Università degli Studi di Milano – Bicocca

Facoltà di Scienze Matematiche Fisiche Naturali

Corso di Laurea in Informatica

Esecutore simbolico per Java
Integrazione dello strumento

Relatore: Chiar.mo Prof. Mauro.Pezzè

Correlatore: Ing. Giovanni Denaro

Relazione di stage di

Marco Giuseppe Gaboardi

matr. 027025

Anno Accademico 2001/2002

Vorrei ringraziare tutti coloro che mi hanno aiutato e sostenuto nel cammino che mi ha portato fino a questo punto.
Innanzitutto il professor Mauro Pezzè per avermi dato la possibilità di collaborare a questo progetto, per avermi ascoltato e guidato durante il lavoro e la stesura di questa tesi.
Giovanni Denaro che ci ha sempre accompagnato nelle diverse fasi del lavoro guidandoci e aiutandoci nei momenti di bisogno.
Un grosso grazie a Eric che mi ha affiancato nell'intera realizzazione del lavoro, condividendo con me le gioie e i momenti di difficoltà.
Ringrazio Paolo, Rubens e Stefano che da anni mi accompagnano in questa bella avventura e gli amici del Lib e del laboratorio di Testing e Analisi.
Un grandissimo grazie infine ai miei genitori, a mio fratello, a mia sorella e agli amici più cari che mi hanno spronato nel corso degli studi e mi han dato la spinta per arrivare fin qui.
Marco

Sommario

Capitolo 1
Introduzione

Un elemento fondamentale del ciclo di vita del software è la verifica della qualità, che ha come obiettivo la ricerca dei difetti in esso contenuti e la loro eliminazione. La verifica di qualità può essere eseguita tramite test e tecniche di analisi. Pur essendo disponibili numerose tecniche di verifica, nessuna è perfetta né in assoluto né in un contesto relativo [7]. E' quindi necessario utilizzare più tecniche distinte che rappresentino un buon compromesso tra qualità e costi a seconda del contesto in cui ci si trova.

Tra le metodologie utilizzate, le tecniche di analisi formale sono sicuramente molto affidabili in quanto eseguono verifiche di tipo matematico. Spesso però queste tecniche risultano di difficile applicabilità in quanto non adeguate alla flessibilità richiesta dall'implementazione dei sistemi reali. Inoltre l'utilizzo di queste tecniche offre un minor guadagno di efficienza, rispetto all'applicazione di tecniche meno formali. Nei progetti industriali, infatti, la verifica del software avviene solitamente tramite tecniche di testing e di ispezione del codice che ben difficilmente ne verificano anche la correttezza formale.

L'esecuzione simbolica è una tecnica di analisi formale, studiata e approfondita nel corso degli anni. La tecnica si basa sull'idea di simulare l'esecuzione di un programma in base a valori simbolici di ingresso, invece che valori numerici, ottenendo in uscita valori che sono funzione dei simboli di ingresso.

Sono state proposte diverse applicazioni di questa tecnica, dalla generazione di casi di test al model checking, dalla specializzazione alla verifica del software.

Sono stati costruiti diversi sistemi che implementano la tecnica di esecuzione simbolica in diversi contesti.

Girgis ha proposto un sistema per il testing di programmi scritti in FORTRAN77 [10], il sistema usa l'esecuzione simbolica per la generazione di casi di test per tutti i cammini che soddisfano la condizione: "ogni ciclo nel programma è percorso zero , una e due volte". Offutt e Seaman hanno usato l'esecuzione

simbolica come tecnica per riscrivere le variabili interne di un programma in funzione dei valori di ingresso per poi applicare tecniche di testing basate sui vincoli per la generazione di casi di test[11]. Dillon, da sola in [12] e con Kemmerer e Harrison in [13] ha utilizzato l'esecuzione simbolica per verificare formalmente le proprietà di sicurezza di programmi scritti in Ada. Coen-Porisini, Paoli, Ghezzi e Mandrioli in [3] hanno proposto una tecnica che utilizza l'esecuzione simbolica per la specializzazione di un generico componente software scritto in Ada. Coen-Porisini, Denaro, Ghezzi e Pezzè in [2] hanno proposto un sistema per la verifica di sistemi critici tramite esecuzione simbolica, utilizzata su programmi scritti in un sottolinguaggio del C.

Di recente l'attenzione dell'industria e dell'ingegneria del software è stata attirata da diverse innovazioni che ne hanno condizionato le scelte strategiche e, di riflesso, gli interessi.
La sempre crescente diffusione del paradigma orientato agli oggetti ha accresciuto l'interesse dell'ingegneria del software e dell'industria verso lo studio di nuove tecniche di sviluppo e di analisi specializzate che rendano maggiormente efficienti o che sostituiscano le tecniche abituali.
Le tecniche classiche per il controllo di qualità risultano spesso insufficienti nel campo del paradigma orientato agli oggetti, infatti i nuovi principi introdotti da questo paradigma possono portare alla creazione di nuove condizioni di errori finora inesplorate [4].
Nella realtà attuale troviamo che le tecniche per l'analisi e il controllo di qualità per software orientato agli oggetti siano ancora in numero insufficiente; l'ingegneria del software, pur movendosi in questa direzione, non è ancora in grado di fornire strumenti sufficienti per sopperire ai bisogni. Diverse tecniche sono state studiate, nel corso degli anni, per il controllo di qualità del software orientato agli oggetti [7,8,9]; queste tecniche non riescono però a coprire tutti gli aspetti che questo paradigma supporta.
Buy, Orso e Pezzè hanno proposto una tecnica per la generazione di casi di test per il testing automatico di classi [1]. Questa tecnica prevede l'utilizzo di tre tecniche, analisi del flusso di dati, esecuzione simbolica, deduzioni automatiche, per la generazione di sequenze di invocazioni di metodi, che esercitino la classe sotto test.
Lo strumento realizzato nell'ambito del progetto TWO [1], prevede nella fase di esecuzione simbolica, l'utilizzo di uno strumento già esistente, appositamente integrato. L'esecutore simbolico integrato è stato sviluppato nell'ambito del progetto SAVE [2] e prevede l'utilizzo per software non orientato agli oggetti, infatti esegue simbolicamente un sottolinguaggio del C denominato *Safer-C*.

La tecnica di esecuzione simbolica, oltre ad essere una tecnica formale molto potente, risulta adatta ad essere una tecnica di modellazione e di analisi anche per software orientato agli oggetti. Nessuno degli strumenti realizzati [2,3,10] prevede però l'esecuzione simbolica di software orientato agli oggetti; in questo contesto nasce l'idea-guida di questa tesi: la realizzazione dello strumento esecutore simbolico per software orientato agli oggetti.

Il successo del paradigma orientato agli oggetti ha alimentato anche nell'ambito dei linguaggi di programmazione, la nascita e la diffusione di linguaggi basati sul paradigma e in grado di sfruttarne pienamente le potenzialità.
Tra i linguaggi che sono basati sul paradigma di programmazione ad oggetti, Java risulta essere il linguaggio che, negli ultimi anni, si è maggiormente affermato.
Il successo di Java è da attribuire, tra le varie motivazioni, anche alla nascita di internet e alla sua folgorante diffusione. Questo fenomeno ha incrementato l'interesse e l'attenzione verso quelle caratteristiche del software che ne agevolano la condivisibilità e la portabilità, favorendo così quei prodotti che posseggono queste caratteristiche, tra essi Java, grazie alla portabilità del linguaggio intermedio interpretabile, prodotto dalla sua compilazione.
La tesi si focalizza sul problema di realizzare un esecutore simbolico per il linguaggio Java, in particolare però l'esecuzione simbolica viene effettuata sul linguaggio intermedio *bytecode*.
La scelta di eseguire simbolicamente il bytecode porta diversi vantaggi.
Il bytecode è un linguaggio intermedio interpretabile prodotto dalla compilazione di sorgenti Java, per questo motivo eseguire simbolicamente il bytecode equivale ad eseguire simbolicamente il linguaggio Java; l'esecuzione simbolica di bytecode non necessita del codice sorgente, nella pratica non sempre recuperabile. Il bytecode essendo un linguaggio intermedio prevede un architettura sintattica molto più semplice rispetto ad un linguaggio di più alto livello, ciò comporta degli algoritmi di esecuzione simbolica semplificati. Infine qualsiasi linguaggio di alto livello, in linea teorica può essere tradotto in bytecode senza perdita di efficacia, vantaggio che rende lo strumento da realizzare meno limitato nel contesto.
Nell'esecuzione tradizionale, il bytecode viene solitamente interpretato e eseguito da una macchina computazionale definita Java Virtual Machine, l'obiettivo è quindi di realizzare una nuova Java Virtual Machine che esegua simbolicamente, e non più numericamente, il bytecode.
Per ottenere il risultato prefisso vengono analizzate le strutture dati necessarie e la modalità di esecuzione classica della Java Virtual Machine, prevedendo l'estensione o la sostituzione delle componenti non adeguate.

La tesi è organizzata come segue. Il capitolo 2 descrive approfonditamente la tecnica di esecuzione simbolica e il suo utilizzo. Il capitolo 3 analizza le motivazioni che ci hanno spinto a realizzare un esecutore simbolico per Java, presenta il linguaggio intermedio bytecode, la Java Virtual Machine e il suo metodo di esecuzione tradizionale. Il capitolo 4 analizza approfonditamente la tecnica di esecuzione simbolica applicata al linguaggio bytecode, presenta la progettazione degli algoritmi di controllo tramite la progettazione dei componenti dello strumento e la presentazione dei componenti esterni utilizzati. Il capitolo 5 infine presenta alcune considerazioni finali sul lavoro svolto e i possibili sviluppi futuri.

Capitolo 2
Esecuzione Simbolica

Questo capitolo approfondisce la tecnica che trova applicazione nel lavoro proposto in questa tesi. La tecnica di esecuzione simbolica è una tecnica formale di analisi del software che trova diverse applicazioni nel contesto della verifica del software.
Si introdurranno i concetti che stanno alla base della tecnica e le applicazioni in cui può essere impiegata.

2.1 Concetti generali

Un programma può essere visto come una funzione che dati valori di ingresso, produce valori di uscita. Quando un programma viene eseguito, tipicamente i suoi valori di ingresso sono variabili non locali, valori passati come parametri, oppure valori forniti dall'utente. Dati valori di ingresso l'esecuzione del programma produce valori di uscita che sono così loro funzione.
L'idea che sta alla base dell'esecuzione simbolica è ottenere la funzione calcolata dall'esecuzione del programma in esame. Fornendo ad un programma come valori di ingresso dei simboli e non dei valori numerici, e simulandone l'esecuzione, i valori di uscita, che vengono calcolati come funzione dei simboli di ingresso, rappresentano esplicitamente la funzione calcolata dall'esecuzione di un singolo cammino. Questo vuol dire che durante l'esecuzione simbolica di un programma, i valori di ingresso sono costituiti da simboli rappresentanti ognuno l'insieme dei valori che la variabile associata potrebbe assumere all'inizio di una normale esecuzione numerica. In ogni istante dell'esecuzione, le variabili sono legate ad espressioni simboliche derivanti dalla combinazione dei simboli di ingresso secondo le modalità stabilite dal codice eseguito fino a quel momento. Le variabili di uscita sono le funzioni dei simboli di ingresso date dal completamento dell'esecuzione del codice.

Ogni esecuzione simbolica di un programma corrisponde all'esecuzione del programma stesso su un sotto insieme del proprio dominio di ingresso; nel caso particolare che tutte le espressioni simboliche corrispondano ad un unico valore numerico, l'esecuzione simbolica corrisponde all'esecuzione numerica tradizionale

2.2 Lo stato dell'esecuzione simbolica

L'esecuzione simbolica differisce chiaramente dall'esecuzione tradizionale.
Nell'esecuzione tradizionale in un determinato istante, lo *stato* dell'esecuzione è completamente identificato dalla coppia <IP,V>, dove *IP* è il puntatore all'istruzione successiva e *V* è l'insieme delle variabili con il rispettivo valore numerico, cioè l'insieme delle coppie <variabile,valore>. Queste informazioni permettono l'esecuzione di un programma da uno stato iniziale a uno stato finale univocamente determinato. Durante l'esecuzione vengono raggiunti dei punti di decisione, grazie allo stato della memoria in quell'istante è sempre possibile determinare univocamente il percorso da seguire. Più approfonditamente, ogni punto di decisone ha associata una condizione, sui valori delle variabili; la condizione può essere valutata vera o falsa grazie al fatto che le variabili della condizione assumono valori numerici ben precisi, cioè grazie allo stato di esecuzione raggiunto nel determinato istante. In questo modo il percorso da eseguire viene identificato in base alle valutazioni delle condizioni dei vari punti di decisione incontrati.
Nell'esecuzione simbolica, lo stato dell'esecuzione è identificato dall'insieme delle variabili e dal loro valore simbolico, cioè dall'insieme delle coppie <variabile,valore simbolico>, questo però non è sufficiente per determinare il percorso da seguire, infatti la valutazione delle condizioni di esecuzione non sempre permette di scegliere univocamente un percorso da seguire.
Quando si incontra un punto di decisione, i valori possono essere divisi in due insiemi: l'insieme dei valori che rende la condizione di esecuzione vera e l'insieme dei valori che rende la condizione di esecuzione falsa. Solo in casi particolari abbiamo una decisione univoca, cioè quando uno dei due insiemi è vuoto.
In generale, nell'esecuzione simbolica, quando si arriva a un punto di decisione, per proseguire si devono fare delle ipotesi sui valori che le variabili potranno assumere, cioè si deve valutare se i valori che le variabili potranno assumere appartengono all'insieme dei valori che rendono vera la condizione di esecuzione oppure all'insieme dei valori che la rendono falsa.

