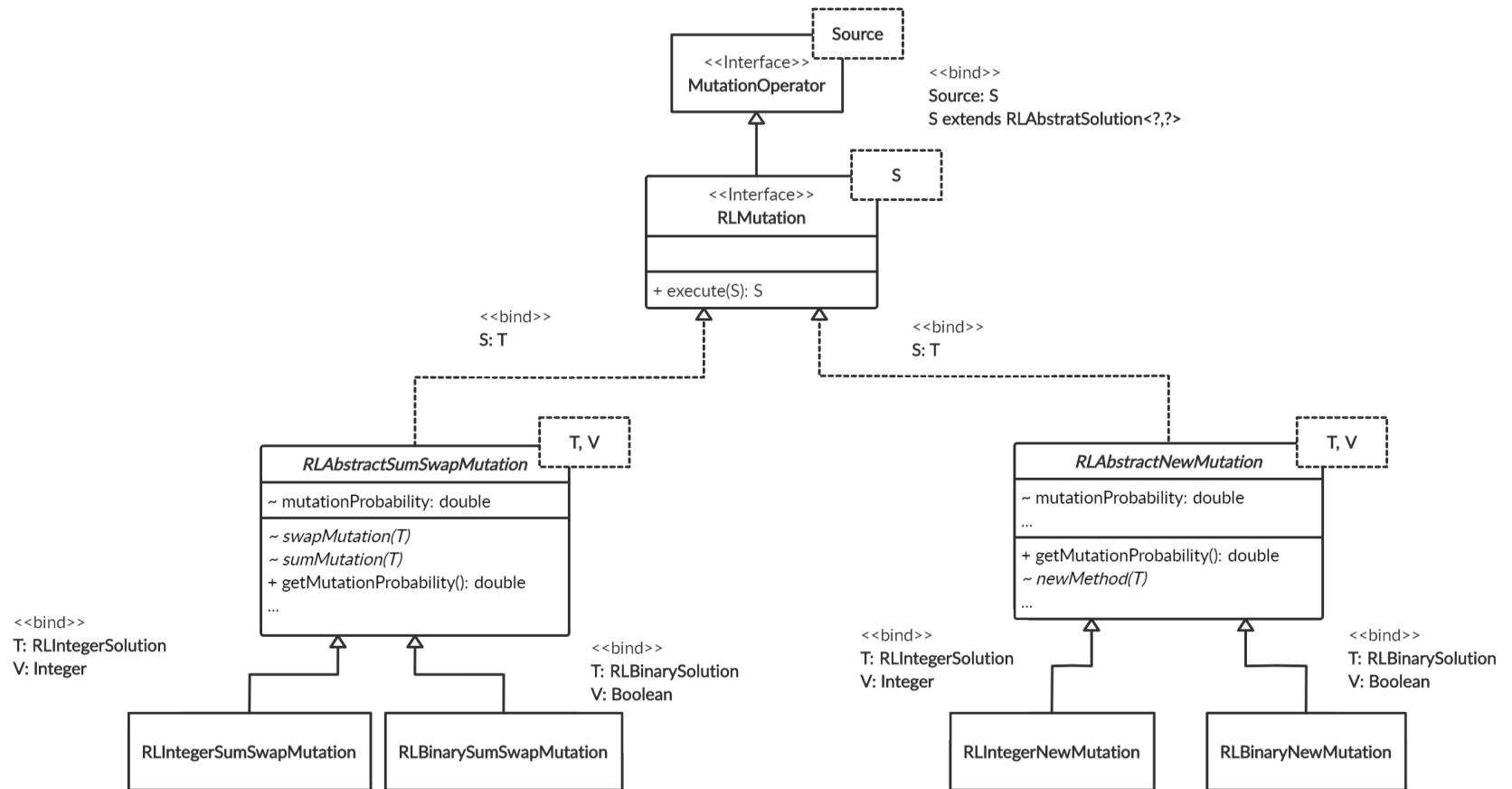


Figura 5.8: Esempio di aggiunta di un nuovo operatore di mutazione alla gerarchia

Capitolo 6

Conclusioni e sviluppi futuri

In questo capitolo viene riassunta l'attività di laboratorio e proposti dei possibili sviluppi futuri per il progetto.