Le informazioni sullo stato, sufficienti per l'esecuzione numerica, non sono più sufficienti per quanto concerne l'esecuzione simbolica.
Durante l'esecuzione simbolica di un cammino è essenziale avere anche le informazioni riguardanti le ipotesi fatte sulle variabili. In particolare, le espressioni finali dei valori di uscita, calcolati lungo un determinato cammino, sono significative solo per i valori dei simboli che rendono possibile l'esecuzione del percorso stesso. Quando viene raggiunto un nuovo punto di decisione, è possibile o meno, che le variabili su cui sono state assunte le ipotesi diano la possibilità di valutare la condizione associata.
Le ipotesi sui simboli di ingresso in relazione ai percorsi di esecuzione, nell'esecuzione simbolica, vengono comprese in un predicato del primo ordine che chiamiamo *path condition*, che viene inserito tra le informazioni importanti dello stato dell'esecuzione. Lo stato dell'esecuzione, nel caso dell'esecuzione simbolica, ad un determinato passo si può riassumere tramite una tripla, e non più una coppia come nel caso dell'esecuzione numerica, <IP,PC,V> dove: *IP* è il puntatore alla prossima istruzione, *V* è l'insieme delle variabili con il rispettivo valore simbolico, cioè l'insieme delle coppie <variabile,valore simbolico>, *PC* è la path condition del cammino eseguito.

2.3 Inizializzazione dello stato simbolico

Quando inizia l'esecuzione simbolica di un modulo, IP punta alla prima istruzione da eseguire. A ciascuna variabile di ingresso viene assegnato un nuovo simbolo, alle altre variabili viene assegnato lo speciale valore "*Undef*". PC ha tipicamente il valore *vero*, se inizialmente non vi sono ipotesi sulle variabili, se al contrario è definita una precondizione che implica delle ipotesi sulle variabili, la path condition può essere opportunamente inizializzata.
Ogni volta che l'esecuzione di un'operazione d'ingresso coinvolge una variabile, alla variabile coinvolta deve essere assegnato un nuovo valore, il nuovo valore è un espressione simbolica che consiste di un simbolo, un valore numerico oppure di un espressione che a sua volta è composta da valori simbolici o meno.

Vediamo un esempio di esecuzione simbolica su un frammento di codice Java.

```
1int example(int in)
2 {
3       int out;
4       if (in>1)
5               if(in>0 and in<2)
6                       out=in;
7                else
8                       out=-in;
9                return(out);
10 }
```

dopo l'esecuzione della riga 3 siamo nella situazione:

V: (in,a), (out, Undef)
PC: True
IP: 4

A questo punto però vediamo che devono essere fatte delle ipotesi sul valore della variabile per poter superare il punto di decisione, ad esempio se vogliamo eseguire il primo ramo dell'if, eseguita l'istruzione alla riga 5 ci troveremo nella situazione:

V: (in,a), (out,a)
PC: a>1
IP:9

A questo punto è necessario valutare il predicato "in>0 and in<2" fornito dalla seconda istruzione condizionale. L'esecuzione di questo cammino è condizionata dalle ipotesi fatte sulle variabili; valendo la path condition "a>1", la valutazione del predicato, della seconda istruzione condizionale e della path condition, si riduce alla valutazione del predicato "a>1 and a<2".

Se avessimo invece voluto seguire il secondo ramo dell'if ci saremmo trovati alla riga 8 nella situazione:

V: (in,a), (out,-a)
PC: a<0
IP:9

La path condition viene costruita applicando l'operazione di and tra il predicato che costituisce la path condition precedente e il predicato dell'istruzione condizionale eseguita. Ad esempio, nel caso del primo ramo dell'if il predicato diventa "true and a>0" riducibile a "a>0".
Ogni volta che si incontra un punto di decisione, la valutazione della veridicità della condizione è determinata dalla path condition: se PC implica la condizione, allora è vera, se implica la negazione della condizione, allora è falsa, in tutti gli altri non è determinata univocamente.
Finora abbiamo parlato di esecuzione simbolica di una singola istruzione, ma come nei casi di esecuzione tradizionale, ha senso parlare sia di esecuzione simbolica di un intero modulo di un programma, sia di esecuzione simbolica eseguita passo passo con interazione da parte dell'utente, la differenza è solo nell'utilizzo diverso dei due modi di interazione.

2.4 Utilizzo della tecnica di esecuzione simbolica.

L'esecuzione simbolica è una tecnica formale di analisi potente che ha diverse possibilità di applicazione.
Dall'analisi degli stati simbolici in vari istanti dell'esecuzione simbolica di un programma si possono ricavare importanti informazioni riguardanti il significato del codice, utili per operazioni di verifica di correttezza o più in generale di analisi del codice.
L'analisi delle espressioni simboliche associate alle variabili in determinati stati di esecuzione, permette ad esempio la verifica di conformità del codice con quanto richiesto dalle specifiche, l'appartenenza delle variabili a determinati domini, il controllo sulla consistenza rispetto a determinate condizioni di ingresso.
Tramite l'esecuzione simbolica è possibile ottenere dati di test riguardo alla copertura di determinati cammini all'interno del codice, questo grazie al fatto che la *path condition*, in qualsiasi punto dell'esecuzione, esprime i vincoli sui valori che devono essere rispettati per eseguire il determinato cammino.

Capitolo 3
Esecuzione simbolica di programmi Java

In questo capitolo introdurremo le motivazioni che ci hanno portato a progettare un esecutore simbolico per linguaggi orientati agli oggetti e spiegheremo il motivo per cui abbiamo scelto Java come linguaggio a cui applicare la tecnica..
Approfondiremo la tecnica di esecuzione simbolica in relazione allo specifico linguaggio, vedremo i principi di funzionamento di Java e analizzeremo le caratteristiche del linguaggio che possono influenzare la tecnica di esecuzione simbolica, infine prenderemo in considerazione il linguaggio intermedio bytecode e le strutture che supportano la sua esecuzione.

3.1 Esecuzione simbolica e Java

La tecnica formale di esecuzione simbolica trova applicazioni nella verifica di qualità del software. La tecnica in particolare, nel corso degli anni, ha trovato alcune applicazioni nella verifica di software tramite analisi di software tradizionale.
Finora la tecnica è stata studiata solo per essere applicata a linguaggi tradizionali e non a linguaggi orientati agli oggetti. Le motivazioni di questo fatto non sono però da attribuire ai principi del paradigma, infatti anche i linguaggi orientati agli oggetti risultano adatti all'applicazione della tecnica di esecuzione simbolica.
Molti linguaggi sono nati per supportare il paradigma di programmazione orientata agli oggetti da quando ha fatto la sua prima comparsa. Tra questi Java è il linguaggio che negli ultimi anni ha maggiormente incrementato il proprio grado di diffusione, arrivando a conquistare una grossa fetta del mercato del software di nuova produzione.
Il successo di Java però non è da ricercarsi esclusivamente nella diffusione del paradigma ad oggetti, ma anche, oltre alle leggi che regolano il mercato

dell'industria, nella diffusione sempre crescente di internet. Java grazie al proprio modo di operare, supporta la caratteristica indispensabile per la diffusione tramite internet: la portabilità.
Grazie alla propria diffusione Java risulta essere un buon candidato come obiettivo di strumenti per la verifica del software. In quest'ottica l'esecutore simbolico è stato ideato per l'esecuzione di software Java.

3.2 Concetti generali di Java

Il linguaggio Java nasce da una versione del linguaggio Oak, ideato da James Gosling, disegnato per applicazioni nei sistemi dedicati all'elettronica di consumo. Come specificato in [14] "Java è un linguaggio di programmazione concorrente, orientato agli oggetti, basato sulle classi, utile per tutti gli usi, disegnato per essere il più possibile indipendente dall'architettura. Dà la possibilità ai programmatori di scrivere un programma una sola volta e di eseguirlo ovunque su Internet".
Java è simile a linguaggi di programmazione come C e C++ rispetto ai quali prevede alcune nuove caratteristiche e ne esclude altre. È un linguaggio di alto livello studiato per essere un linguaggio applicativo e non di ricerca.
L'esecuzione di un programma Java prevede due fasi ben distinte:

- la prima fase consiste nella compilazione del codice scritto in Java, la compilazione produce un codice intermedio, immagazzinato in un file binario strutturato (chiamato file "*.class"*).
- la seconda fase consiste nell'interpretazione del programma, viene tradotto ed eseguito il codice contenuto nei file ".class" tramite un componente chiamato Java Virtual Machine

Proprio la divisione in due fasi ha contribuito a determinare il successo di Java, infatti un programma scritto in Java può essere facilmente trasportato, senza perdita rilevante di informazioni per l'esecuzione, nella sua forma binaria di file ".class", per poi essere interpretato e quindi eseguito su diverse piattaforme.

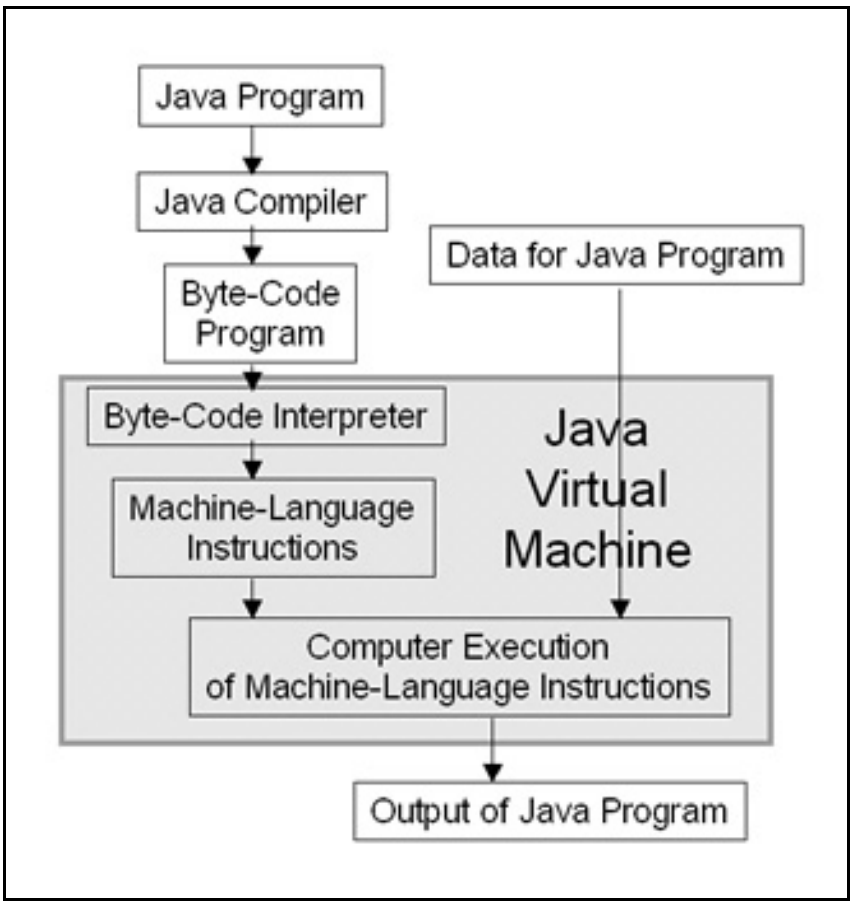


Figura 1: Modalità di esecuzione di un programma Java

La Java Virtual Machine è il componente della tecnologia, responsabile dell'indipendenza dall'hardware e dal sistema operativo. La Java Virtual Machine è in realtà una macchina computazionale astratta, di cui si possono avere diverse implementazioni in base alla piattaforma sulla quale devono essere eseguiti i programmi.
La Java Virtual Machine ha il compito di eseguire i file ".class". I file ".class" seguono uno standard, chiamato "class file format" che prevede che essi contengano il linguaggio che deve essere interpretato dalla Java Virtual Machine e le informazioni aggiuntive importanti per l'esecuzione del programma.

3.3 Esecutore simbolico per Java

La tecnica di esecuzione simbolica, diversamente dalla tecnica di esecuzione tradizionale, prevede che il codice di un programma venga eseguito utilizzando valori simbolici.
Al fine di applicare la tecnica di esecuzione simbolica si deve creare una macchina astratta computazionale a stack, di tipo SIMPLESEM[16], che sia in grado di lavorare con valori simbolici e che sia in grado di eseguire del codice sorgente. La macchina astratta deve quindi essere in grado di manipolare algebricamente i valori simbolici.
La manipolazione dei valori, nell'esecuzione tradizionale, avviene solitamente tramite l'esecuzione delle istruzioni del linguaggio. Ogni singola istruzione è in realtà un insieme di passi che descrivono come manipolare lo stato dell'esecuzione. Allo stesso modo, l'esecutore simbolico deve prevedere un insieme di passi corrispondenti al modo di elaborare lo stato simbolico in base alle istruzioni del linguaggio che si vuole eseguire, gli insiemi di questi passi costituiscono gli algoritmi di esecuzione simbolica
Un esecutore simbolico non è altro che un programma che implementi gli algoritmi di esecuzione simbolica e le strutture dati necessarie alla memorizzazione dello stato simbolico. Nel nostro caso quindi l'esecutore simbolico sarà un programma che implementerà gli algoritmi di esecuzione simbolica rispetto a istruzioni Java, e tutte le strutture dati necessarie a immagazzinare lo stato simbolico di programmi Java.

3.3.1 Esecuzione simbolica su bytecode

L'esecutore simbolico è una macchina computazionale, per questo motivo, il codice che il componente dovrà eseguire è bene che sia di semplice comprensione ma che allo stesso tempo sia il più possibile vicino al linguaggio direttamente eseguibile da un calcolatore, così da eliminare tutte le specializzazioni che sono fornite da un linguaggio di alto livello. Per questo motivo non è conveniente lavorare direttamente con codice Java, ma è preferibile lavorare su codice scritto in un linguaggio intermedio.
Il bytecode come visto è il linguaggio intermedio prodotto dalla compilazione di programmi Java, viene naturale pensare quindi di applicare la tecnica di esecuzione simbolica al bytecode.
Nella progettazione di un componente che implementi la tecnica di esecuzione simbolica, il fatto di applicare la tecnica al linguaggio intermedio bytecode può portare diversi vantaggi:

- Il bytecode viene prodotto dalla fase di compilazione di programmi scritti in Java, tramite il compilatore "javac", in file binari strutturati ".class" senza perdita di informazioni rilevanti all'esecuzione simbolica.
- Il bytecode è un linguaggio intermedio contenente tutte le informazioni utili all'esecuzione, quindi non è strettamente necessario avere il file sorgente Java.
- Un primo parsing del codice risulta già effettuato, il programma in bytecode risulta così essere sintatticamente corretto secondo la sintassi Java.
- Il bytecode può in linea teorica essere ottenuto anche dalla compilazione, tramite strumenti appositamente realizzati, di linguaggi di programmazione diversi da Java.
- Il bytecode, essendo un linguaggio di basso livello, prevede una struttura notevolmente semplificata dei costrutti propri dei linguaggi di alto livello, per questo motivo gli algoritmi di esecuzione simbolica risultano di più semplice implementazione.

Il bytecode è immagazzinato nei file binari strutturati di tipo ".class", tipicamente questi file vengono interpretati dalla Java Virtual Machine. Allo stesso modo per eseguire simbolicamente il bytecode deve essere previsto uno strumento che svolga lo stesso lavoro della Java Virtual Machine, utilizzando valori simbolici anziché valori numerici.

3.3.2 Java Virtual Machine e esecuzione simbolica.

La Java Virtual Machine è la macchina astratta che ha il compito di interpretare tutti i programmi contenuti in una forma strutturata denominata "class file format", in realtà i file ".class" sono solo un particolare caso di struttura di immagazzinamento.

Abbiamo già visto come sia possibile e conveniente eseguire simbolicamente il bytecode; per fare questo deve essere prevista una macchina astratta che sia in grado di eseguire simbolicamente, e quindi in maniera simile ma non uguale alla Java Virtual Machine, il codice.

L'esecutore simbolico, analogamente alla Java Virtual Machine, è una macchina computazionale astratta, che consente di eseguire il bytecode. Il bytecode prevede ricorsione, per questo motivo la macchina astratta dovrà prevedere di immagazzinare i valori delle variabili, siano essi simbolici o numerici, e tutte le informazioni rilevanti ai fini dell'esecuzione della funzione stessa in record di attivazione. Durante l'esecuzione i record di attivazione delle funzioni attive in un particolare istante sono organizzati in una struttura a pila.

Come tutte le macchine computazionali, anche l'esecutore simbolico dovrà prevedere un insieme di istruzioni, che definiscono il linguaggio accettato. Nel nostro caso le istruzioni saranno tutte le istruzioni previste dal bytecode.
Al pari della JVM l'esecutore simbolico dovrà poter agire su tipi di dati differenti, in particolare però dovrà prevedere anche l'utilizzo di tipi di dati utili alla manipolazione di valori simbolici.
Lo stato dell'esecuzione nel caso simbolico è composto non solo dai valori simbolici e numerici, ma anche dalle path conditions; per questo motivo l'esecutore simbolico non solo dovrà prevedere tutte le strutture di memoria classiche delle macchine computazionali, ma dovrà anche contenere alcune strutture appropriate, al fine di contenere valori simbolici e le ipotesi che compongono le path condition.

Capitolo 4
Progettazione dell'esecutore simbolico per Java

Questo capitolo presenta la fase di progettazione degli algoritmi di controllo. Nella prima parte verrà effettuata un analisi approfondita del codice per il quale viene richiesta l'esecuzione simbolica e saranno presi in considerazione i requisiti dello strumento, nella seconda parte sarà presentata l'architettura pensata per l'esecutore simbolico e conseguentemente l'architettura del prototipo, infine verranno presentati e analizzati i componenti effettivamente utilizzati.

4.1 Il codice sorgente

L'esecutore simbolico deve essere in grado di eseguire programmi scritti in Java. Nell'ambito del progetto TWO[1], Buy, Orso e Pezzè hanno realizzato uno strumento per il testing automatico di software orientato agli oggetti. La tecnica, implementata nello strumento, prevede una fase di esecuzione simbolica. L'esecutore simbolico dello strumento, è utile all'applicazione della tecnica a programmi scritti in C, sfrutta un front-end per produrre dal sorgente originale un codice di livello intermedio. Allo stesso modo abbiamo pensato di utilizzare un front-end che partendo da un file Java, producesse un linguaggio intermedio simile a quello analizzato dallo strumento già esistente; al fine di non dover progettare nuovamente tutto lo strumento, ma solo adattare e modificare i componenti già esistenti. Per questo motivo è stato analizzato un front-end prodotto dall'Edison Group[17], analogo a quello utilizzato nel progetto TWO. Questo strumento, prendendo in ingresso un programma scritto in Java, produce codice equivalente in un formato intemedio chiamato "Java Intermediate Language". Il linguaggio prodotto non è risultato simile al linguaggio eseguito dall'esecutore simbolico utilizzato all'interno del progetto TWO, così abbiamo dovuto abbandonare l'idea di estendere semplicemente questo componente.

I programmi Java attraverso la compilazione, quindi analizzati sintatticamente, sono tradotti in un codice intermedio di basso livello (*bytecode)*, e memorizzati in file di tipo ".class". Per eseguire un programma Java è necessario far interpretare ad una macchina computazionale che viene definita Java Virtual Machine [15] i file di tipo ".class".
Dati questi fattori e dopo una approfondita analisi [Appendice A] abbiamo deciso di adottare il bytecode come codice su cui applicare la tecnica di esecuzione simbolica.

4.1.1 Il byte code

La fase di compilazione di un programma Java tramite il compilatore "javac" produce file binari strutturati secondo la struttura definita dal "class file format". I file ottenuti, contengono tutte le informazioni utili per l'esecuzione del programma. Per semplicità da ora in poi supporremo che un singolo file ".class" così prodotto, contenga una sola classe.
I file ".class" che contengono il byte code sono suddivisi in tre parti principali:

- Constant pool table
- Fields info
- Methods info

Constant pool table. Contiene i nomi delle classi utilizzate, i riferimenti ai file che contengono il codice delle classi esterne utilizzate e tutte le informazioni utili riguardanti i metodi e gli attributi a cui viene fatto riferimento all'interno del codice.

Fields info. Contiene tutte le informazioni degli attributi della classe. Per gli attributi, le informazioni utili all'esecuzione, sono solamente marcatore di visibilità, nome e tipo.

Methods info. Contiene tutte le informazioni riguardanti tutti i metodi della classe. Ogni metodo, incluso ogni costruttore o distruttore, definito all'interno del codice sorgente si trova in questa sezione. Per ogni metodo sono memorizzati il modificatore di visibilità, il nome e il tipo.I metodi all'interno della sezione Methods info vengono differenziati grazie al tipo di parametri passati e al tipo del valore di ritorno, in questo modo viene anche risolto l'overloading dei metodi. Il tipo di un metodo, descrive sotto forma di stringa, il tipo di valore ritornato dal metodo e il tipo delle variabili passate per parametro. All'interno di questa sezione troviamo anche il codice di basso livello tradotto in fase di compilazione, le

informazioni riguardanti le variabili locali e le informazioni riguardanti la quantità di memoria statica necessaria per l'esecuzione del codice. Infine a seconda di come viene eseguita la compilazione, in questa sezione, possiamo trovare delle informazioni aggiuntive non indispensabili all'esecuzione dei programmi, ma utili per operazioni di controllo e di analisi del codice.

Quando si vuole eseguire il codice in modalità di analisi è spesso utile avere la possibilità di potersi confrontare con il codice sorgente. Questa possibilità viene offerta tipicamente dal compilatore "javac" tramite l'opzione "-g", che permette di aggiungere delle informazioni che collegano il sorgente Java con il bytecode. Le informazioni offerte sono: i nomi delle variabili locali e i riferimenti tra le istruzioni di bytecode e le istruzioni del codice sorgente, le informazioni sono memorizzate in apposite strutture della sezione Methods info.
Il prodotto della compilazione dei programmi Java risulta essere una struttura complessa di non immediata interpretazione; per questo motivo al fine di estrapolare le informazioni necessarie alla comprensione del bytecode dai file ".class" abbiamo utilizzato una libreria di uno strumento esterno fornitoci: JABA (Java Architecture for Bytecode Analysis).

4.1.2 Linguaggio dell'esecutore simbolico

Al fine dell'interpretazione, una volta ottenuto il codice sorgente, è necessario conoscerne la struttura e conoscere la struttura delle singole istruzioni.
Le istruzioni del linguaggio bytecode seguono una struttura del tipo:
<Opcode> <operand1> <operand2> <...>
Ogni elemento della struttura ha una dimensione di 1 byte; le istruzioni sono generate in fase di compilazione e non contengono riferimenti generati a tempo di esecuzione ma solo riferimenti statici. Le istruzioni che coinvolgono variabili dinamiche, vengono implementate facendo uso di istruzioni per il passaggio parametri tramite *l'operand stack.*
<Opcode> (operator code) è il nome dell'istruzione, identifica il tipo di algoritmo da utilizzare per eseguire l'istruzione.La dimensione dell'opcode permette di avere 256 istruzioni; in realtà la Java Virtual Machine non sfrutta pienamente questa possibilità, infatti ha un set di sole 201 istruzioni più 3 opcode riservati.
Il numero e il tipo di operandi dipendono dalla singola istruzione e sono definiti a tempo di compilazione, generalmente gli operandi sono riferimenti alla constant pool oppure costanti. Le istruzioni che necessitano di operandi dinamici, quali variabili locali e riferimenti a classi, sfruttano l'operand stack come mezzo per la memorizzazzione di risultati parziali e passaggio parametri. Questo fatto rende il

funzionamento della Java Virtual Machine quello tipico di una macchina a stack e viene così eliminata la fase di valutazione delle variabili.

Prendiamo come esempio questa semplice classe:

```
public class EsempioClasse
{
      public void EsempioMetodo()
      {
            int a=0,b=0,c=0;
            if ((a==0) || (b>c))
                  a=b+3;
            else
                  c=0;
            a=b+c;
      }
}
```

La compilazione produce il seguente bytecode:

```
Method EsempioClasse()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void EsempioMetodo()
   0 iconst_0 // Costante 0 messa sull'operand stack
   1 istore_1 // Memorizzazione nella variabile in posizione 1
   2 iconst_0 // Costante 0 messa sull'operand stack
   3 istore_2 // Memorizzazione nella variabile in posizione 2
   4 iconst_0 // Costante 0 messa sull'operand stack
   5 istore_3 // Memorizzazione nella variabile in posizione 1
   6 iload_1  // Variabile 1 messa sull'operand stack
   7 ifeq 15  // Salta a istruzione 15 se uguale a 0
  10 iload_2  // Variabile 2 messa sull'operand stack
  11 iload_3  // Variabile 3 messa sull'operand stack
```

```
12 if_icmple 22 // Confronto tra i 2 valori
15 iload_2  // Variabile 2 messa sull'operand stack
16 iconst_3 // Costante 3 messa sull'operand stack
17 iadd     // Somma
18 istore_1 // Memorizzazione nella variabile in posizione 1
19 goto 24  // Salto incondizionato a istruzione 24
22 iconst_0 // Costante 0 messa sull'operand stack
23 istore_3 // Memorizzazione nella variabile in posizione 3
24 iload_2  // Variabile 2 messa sull'operand stack
25 iload_3  // Variabile 3 messa sull'operand stack
26 iadd     // Somma
27 istore_1 // Memorizzazione nella variabile in posizione 3
28 return   // Fine
```

Dall'esempio risulta evidente che non è presente nessun riferimento a nomi di variabili, queste vengono identificate grazie alla posizione nella tabella delle variabili locali.
Provando ad eseguire manualmente il codice ci si rende conto che l'esecuzione risulta assai semplice ed immediata, questo grazie al fatto che il bytecode è un linguaggio di livello intermedio.

Una possibile classificazione delle istruzioni del byte code della Java Virtual Machine è la seguente:

- Istruzioni per l'*Operand stack*
  Sono tutte le istruzioni che permettono di memorizzare costanti, variabili locali o riferimenti alla constant pool sull'operand stack e tutte quelle istruzioni che eliminano e duplicano parti dell'operand stack.
- Istruzioni per la *Local Variable Table*
  Sono tutte le istruzioni che permettono di memorizzare nella tabella delle variabili locali i valori presenti sull'operand stack
- Operazioni
  Sono tutti i tipi di istruzioni che prendono generalmente valori dall'operand stack e memorizzano il risultato dell'operazione nuovamente sull'operand stack.
  Le operazioni che possono essere effettuate sono, le quattro operazioni aritmetriche per tutti i diversi tipi di dati primitivi ammessi dalla JVM, operazioni unarie quali shift e casting e tutte le operazioni booleane bit a bit.

- Istruzioni per il controllo di flusso
  Sono tutte le istruzioni di salto, sia condizionato che incondizionato, le istruzioni di salto a subroutine e le istruzioni di ritorno, in questa sezione possiamo includere anche le istruzioni per il lancio delle eccezioni.
- Istruzioni per chiamate ai metodi e riferimento agli attributi
  Queste istruzioni permettono di chiamare i metodi e di accedere agli attributi di una classe. Non vi sono distinzioni tra attributi e metodi locali della classe stessa e quelli di altre classi. Le chiamate richiedono sempre l'esplicitazione dell'oggetto su cui devono essere eseguite.
  Ci sono 4 tipi diversi di invocazione di metodi, ogni tipo gestisce aspetti diversi delle chiamate a metodi.
- Istruzioni per la sincronizzazione dei thread
  Sono le istruzioni che sfruttano la tecnica del monitor per ovviare al problema della concorrenza tra thread.
- Istruzioni per l'allocazione dinamica
  Sono istruzioni che richiedono l'allocazione di nuovi oggetti nella zona di memoria dello heap. In particolare le istruzioni new per oggetti, array e array multidimensionali.
  Nel bytecode l'allocazione di un nuovo oggetto e la chiamata al suo costruttore vengono tradotti con un'istruzione di allocazione e la chiamata al metodo costruttore dell'oggetto appena allocato
- Istruzioni di servizio
  Sono istruzioni che permettono di verificare il tipo degli oggetti e sono istruzioni aggiuntive per l'inserimento di breakpoint all'interno del codice.