6.1 Risultati ottenuti

Nel corso di questa relazione è stata discussa la realizzazione dei due software *Relevance List* e *Relevance List Generics*.

Il programma *Relevance List* risolve correttamente il problema proposto, ovvero generare in modo stocastico profili di rilevanza a valori binari. È stato implementato rispettando la struttura di base del framework jMetal, senza alterarne la filosofia e le relazioni tra le classi principali. Per rappresentare il problema multi-obiettivo sono state sfruttate le soluzioni proposte da jMetal, senza allontanarsi eccessivamente dalla logica implementata. La struttura base è stata tuttavia arricchita da nuove classi ove ritenuto migliorassero la qualità del progetto in termini di estensibilità e organizzazione. Sono stati rispettati i principi di programmazione OO preposti, le classi sono coese e non sono stati violati i confini di incapsulamento.

Il programma *Relevance List Generics* si è prefisso di estendere il programma funzionante *Relevance List*, in modo da fornire gli strumenti per risolvere il problema per valori di rilevanza non binari. La gerarchia di classi implementata, grazie all'utilizzo dei tipi generici di Java, permette di estendere il software sotto molteplici punti di vista:

- aggiungere nuove tipologie di soluzione;
- aggiungere nuovi operatori;
- aggiungere nuove metriche di valutazione;

- utilizzare un diverso algoritmo genetico.

Tali estensioni sono realizzabili con sforzo minimo, come mostrato in parte nella sezione 5.7, senza modificare il codice esistente (all’infuori chiaramente della classe **Program**) e estendendo con facilità le classi astratte create. Un esempio di nuova tipologia di soluzione è stato implementato basandosi su valori di rilevanza interi.

Entrambi i progetti fanno utilizzo di **Maven** per la gestione delle dipendenze, facilitando il mantenimento del progetto in futuro.

I programmi sono eseguibili tramite linea di comando e stampano i risultati sul file indicato, come descritto nel capitolo 4.

I repository di entrambi i progetti sono visionabili ai seguenti link:

<https://github.com/cussighfilippo/RelevanceList>

<https://github.com/cussighfilippo/RelevanceListGenerics>

6.2 Sviluppi futuri

Il progetto *Relevance List* viene considerato completo, raggiunge gli obiettivi stabiliti ed è servito come base su cui costruire la sua estensione. Siccome il progetto *Relevance List Generics* adempie anche a tutte le funzioni del caso binario, è naturale aprire a possibili sviluppi quest’ultimo:

- effettuare un operazione di refactoring sui nomi di classi e metodi affinché rispettino maggiormente la terminologia del dominio del problema e risultino più coesi tra loro. I nomi adottati risultano in alcuni casi non appropriati e spesso eccessivamente lunghi;
- completare le operazioni di mutazione e crossover per la soluzione basata su liste di interi, lasciate al momento non implementate;
- commentare il codice in modo formale introducendo i contratti dei metodi;
- aggiungere delle operazioni di logging durante l’esecuzione, che al momento non restituisce feedback all’utente all’infuori dei risultati delle iterazioni. Durante un’esecuzione non viene mostrato all’utente lo stato d’avanzamento della singola iterazione dell’esperimento;
- testare il software ripetendo gli esperimenti di Roitero et al. [1];

- effettuare i precedenti esperimenti su nuovi valori di rilevanza non binari.

Bibliografia

- [1] K. Roitero, A. Brunello, J. Urbano, S. Mizzaro. "Towards Stochastic Simulations of Relevance Profiles". University of Udine, Italy, 2019.
- [2] Antonio J. Nebro. jMetal Web Site, <http://jmetal.github.io/jMetal>, ultima visita: 19/09/2020.
- [3] Antonio J. Nebro. jMetal Github repository, <https://github.com/jMetal>, ultima visita: 19/09/2020.
- [4] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.
- [5] M. Soprano. "Riduzione del costo di valutazione dei sistemi di Information Retrieval mediante algoritmi genetici: implementazione ed esperimenti", 2017.