Al fine della gestione delle eccezioni la Java Virtual Machine prevede l'uso di tabelle contenenti gli indici dei segmenti di codice dei costrutti try e catch. Leggendo in modo sequenziale il codice che contiene il costrutto "try catch" le istruzioni del blocco catch sono isolate da un salto incondizionato. La presenza di tale blocco viene segnalata dalla tabella delle eccezioni presente nelle informazioni del metodo. Quando viene sollevata un'eccezione la Java Virtual Machine risale la catena di invocazioni finché viene individuata nel metodo attivo una tabella delle eccezioni.

4.2 Progettazione

Individuato ed analizzato il linguaggio sul quale dovrà agire l'esecutore simbolico, diretta conseguenza è pensare al modo in cui realizzare la simulazione.
Partendo dai file strutturati contenenti bytecode, e conoscendo le specifiche di un interprete capace di eseguire il bytecode, abbiamo ipotizzato di costruire una Java Virtual Machine estesa, capace di trattare non solo variabili numeriche ma anche variabili simboliche.
La progettazione può essere suddivisa in quattro diverse fasi:

- Progettazione della strutture dati
- Progettazione degli algoritmi di controllo
- Progettazione dell'interfacciamento con l'esterno
- Progettazione delle interfacce con i componenti esterni

4.2.1 Progettazione delle strutture dati

Partendo dalla struttura della Java Virtual machine, estendendola dove necessario, abbiamo progettato tutte le strutture dati necessarie a mantenere informazioni utili per l'esecuzione simbolica [Appendice B].
Come previsto dalle specifiche della Java Virtual Machine, la nostra struttura di memoria si basa sul componente principale rappresentato dallo *heap*. Contrariamente a quanto previsto assumiamo che non vi siano algoritmi di garbage collection per diversi motivi: sarebbero inutili in quanto la tecnica di esecuzione simbolica è solo la simulazione di una vera e propria esecuzione. Spesso la tecnica viene applicata solo a un modulo del programma sotto esame, difficilmente viene utilizzata su tutto il programma contemporaneamente. La complessità di prevedere un algoritmo di garbage collection per la nostra memoria sarebbe più dispendiosa del guadagno effettivo risultante.

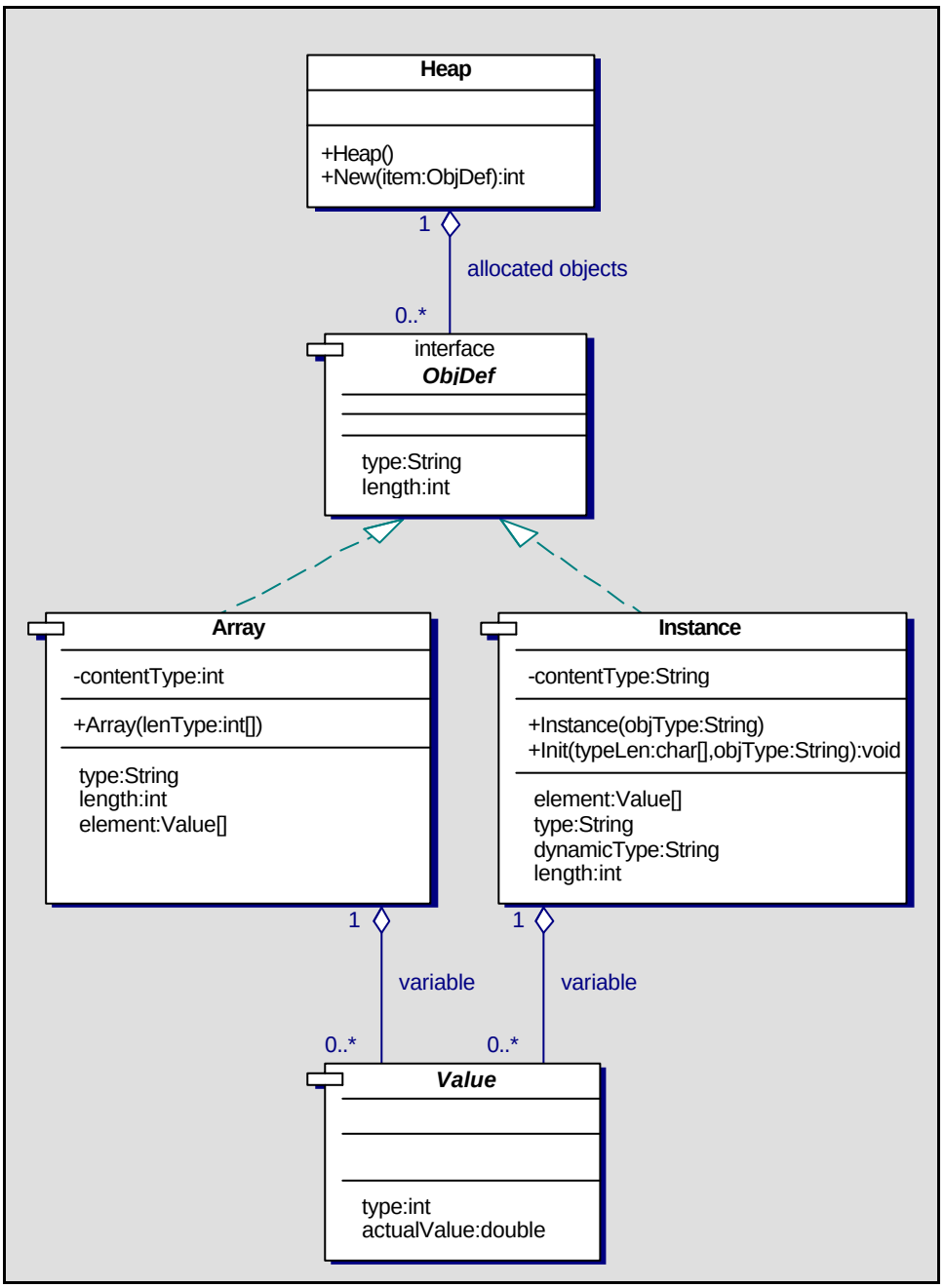


Figura 2: diagramma UML della strutture dati per allocazioni in memoria di massa.

Lo heap ha la funzione di allocare tutte le strutture dati necessarie all'immagazzinamento degli oggetti creati, in particolare può contenere *array* e

oggetti; gli array come in tutti i linguaggi di programmazione sono celle contigue di memoria adatti per la memorizzazione di valori tutti dello stesso tipo; nelle specifiche della Java Virtual Machine non viene definito come vengono memorizzate le informazioni per gli array e per gli oggetti, così è stato necessario progettare un tipo di dato capace di rappresentarli. Abbiamo progettato due oggetti che implementano la medesima interfaccia; semplificando notevolmente l'implementazione dello heap.

Il Frame stack è la struttura dati prevista per contenere tutti i dati necessari per la memorizzazione delle variabili e dei risultati parziali. Queste informazioni sono strutturate all'interno dei *frame*, e il frame stack in realtà contiene frame.

I frame contengono le strutture necessarie ai metodi, per ogni metodo attivato viene creato un nuovo frame, ogni frame contiene la tabella delle variabili locali (*LVSE*) e l'operand stack (*OPSE*) del metodo, e una cella utile per memorizzare valore di ritorno restituito dal metodo.

La tabella delle variabili locali contiene le variabili locali dichiarate per un singolo metodo, le variabili passate come argomento al metodo e l'handle "this": il riferimento all'oggetto a cui il metodo fa riferimento. Una variabile è definita da una coppia <nome,valore>, queste sono state rappresentate tramite un apposita struttura dati.

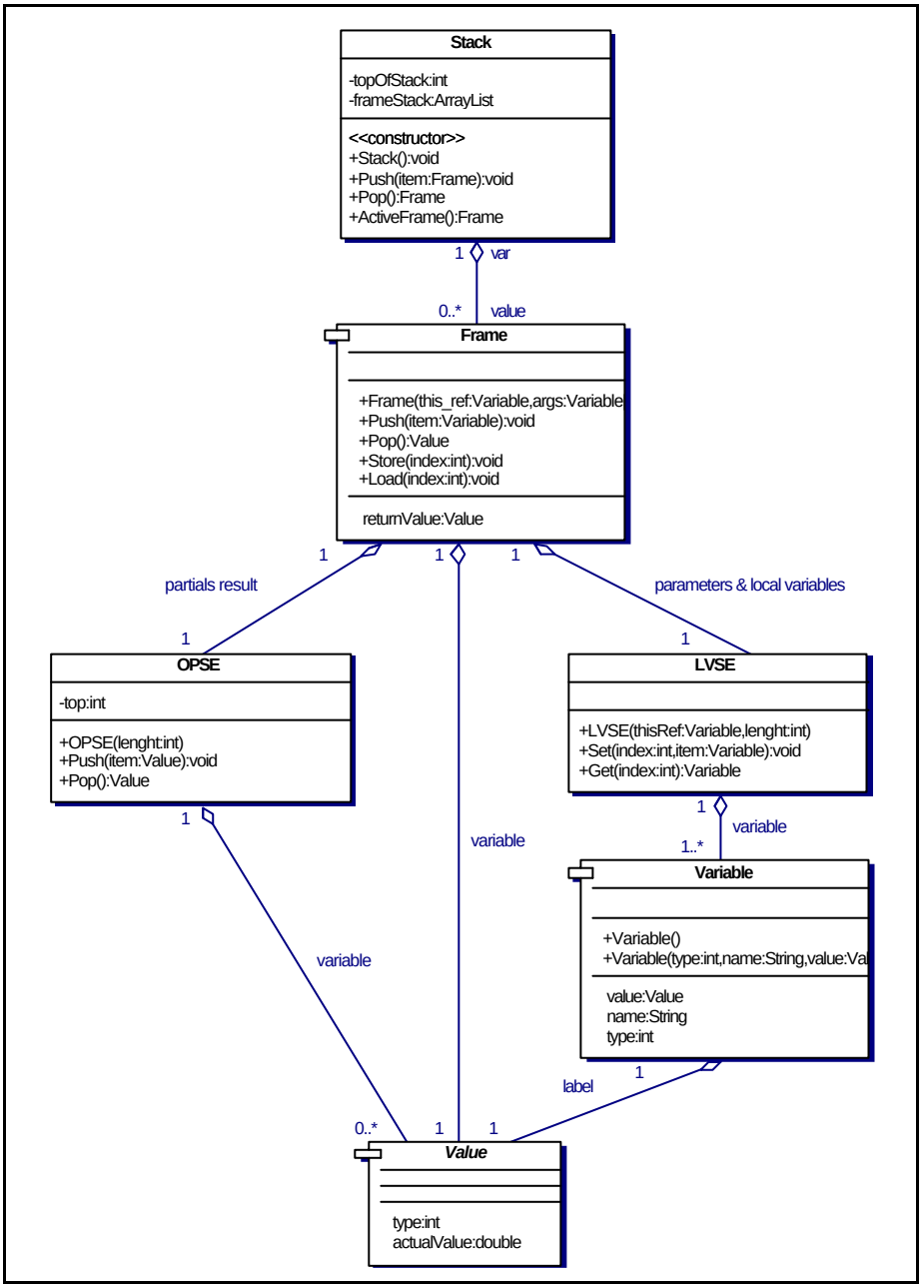


Figura 3: diagramma uml della strutture dati rappresentanti i frame

Estendendo la Java Virtual Machine abbiamo dovuto ridefinire anche i concetti di variabile e di valore: una variabile è definita da una coppia <nome,valore>; i valori possono essere di due tipi: primitive type e reference type.

I primitive type sono tutti i tipi primitivi di valori previsti dalla JVM, appositamente estesi con un tipo aggiuntivo necessario alla rappresentazione di valori simbolici.
I reference type sono indici validi per lo heap da noi definito in cui vengono allocati nuovi oggetti. I reference type non possono essere valori simbolici perché nell'esecuzione simbolica sarebbe inutile e assurdo dare un valore simbolico a una variabile di tipo reference.
L'operand stack è una struttura a pila che memorizza solo valori, esattamente come quello definito dalla JVM, l'operand stack è la struttura dati necessaria per il passaggio dei parametri alle singole istruzioni di byte code.

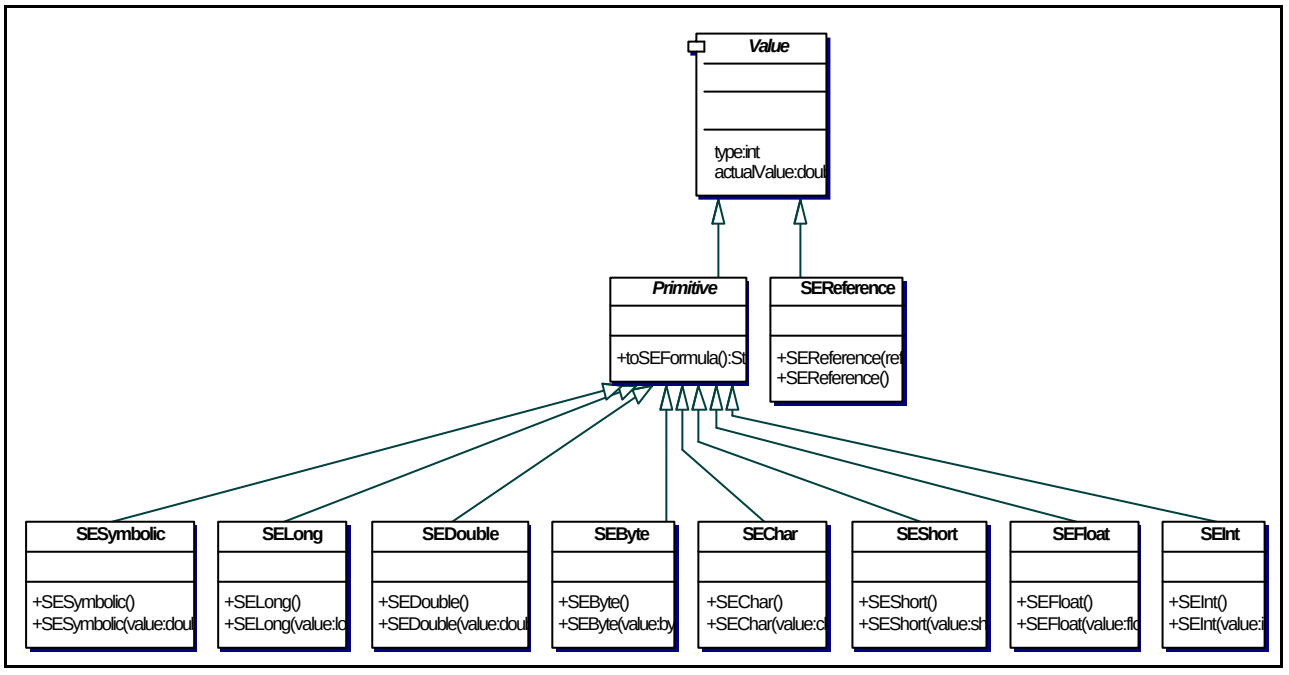


Figura 4: diagramma UML delle strutture dati dei tipi di valori supportati dall'esecutore simbolico.

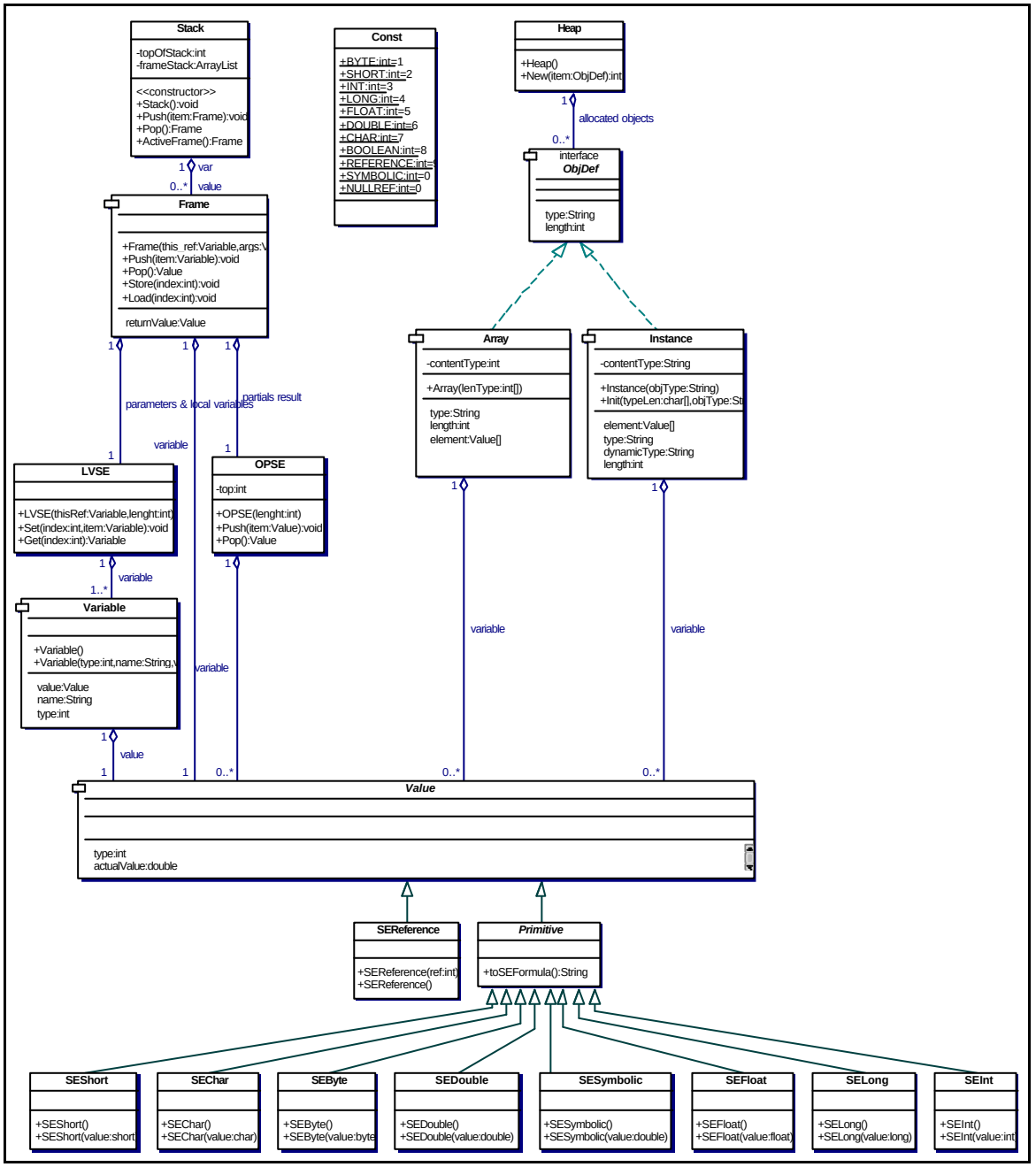


Figura 5: diagramma UML riassuntivo delle strutture dati

4.2.2 Progettazione componenti dell'esecutore simbolico

La progettazione del componente principale, come quella delle strutture dati è stata pensata partendo dalla struttura della Java Virtual Machine e dal suo modo di eseguire i programmi.
Sono stati individuati quattro componenti [Appendice B], che si occupano di compiti differenti; tramite la loro interazione e l'utilizzo delle strutture dati necessarie viene realizzata la tecnica di esecuzione simbolica.

- SEExecutor: si occupa della gestione globale del programma, è il primo componente che viene utilizzato, gestisce le interazioni tra le varie classi di un cluster. Offre i servizi di esecuzione simbolica di una singola istruzione, di esecuzione simbolica di un determinato metodo e di visualizzazzione dello stato dell'esecuzione. Quando viene richiesta l'esecuzione simbolica di un programma o di un metodo di un programma il componente SEExecutor si occupa di recuperare le informazioni relative al package e alle classi che dovranno essere eseguite simbolicamente.
- SEClass: si occupa della gestione di una sola classe, gestisce i riferimenti tra metodi diversi della stessa classe. Offre i servizi di esecuzione simbolica passo passo, di esecuzione simbolica di un singolo metodo e di visualizzazzione dello stato dell'esecuzione; si occupa di recuperare le informazioni e di allocare la memoria necessaria per l'esecuzione simbolica dei metodi.
- SEMethod: si occupa della gestione di un singolo metodo e dei riferimenti interni allo stesso. Si occupa di eseguire i cammini necessari in accordo con le analisi precedentemente svolte, questo componente è in legame stretto con il motore di esecuzione simbolica, gestisce l'instruction pointer (IP) e richiede l'esecuzione di ogni singola istruzione a SEE.
- SEE: è il vero e proprio motore di esecuzione simbolica. È il componente che implementa gli algoritmi di esecuzione simbolica, gestendo così l'esecuzione simbolica di una singola istruzione. Ha accesso diretto al frame attivo del metodo e allo heap, ne modifica lo stato simbolico eseguendo gli algoritmi applicati alle istruzioni. È il componente che sfrutta maggiormente i servizi forniti dal semplificatore.

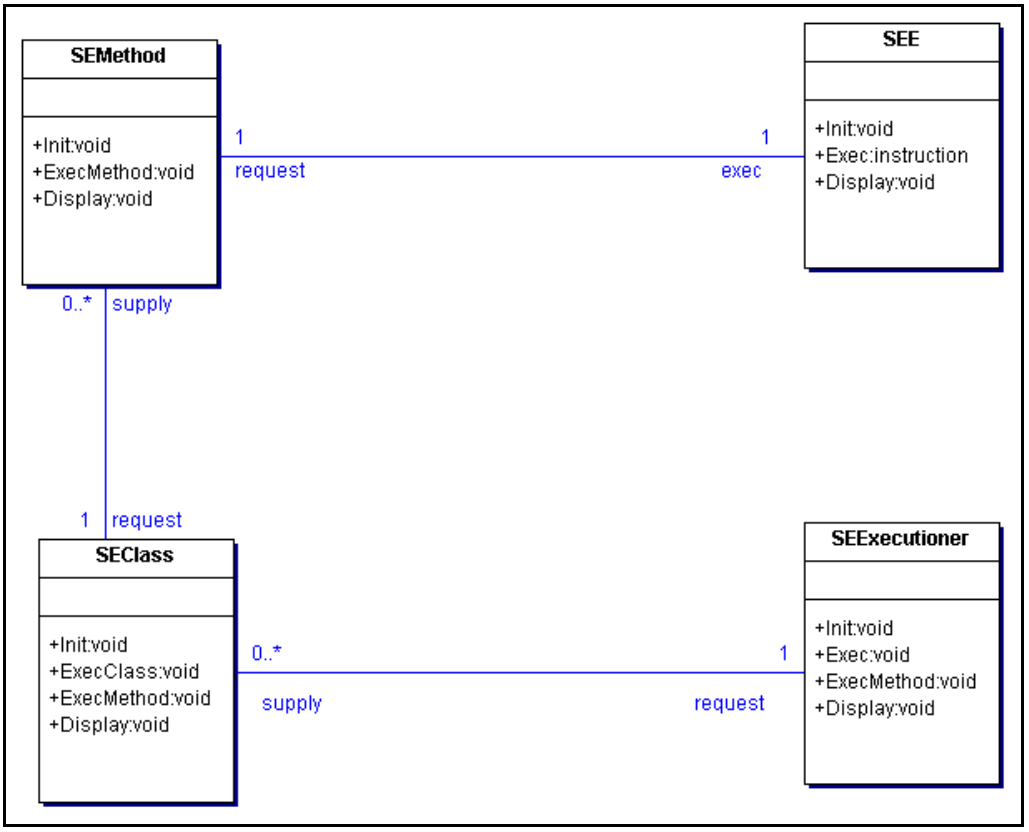


Figura 6: diagramma UML componenti esecutore simbolico.

4.2.3 Interazione dei componenti

La tecnica di esecuzione simbolica viene realizzata tramite l'interazione dei diversi componenti.

L'esecuzione simbolica richiede una fase di inzializzazione durante la quale vengono definiti gli stati iniziali delle variabili e le precondizioni all'esecuzione del codice.

Sono state previste due modalità di esecuzione simbolica:

- la prima modalità prevede che l'esecuzione simbolica avvenga passo passo, una singola istruzione alla volta, in questo modo si possono costantemente controllare i cambiamenti che avvengono nello stato dell'esecuzione.
- la seconda modalità prevede che l'esecuzione simbolica sia effettuata su un intero metodo prima di produrre risultati.

L'idea di prevedere queste due modalità nasce dal tipico utilizzo che si può fare dei risultati dell'esecuzione simbolica, infatti tipicamente è interessante analizzare

sia risultati prodotti dall'esecuzione simbolica su un singolo metodo, sia i risultati ottenuti dall'esecuzione passo passo delle singole istruzioni.
L'esecuzione simbolica prevede in entrambe le sue modalità di effettuare delle operazioni d'interazione coi componenti, indispensabili per il raggiungimento degli obiettivi.
L'esecuzione prevede come primo passo l'inizializzazzione di tutti i componenti e delle strutture dati necessarie per il recupero delle informazioni necessarie dai file ".class" tramite l'utilizzo dello strumento JABA.
La vera e propria simulazione prevede una collaborazione tra i componenti dello strumento; i quattro componenti che eseguono compiti diversi, interagiscono tra loro creando una catena di esecuzione. La richiesta di esecuzione di un determinato metodo di una determinata classe, viene passata inizialmente al componente SEExecutor, il quale recupera le informazioni necessarie e gestirà tutti i futuri riferimenti tra diverse classi. Il componente SEExecutor, creerà un componente SEClass per la classe a cui appartiene il determinato metodo da eseguire, al quale passerà poi la richiesta di esecuzione; SEClass gestirà tutti i riferimenti tra i vari metodi della stessa classe. Il componente SEClass creerà a sua volta un componente SEMethod per il metodo da eseguire, al quale passerà la richiesta di esecuzione; SEMethod gestirà tutti i riferimenti tra le istruzioni all'interno dello stesso metodo.Infine il componente SEMethod creerà un componente SEE che avrà il compito di eseguire la prima istruzione del metodo da eseguire.
Le possibilità relative all'istruzione da eseguire sono:

- se l'istruzione sarà un'istruzione semplice che non prevede interazioni particolari oppure un istruzione di salto, verrà eseguita e il controllo tornerà normalmente al componente SEMethod
- se l'istruzione sarà un'invocazione a un'altro metodo della stessa classe, la catena verrà ripercorsa per passare il controllo al componente SEClass che creerà un'altra catena per il metodo invocato
- se l'istruzione sarà un'invocazione a un'altro metodo di un'altra classe, la catena verrà ripercorsa per passare il controllo al componente SEExecutor che creerà una nuova catena per l'esecuzione del metodo invocato.

Questa strutturazione gerarchica dei componenti creerà a tempo di esecuzione un albero che come radice possiede un solo componente SEExecutor, in cui per ogni metodo invocato esiste il relativo componente SEMethod che lo gestisce.L'idea è che il controllo venga indirizzato a chi è in grado di gestire i riferimenti richiesti, questo per non avere un appiattimento dei componenti.

Un'altra caratteristica fondamentale dell'esecuzione simbolica deve esser presa in considerazione: quando l'esecuzione arriva a un punto di decisione, non sempre si può identificare in modo univoco il percorso da seguire, ma spesso bisogna fare ipotesi sui valori simbolici. Per questo motivo vengono introdotte diverse modalità di esecuzione, che seguono varie politiche, che vengono seguite quando non è univocamente identificata la veridicità o meno della condizione.
Le possibili politiche da seguire in questi casi sono:

- Assumere sempre vera la proposizione
- Assumere sempre falsa la proposizione
- Chiedere l'interazione di un componente esterno

Il componente esterno può essere sia un utente sia un componente software che può indicare il cammino da percorrere.
Le politiche da seguire vengono specificate al momento dell'inizializzazione; per quanto riguarda la modalità di esecuzione passo passo, le politiche possono essere modificate lungo il percorso.

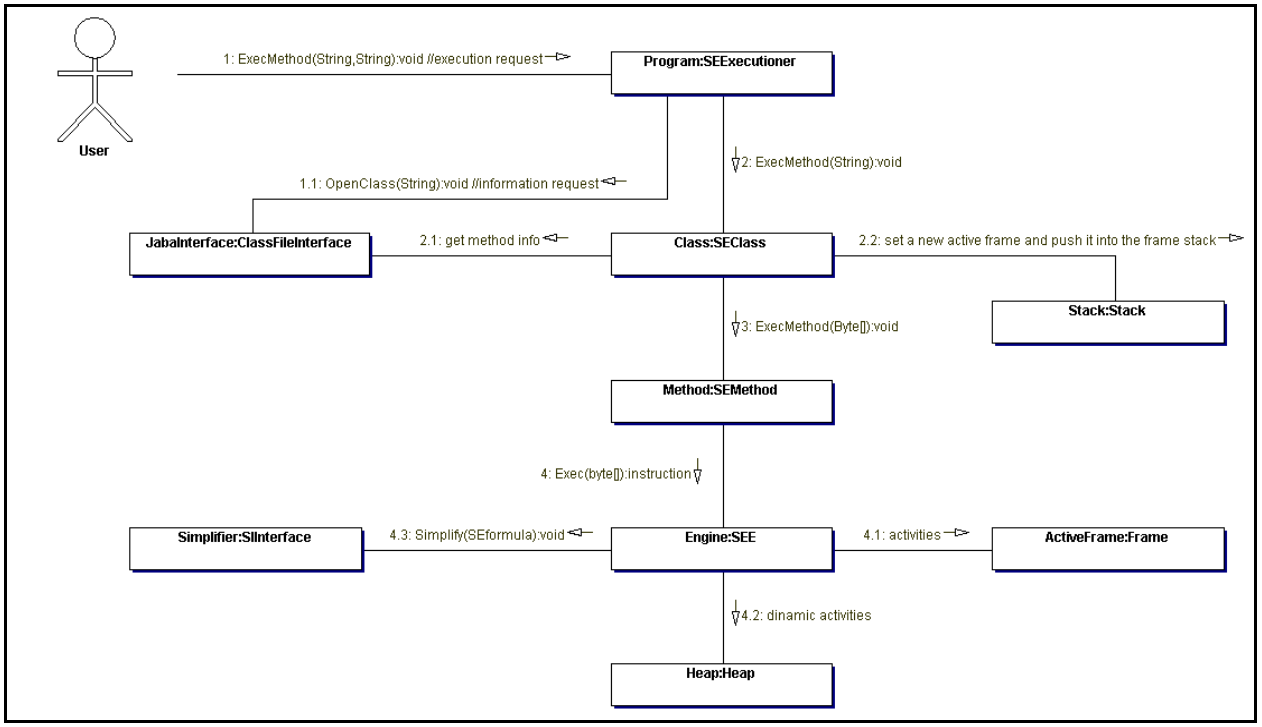


Figura 7: Interazione dei componenti

4.2.4 Interfaccia

I servizi forniti dall'esecutore simbolico possono essere utilizzati tramite la sua interfaccia. L'interfaccia fornisce in particolare quattro funzioni:

- **Init**
- **Exec**
- **ExecMethods**
- **Display**

Init permette l'inizializzazione dello stato simbolico secondo le modalità richieste dall'esecuzione simbolica prevista. Il servizio, data in ingresso l'informazione di quale metodo si vuole effettuare l'esecuzione simbolica, inizializza i componenti necessari. Tramite questa funzione si può specificare anche la path condition iniziale che può essere diversa dal valore "true" nel caso siano note ipotesi sui valori iniziali dei simboli.
Oltre alla specifica della path condition la funzione crea le strutture dati necessarie all'esecuzione; in particolare viene creato lo stack dello strumento, utilizzato poi per contenere i frame di ogni metodo, viene creato il frame del metodo richiesto e viene caricato sullo stack come frame attivo; la funzione infine inizializza la tabella delle variabili locali del metodo.
La funzione restituisce al chiamante un controllo sulla corretta esecuzione.

Exec svolge la fondamentale funzione richiesta all'esecutore simbolico, l'esecuzione di un passo di esecuzione simbolica.
Data in ingresso l'istruzione che si vuole eseguire, lo strumento seleziona l'opportuno algoritmo di esecuzione simbolica tramite l'invocazione dell'apposita funzione di esecuzione. Dopo aver effettuato gli opportuni cambiamenti allo stato simbolico la funzione ritorna il controllo al chiamante, restituendo il puntatore alla prossima istruzione.
Come già visto quando si raggiunge un punto di decisione, l'esecuzione simbolica necessita di ipotesi sui valori dei simboli per proseguire; tali ipotesi dipendono dal ramo del programma in cui si vuole proseguire.
Per questo motivo possono essere utilizzate diverse politiche per la scelta delle ipotesi, queste politiche possono essere impostate dall'esterno sia a priori, sia in fase di esecuzione.

ExecMethod svolge la funzione di esecuzione di un intero metodo, questa funzione è solo un estensione della funzione implementata da Exec, viene introdotta in quanto risulta molto utile avere il risultato dell'esecuzione simbolica

su un intero metodo, infatti pur non essendo l'unità minima sulla quale può essere applicata l'esecuzione simbolica, un singolo metodo è la tipica unità di analisi.
In questo caso però, per avere dei risultati dall'esecuzione simbolica complessiva di un metodo, devono essere previste obbligatoriamente delle politiche da seguire per quanto riguarda le ipotesi sui valori dei simboli quando si arriva ad un punto di decisione.

Display svolge la funzione di interrogazione dello stato simbolico. Permette di recuperare tutte le informazioni sullo stato simbolico attuale. La funzione è utile sia usata in un esecuzione passo passo effettuata in collaborazione con la funzione Exec, sia in collaborazione con la funzione ExecMethod, per visualizzare lo stato simbolico prodotto da quest'ultima. La funzione non ha parametri di ingresso e restituisce lo stato simbolico in un apposita struttura.

4.2.5 Limitazioni dello strumento

In fase di progettazione sono state anche affrontate tutte le problematiche riguardanti la tecnica di esecuzione simbolica in relazione allo specifico linguaggio, individuando le limitazioni da imporre allo strumento.
Le limitazioni individuate sono:

- I cicli sono implementati con l'uso di variabili. Le variabili durante l'esecuzione simbolica non hanno un valore sempre determinato; questo può provocare troppo spesso l'insorgere di cicli infiniti provocati dalle difficoltà di interpretazione delle condizioni di esecuzioni di un ciclo. Per ovviare al problema si può decidere in precedenza quante iterazioni eseguire per ogni ciclo.
- La ricorsione analogamente a quanto detto per i cicli, può provocare l'insorgere di infinite chiamate ricorsive. Per risolvere il problema, per il metodo ricorsivo, si devono conoscere in precedenza le post-condizioni.
- Il valore di un reference non può essere simbolico; un reference è un puntatore a una struttura in memoria. Dare un valore simbolico a un reference equivarrebbe a affidare al reference una zona di memoria non tenendo conto delle strutture in essa contenute. Il problema può essere facilmente risolto supponendo che i reference non possano assumere valori simbolici.
- Una valida caratteristica degli array è che essi vengono spesso indicizzati da variabili. Le variabili, durante l'esecuzione simbolica, assumono valori non sempre determinabili, conseguentemente non è sempre determinabile l'elemento dell'array a cui si fa riferimento. Per ovviare al problema

quando si fa riferimento ad un array, lo stato simbolico viene memorizzato per l'intero array anziché per ogni singolo elemento.

4.3 Componenti utilizzati

Durante la fase di realizzazione dell'esecutore simbolico sono stati acquisiti due strumenti utili ai compiti di costruzione di una struttura contente le informazioni necessarie, partendo dal bytecode, e di semplificazione di formule matematiche.

4.3.1 JABA

4.3.1.1 Lo strumento JABA

JABA(Java Architecture for Bytecode Analysis) è un insieme di librerie per l'analisi di programmi scritti in Java a livello del bytecode. In realtà JABA è una API per l'analisi di programmi scritti in Java tramite l'analisi del bytecode. In particolare si compone di diverse librerie che hanno particolari compiti all'interno dell'analisi del codice.

Al fine di decidere se utilizzare lo strumento JABA o un altro strumento per il recupero di informazioni, abbiamo effettuato una analisi di tutte le librerie dello strumento per individuare quelle utili all'esecuzione simbolica. Inoltre abbiamo effettuato vari test al fine di valutare le prestazioni dello strumento.[Appendice C]

Durante la fase di realizzazione dell'esecutore simbolico abbiamo utilizzato lo strumento JABA per il recupero del bytecode e di tutte quelle informazioni indispensabili all'esecuzione di un programma Java.

Per il recupero del bytecode e delle informazioni di esecuzione, abbiamo utilizzato una singola libreria di JABA, la libreria da noi utilizzata è la libreria "classfile". Questa libreria permette di costruire una struttura dati che seguendo le specifiche della Java Virtual Machine contenga tutte le informazioni, contenute in un file ".class", necessarie per l'esecuzione del programma.

La libreria "classfile" è l'unica libreria di rilevante interesse nell'ambito della realizzazione dell'esecutore simbolico, infatti abbiamo utilizzato lo strumento JABA solo come lettore di file ".class" e non come analizzatore di bytecode.

La libreria viene utilizzata tramite un interfaccia appositamente progettata. L'interfaccia è stata inserita per mantenere l'indipendenza dallo strumento, cosicché sia possibile, in futuro, sostituire lo strumento JABA con un qualsiasi altro strumento che fornisce le stesse funzionalità di recupero informazioni da file ".class", riscrivendo solo l'interfaccia senza modificare l'esecutore simbolico.

L’interfaccia è stata poi estesa per permettere il recupero di informazioni da più file “.class” così da permettere l’analisi di un cluster di classi.

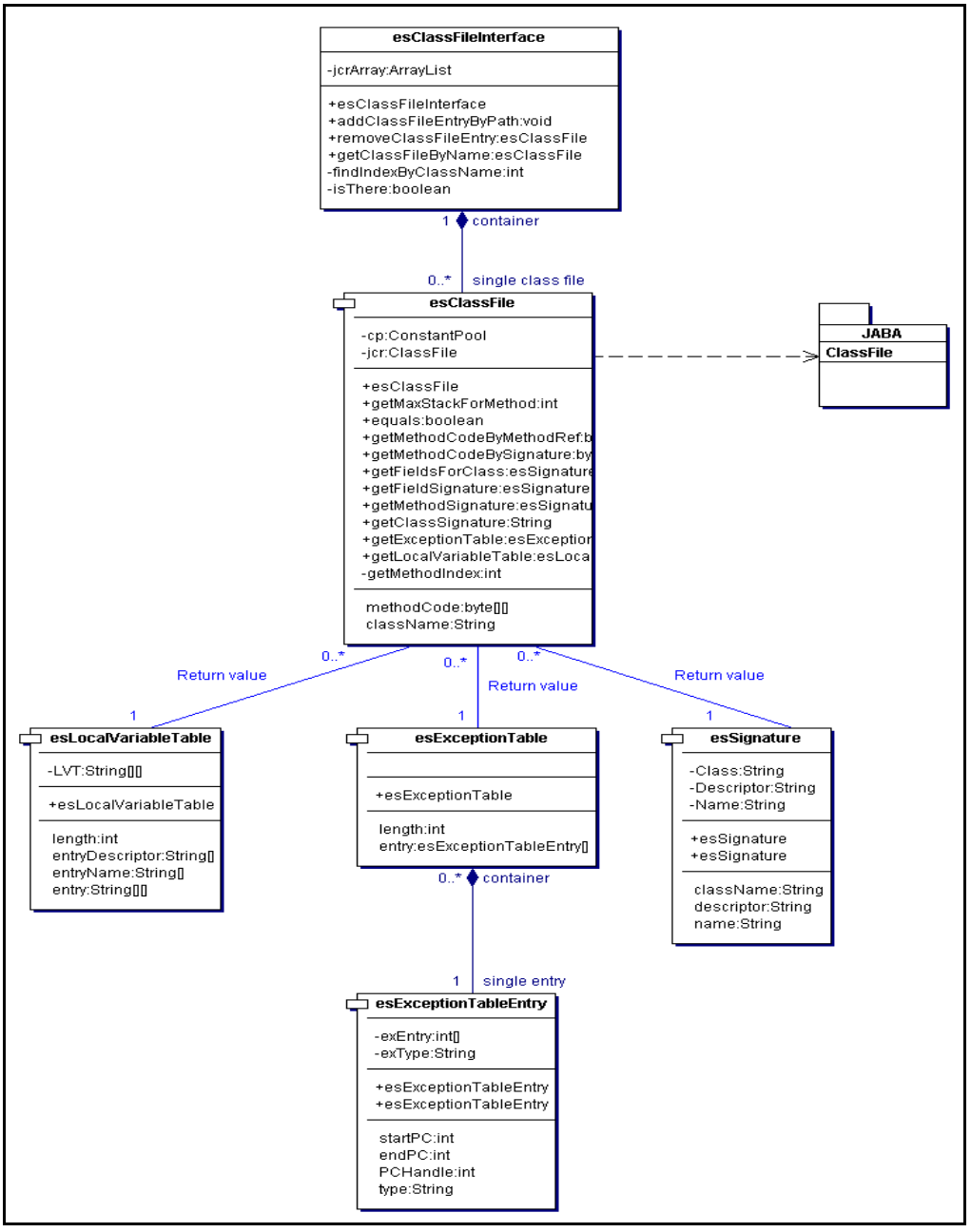


Figura 8: diagramma UML interfacciamento con strumento JABA

4.3.1.2 Integrazione di JABA

Per utilizzare lo strumento JABA abbiamo creato un interfaccia studiata appositamente. Le caratteristiche essenziali richieste all'interfaccia sono:

- mantenere l'indipendenza dello strumento, al fine di realizzare un esecutore simbolico che utilizzi JABA, ma che non sia basato sulle sue strutture dati.
- recuperare tutte le informazioni necessarie, cioè fare già una prima selezione delle informazioni utili alla tecnica di esecuzione simbolica, eliminando quelle superflue.
- fornire strutture dati, utilizzabili dall'esecutore simbolico, per il mantenimento delle informazioni necessarie allo stesso.

Le caratteristiche richieste sono state tenute in considerazione sia in fase di progettazione sia in fase di realizzazione dell'interfaccia.

L'interfaccia si compone di cinque classi:

esClassFileInterface: si occupa di gestire diversi esClassFile, permette cioè di recuperare contemporaneamente informazioni da diversi file .class e mantenerle in apposite strutture dati distinte. Permette di effettuare l'analisi di un completo cluster di classi.

esClassFile: prevede la struttura dati necessaria a mantenere tutte le informazioni recuperate da un singolo file ".class". È l'interfaccia vera e propria con cui l'utilizzatore di JABA andrà a interagire per il recupero di tutte e sole le informazioni necessarie. Si basa sugli altri componenti per mantenere l'indipendenza dallo strumento. Fornisce tutti i metodi per ottenere le informazioni distinte.

esLocalVariableTable: classe che fornisce la struttura dati necessaria a mantenere le informazioni relative alla tabella delle variabili locali, fornisce anche tutti i servizi atti al recupero delle informazioni sulle variabili locali.

esExceptionTable: fornisce i servizi e la struttura dati necessaria a recuperare le informazioni su tutte le eccezioni presenti nel codice. Le informazioni sono strutturate in un array di esExceptionTableEntry.

esExceptionTableEntry: si occupa di fornire i servizi e la struttura dati necessarie al recupero e alla gestione di informazioni riguardo alle singole eccezioni.

esSignature: rappresenta la segnatura di un metodo di un campo o di una classe. È una struttura dati necessaria per la comunicazione sia tra utente e JABA, sia per la comunicazione tra i componenti.

L'insieme dei componenti rende possibile raggiungere le caratteristiche richieste all' interfaccia.

4.3.2 Semplificatore

Per mantenere costantemente aggiornati i valori simbolici delle entità (variabili di stato, parametri, variabili locali) coinvolte nell'esecuzione del codice, e quindi più facilmente valutabili, l'esecutore simbolico si deve appoggiare a delle funzioni che svolgano questo compito. La valutazione delle path condition a sua volta risulta essere un compito difficile, non sempre è immediatamente possibile; a volte è possibile valutare le path condition solo grazie a una semplificazione delle espressioni. La semplificazione delle espressioni d'altra parte non è un compito di facile risoluzione, anzi è un compito risolvibile solo tramite strumenti di alta complessità. Da questo problema nasce l'idea di appoggiarsi ad uno strumento esterno per quanto riguarda sia la semplificazione delle path condition, sia il compito di mantenere aggiornati i valori simbolici delle entità coinvolte.

Lo strumento esterno acquisito è Sicstus Prolog, un interprete Prolog sviluppato dal SICS (Swedish Institute of Computer Science) composto da due librerie dedicate al *costraint solving*, che offrono funzionalità di supporto alla semplificazione delle formule logiche.

L'interprete Sicstus Prolog già utilizzato nell'ambito del progetto TWO, studiato in [6], viene da noi esteso, per il suo utilizzo, con un interfaccia. L'interfaccia è indispensabile in questo caso per due motivi: anzitutto mantenere l'indipendenza dello strumento, cosi da poter in futuro sostituire l'interprete Sicstus Prolog con qualsiasi altro semplificatore senza dover modificare il codice intero dell'esecutore simbolico ma andando solo a riscrivere l'interfaccia; in secondo luogo il semplificatore è un interprete scritto in prolog interrogato tramite un interfaccia scritta in C, il componente Simplifier; per questo motivo è necessario scrivere un'interfaccia che utilizzando il codice nativo permetta l'interazione tra l'esecutore simbolico scritto in Java e il componente scritto in C.

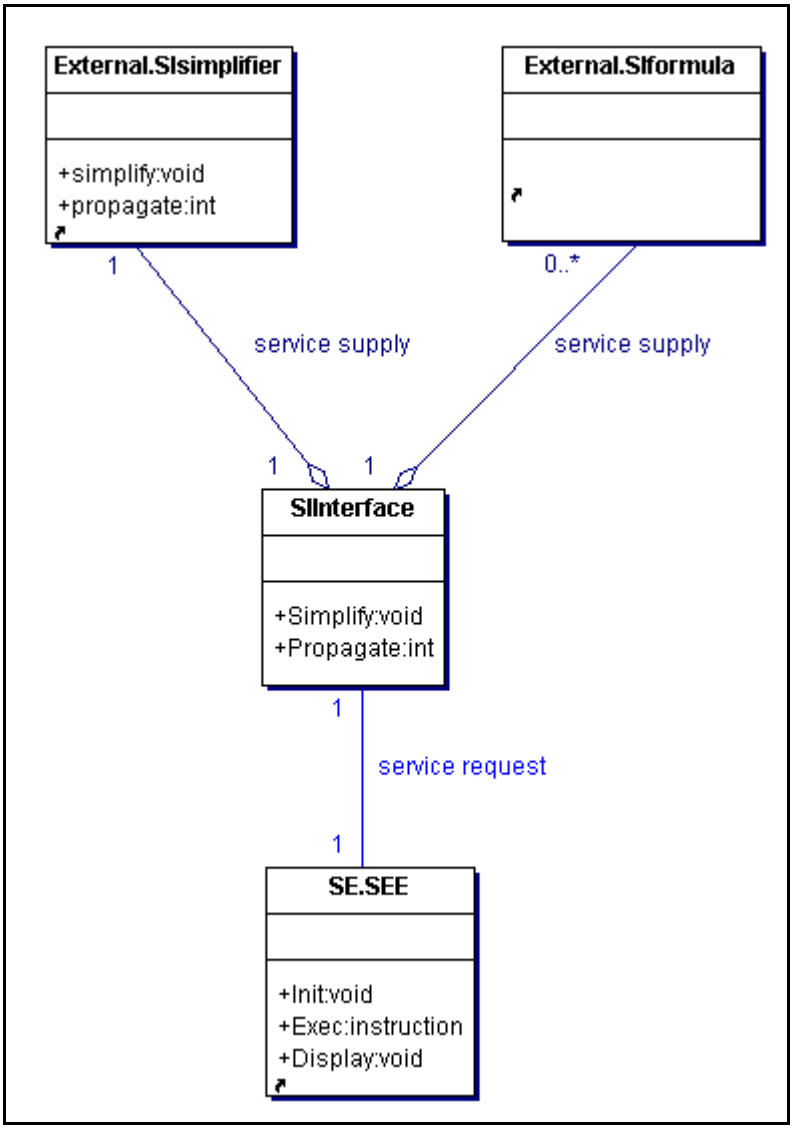


Figura 9: diagramma UML interfacciamento con strumento semplificatore.

Capitolo 5
Conclusioni e sviluppi futuri

La tecnica formale di esecuzione simbolica, utile nella modellazione e nell'analisi del software, era finora rivolta all'analisi di software scritto in linguaggi tradizionali.
Lo scopo originale di questa tesi era la progettazione di uno strumento che permettesse di applicare la tecnica di esecuzione simbolica a software scritto in linguaggio Java.
Il lavoro di analisi ha esplorato il contesto della tecnica e la realizzabilità di uno strumento efficace individuandone le limitazioni; la fase di progettazione di un prototipo ha esplorato in profondità il funzionamento di un componente adatto all'esecuzione simbolica e le sue problematiche, portando a una definizione precisa del modello da cui partire nella fase di implementazione.
Durante la tesi abbiamo anche realizzato un primo sottoinsieme del prototipo che comprende l'interfaccia utente, l'interfaccia verso il bytecode e la gestione della memoria.
Tramite il prototipo attuale, è possibile recuperare tutte le informazioni dai programmi ed utilizzare la memoria del componente.
Gli sviluppi futuri prevedono il completamento dell'opera di implementazione del prototipo nelle sue componenti mancanti. In particolare, si procederà ad analizzare ed applicare gli algoritmi di esecuzione simbolica.
Infine il testing del componente verrà effettuata tramite applicazione su casi reali, al fine di poter individuare le reali capacità del prototipo per poi inserirlo in uno strumento per applicazioni di più largo ambito.

Bibliografia

[1] U. Buy, A. Orso and M. Pezzè. Automated Testing of Classes. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '00), Agosto 2000.

[2] A. Coen-Porisini, G. Denaro, C. Ghezzi and M. Pezzè. Using Symbolic Execution for Verifying Safety-Critical System. In Proceedings of the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE01), ACM Software Engineering Notes, Settembre 2001.

[3] A. Coen-Porisini, F. De Paoli, C. Grezzi and D. Mandrioli. Software Specialization Via Symbolic Execution. In IEEE Transactions on Software Engineering, SE17(9): 884-889, Settembre 1991.

[4] A. Orso. Integration Testing of Object-Oriented Software. PhD Thesis, Politecnico di Milano, 1998.

[5] A. Sicolo. Automated Formula Evaluation and Simplification: an Application to Object Oriented Structural Test. Tesi di master, Master of Science in Electrical Engineering and Computer Science, 2001.

[6] M. Pezzè, M. Young. Software Test and Analysis: Process, Principles, and Techniques. Pubblicazione prevista per il 2003. Disponibile a www.cs.uoregon.edu/~michal/book/

[7] R. K. Doong , P. G. Frankl. The ASTOOT Approach to Testing Object-oriented Programs. ACM Transaction on Software Engineering and Methodology, 3(2): 101-130, Aprile 1994.

[8] C. D. Turner, D. J. Robson. The State-based Testing of Object-oriented Programs. In International Conference on Software Maintenance, IEEE Society Press, 302-310, Settembre 1993.

[9] P. Jorgensen, C. Erickson. Object-oriented Integration Testing. Communications of the ACM, 37(9): 30-38, Settembre 1994.

[10] M. R. Girgis. An Experimental Evaluation of a Symbolic Execution System. IEEE Software Engineering Journal, 7(4): 285-290, Luglio 1992.

[11] A. J. Offut, E. J. Seaman. Using Symbolic Execution to Aid Automatic Test Data Generation. In Compass '90: 5th Annual Conference on Computer Assurance, 12-21, 1990.

[12] L. K. Dillon. Using Symbolic Execution for Verification of Ada Tasking Programs. ACM Transactions on Programming Languages and Systems, 12(4): 643-669, Ottobre 1990.

[13] L. K. Dillon, R. A. Kemmerer, L. J. Harrison. An experience with two symbolic execution-based approaches to formal verification of Ada tasking programs. In Proceedings of the Second Workshop on Software Testing, Verification and Analysis, 114-122, 1988.

[14] J. Gosling, B. Joy, G. Steele, G. Bracha. Java™ Language Specification, Second Edition. Giugno 2000.

[15] T. Lindholm, F. Yellin. The Java™ Virtual Machine Specification, Second Edition. Aprile 1999.

[16] C. Ghezzi, M. Jazayeri. Programming Language Concepts. Giugno 1997.

[17] Edison Design Group. http://www.edg.com

Appendice A

Analisi del bytecode

L'analisi è stata svolta in via sperimentale, scrivendo programmi, compilandoli e studiando il bytecode corrispondente. L'analisi inizia da una semplice classe(classe punto) composta da solo alcuni metodi e con costrutti poco compless e continua con la creazione di classi sempre più specializzate con costrutti sempre più complessi. L'obbiettivo dell'analisi è capire in che modo le istruzioni di codice Java vengono tradotte in istruzioni di bytecode.
Nel documento sono presenti commenti che cercano di dare la visione del problema per l'esecuzione simbolica.

Classe punto.
È una classe semplice che rappresenta un punto nello spazio cartesiano.
La classe possiede due variabili private di tipo intero, due costruttori e 4 funzioni di incapsulamento, due delle quali hanno dei costrutti per il controllo del flusso al fine di verificarne la traduzione in byte code.
Il metodo test ci serve per verificare come vengono trattati i parametri passati per parametro e le variabili locali.

```
public class punto
{
	private int x,y;


//Costruttori
	punto()
	{
		x=0;
		y=1;
	}

	punto(int x,int y)
	{
		this.x=x;
		this.y=y;
	}
```

```
//Metodi
	public boolean movex(int spost)
	{
	if (spost<5 && spost>-5)
		{
		x+=spost;
		return(true);
		}
	return(false);
	}
	public boolean movey(int spost)
	{
	if (spost<5 && spost>-5)
		{
		y+=spost;
		return(true);
		}
	return(false);
	}
	public int getx()
	{
	return(x);
	}

	public int gety()
	{
	return(y);
	}

//Questo metodo serve per verificare la differenza tra variabile passata per parametro e
variabile locale	public void test(int test_variable)
	{
	int i,result=0;
	if (test_variable>0)
		for (i=0;i<test_variable;i++)
			result+=i;
	else
		for (i=test_variable;i>0;i--)
			result*=i;
	}

}
```

Lo stack viene diviso per ogni singolo metodo in un frame che contiene i parametri le variabili locali e un operand stack che serve per passare i parametri alle istruzioni della JVM.

```
Compiled from punto.java
public class punto extends java.lang.Object {
   punto();
   punto(int,int);
   public boolean movex(int);
   public boolean movey(int);
   public int getx();
```

```
    public int gety();
}

Method punto()
   0 aload_0          ->puntatore this
   1 invokespecial #1 <Method java.lang.Object()> -> metodo 1 dell'oggetto this
   4 aload_0
   5 iconst_0                    --> mette una costante nello stack degli operandi
   6 putfield #2 <Field int x>              --> la carica nel campo 2 che corrisponde al campo
della variabile X
   9 aload_0
  10 iconst_1
  11 putfield #3 <Field int y>
  14 return

Qui vediamo che i parametri sono considerati come delle variabili locali.
Method punto(int,int)
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 aload_0
   5 iload_1                     --> carica la variabile locale (che è stata passata per parametro)
   6 putfield #2 <Field int x>              --> la registra nel capo 2 che corrisponde alla variabile
X
   9 aload_0
  10 iload_2
  11 putfield #3 <Field int y>
  14 return

Method boolean movex(int)
   0 iload_1          -> carico nell'OP la variabile passata per parametro.
   1 iconst_5         -> carico il valore 5
   2 if_icmpge 23  -> chiamo un'istruzione di confronto con un jump
   5 iload_1
   6 bipush -5
   8 if_icmple 23
  11 aload_0         -> puntatore this.
  12 dup
  13 getfield #2 <Field int x>             -> campo 2 del puntatore this.
  16 iload_1                    -> parametro
  17 iadd                       -> somma tra parametro e X
  18 putfield #2 <Field int x>              ->salvataggio del risultato.
  21 iconst_1
  22 ireturn
  23 iconst_0
  24 ireturn

*Questa funzione si comporta nello stesso modo di movex(int)
Method boolean movey(int)
   0 iload_1
   1 iconst_5
   2 if_icmpge 23
   5 iload_1
   6 bipush -5
   8 if_icmple 23
  11 aload_0
  12 dup
  13 getfield #3 <Field int y>
```

```
 16 iload_1
 17 iadd
 18 putfield #3 <Field int y>
 21 iconst_1
 22 ireturn
 23 iconst_0
 24 ireturn

Method int getx()
  0 aload_0
  1 getfield #2 <Field int x>
  4 ireturn

Method int gety()
  0 aload_0
  1 getfield #3 <Field int y>
  4 ireturn

Method void test(int)
  0 iconst_0
  1 istore_3        -> inizializzo la variabile 3 (result)
  2 iload_1
  3 ifle 26                 -> se parametro 1 minore uguale a 0 salto a 26
  6 iconst_0
  7 istore_2        ->inizializzo variabile 2 (i)
  8 goto 18
 11 iload_3         result
 12 iload_2         i
 13 iadd            i + result
 14 istore_3        dentro result
 15 iinc 2 1        i++
 18 iload_2         confronto
 19 iload_1
 20 if_icmplt 11    -> se parametro 2 <1 salta a 11 rimane in ciclo finchè i < test_variable
 23 goto 42
 26 iload_1
 27 istore_2         i = test_var
 28 goto 38
 31 iload_3         result
 32 iload_2         i
 33 imul            i * result
 34 istore_3        ->result
 35 iinc 2 -1       incremento i di -1
 38 iload_2         i
 39 ifgt 31         i > 0 -> 31
 42 return
```

javap.exe mette nelle parentesi <> i riferimenti simbolici alle funzioni e alle variabili di cui tiene traccia nel file .class analizzato. I riferimenti sono indici della constant pool.

I parametri locali e passati per parametri non hanno nessun legame simbolico con il codice sorgente.Solo nel caso di una compilazione particolare, con informazioni aggiuntive per il debug, possiamo avere i

nomi delle variabili. E’ da notare che le variabili hanno gli indici nell'ordine in cui sono stati dichiarate all'interno del metodo.
Le funzioni e gli attributi di una classe vengono identificati dal reference all'oggetto al quale l’istruzione deve far riferimento. Ne abbiamo un esempio prima delle istruzioni invokespecial e getfield.
L'istruzione aload carica sull’operand stack una variabile presente all’indice specificato nell’istruzione. La variabile che viene caricata è una variabile di tipo reference. La posizione 0 della local variable contiene sempre il puntatore this.

Esecuzione simbolica:
In questo codice non troviamo particolari problemi per un ipotetico motore di esecuzione simbolica.
Ci rendiamo subito conto che gli algoritmi di esecuzione simbolica per questo tipo di programma non sono particolarmente complessi, notiamo anche che il funzionamento della JVM deve essere simile a quello di una macchina a stack. Non sono mai presenti, almeno fino ad ora, espressioni in cui si rende necessaria una fase di valutazione.

Overloading

Questa classe utilizza la classe punto già definita.

In questa classe proviamo ad utilizzare altri oggetti. Vedremo come questi vengono allocati e come vengono richiesti loro servizi.

Questa classe contiene 3 metodi (test) overloadati tra loro. Vedremo così come viene implementato l'overloading dei metodi. Vengono richiamati diversi metodi per verificare come avviene la risoluzione dei nomi.

```
import punto;

public abstract class figura
{
	protected punto pntPrimary,pntSecondary;

	// Costruttori
	public figura()
	{
		pntPrimary=new punto(0,0);
		pntSecondary=new punto(0,0);
	}

	public figura(int x1, int y1, int x2, int y2)
	{
		pntPrimary=new punto(x1,y1);
		pntSecondary=new punto(x2,y2);
	}

	public figura(punto pntP1,punto pntP2)
	{
		pntPrimary=pntP1;
		pntSecondary=pntP2;
	}

	//Fine dei Costruttori
	public int geth()
	{
		int a,b;
		test();
		test(6);
		test('h');
		a=pntPrimary.gety();
		b=pntSecondary.gety();
		if(a<b)
			return(b-a);
		return(a-b);
	}


	public int getl()
	{
		int a,b;
		test();
		test(3);
		test('c');
```

```
                a=pntPrimary.getx();
                b=pntSecondary.getx();
                if(a<b)
                        return(b-a);
                return(a-b);
        }

        private void test()
        {
                punto p1;
                int a,b,c;
                char q,w,e,r,t,y;
                double l,i,o,n;
                a=b=c=0;
                p1=new punto(a,b);
                q=w=e=r=t=y='f';
                l=i=o=n=6;
        }

        public void test(int test_variable)
        {
                int a,b,c;
                char q,w,e,r,t,y;
                double l,i,o,n;
                a=b=c=100;
                q=w=e=r=t=y='g';
                l=i=o=n=63;
        }

        public void test(char test_variable)
        {
                char b;
                b=test_variable;
                pntPrimary.test(5);
        }

        public abstract int iCalcolaArea();
}

Compiled from figura.java
public abstract class figura extends java.lang.Object {
  protected punto pntPrimary;
  protected punto pntSecondary;
  public figura();
  public figura(int,int,int,int);
  public figura(punto,punto);
  public int geth();
  public int getl();
  public void test(int);
  public abstract int iCalcolaArea();
}

Method figura()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0         -> per la putfield che prende il puntatore all'oggetto a cui ci si riferisce
  5 new #2 <Class punto> -> lascia il puntatore all'oggeto allocato sullo stack
```

```
  8 dup                         -> lo dupplica per passarlo al costruttore
  9 iconst_0                    -> parametro
 10 iconst_0                    ->parametro
 11 invokespecial #3 <Method punto(int,int)>->costruttore
 14 putfield #4 <Field punto pntPrimary>  -> prende il primo riferimento(della new) e lo lega
alla variabile 4 dell'oggetto
 17 aload_0                     ->per la putfield
 18 new #2 <Class punto>-> idem
 21 dup
 22 iconst_0
 23 iconst_0
 24 invokespecial #3 <Method punto(int,int)>-> metodo 3 dell'oggetto della classe punto
 27 putfield #5 <Field punto pntSecondary>
 30 return

Method figura(int,int,int,int)
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0
  5 new #2 <Class punto>
  8 dup
  9 iload_1
 10 iload_2
 11 invokespecial #3 <Method punto(int,int)>
 14 putfield #4 <Field punto pntPrimary>
 17 aload_0
 18 new #2 <Class punto>
 21 dup
 22 iload_3
 23 iload 4
 25 invokespecial #3 <Method punto(int,int)>
 28 putfield #5 <Field punto pntSecondary>
 31 return

Method figura(punto,punto)
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0         this
  5 aload_1         parametro1 (punto)
  6 putfield #4 <Field punto pntPrimary>       ->nel campo 4
  9 aload_0
 10 aload_2
 11 putfield #5 <Field punto pntSecondary>  -> nel campo 5
 14 return

Method int geth()
  0 aload_0
  1 invokespecial #6 <Method void test()>     -> Questa era una funzione privata è stata
usata una invokespecial
  4 aload_0
  5 bipush 6
  7 invokevirtual #7 <Method void test(int)>   -> in questa viene usata invokevirtual
 10 aload_0
 11 bipush 104
 13 invokevirtual #8 <Method void test(char)>
 16 aload_0
 17 getfield #4 <Field punto pntPrimary>
```

```
 20 invokevirtual #9 <Method int gety()>        -> chiamo gety per pntPrimary
 23 istore_1
 24 aload_0
 25 getfield #5 <Field punto pntSecondary>
 28 invokevirtual #9 <Method int gety()>        -> per pntSecondary
 31 istore_2
 32 iload_1
 33 iload_2
 34 if_icmpge 41
 37 iload_2
 38 iload_1
 39 isub
 40 ireturn
 41 iload_1
 42 iload_2
 43 isub
 44 ireturn

Method int getl()
  0 aload_0
  1 invokespecial #6 <Method void test()>
  4 aload_0
  5 iconst_3
  6 invokevirtual #7 <Method void test(int)>
  9 aload_0
 10 bipush 99
 12 invokevirtual #8 <Method void test(char)>
 15 aload_0
 16 getfield #4 <Field punto pntPrimary>
 19 invokevirtual #10 <Method int getx()>
 22 istore_1
 23 aload_0
 24 getfield #5 <Field punto pntSecondary>
 27 invokevirtual #10 <Method int getx()>
 30 istore_2
 31 iload_1
 32 iload_2
 33 if_icmpge 40
 36 iload_2
 37 iload_1
 38 isub
 39 ireturn
 40 iload_1
 41 iload_2
 42 isub
 43 ireturn

Method void test()
  0 iconst_0
  1 dup
  2 istore 4          c=0
  4 dup
  5 istore_3          b=0
  6 istore_2          a=0
  7 new #2 <Class punto>              -> creazione dell'oggetto punto
 10 dup
 11 iload_2           a
```

```
 12 iload_3         b
 13 invokespecial #3 <Method punto(int,int)> -> costruttore (a,b)
 16 astore_1        ref registrato nella var 1
 17 bipush 102      inizio inizializzazione caratteri
 19 dup
 20 istore 10
 22 dup
 23 istore 9
 25 dup
 26 istore 8
 28 dup
 29 istore 7
 31 dup
 32 istore 6
 34 istore 5        fine caratteri
 36 ldc2_w #11 <Double 6.0>         inizializzazione double
 39 dup2
 40 dstore 17
 42 dup2
 43 dstore 15
 45 dup2
 46 dstore 13
 48 dstore 11                       fine double
 50 return


Method void test(int)          -> idem senza oggetto
  0 bipush 100
  2 dup
  3 istore 4
  5 dup
  6 istore_3
  7 istore_2
  8 bipush 103
 10 dup
 11 istore 10
 13 dup
 14 istore 9
 16 dup
 17 istore 8
 19 dup
 20 istore 7
 22 dup
 23 istore 6
 25 istore 5
 27 ldc2_w #13 <Double 63.0>
 30 dup2
 31 dstore 17
 33 dup2
 34 dstore 15
 36 dup2
 37 dstore 13
 39 dstore 11
 41 return

Method void test(char)
  0 iload_1         parametro
```

```
 1 istore_2       var1=parametro
 2 aload_0        this
 3 getfield #4 <Field punto pntPrimary>                this.getfield
 6 iconst_5
 7 invokevirtual #15 <Method void test(int)>
10 return
```

La risoluzione dei nomi dei metodi come abbiamo verificato avviene fatto a tempo di compilazione. Tutti i metodi, infatti risultano memorizzati all'interno della struttura del class file come nome e parametri formali più tipo di valore di ritorno.
Possiamo vedere che per richiamare i metodi non sono sempre utilizzate le stesse istruzioni. Invokevirtual serve per richiamare metodi pubblici, invokespecial viene utilizzato per costruttori e metodi privati e per gestire i metodi ereditati.
I riferimenti ai metodi che troviamo all'interno del codice sono riferimenti alla constant pool. La constant pool contiene tutti i nomi di tutti i metodi e gli attributi richiamati all'interno del codice. Non troviamo però il nome di tutti i metodi, nel caso infatti che un metodo non venga richiamato all'interno del codice della classe non ne troveremo il riferimento all'interno della constant pool. Ne troveremo comunque tutte le informazioni utili in un'altra sezione del class file.

Abbiamo verificato le modalità di allocazione di nuovi oggetti, possiamo notare che la chiamata al costruttore viene gestita come una chiamata a metodo. La fase di allocazione e la chiamata al costruttore sono in realtà due passi ben distinte.

Anche per questo tipo di programmi sembra che non ci possano essere grossi problemi a livello di esecuzione simbolica. È importante riuscire a strutturare al meglio la memoria per riuscire ad eseguire correttamente il codice (soprattutto per l'allocazione di nuovi oggetti)

Ereditarietà
Viene presentata una classe che specializza quella precedente sovrascrivendo un metodo della classe padre, questo per capire come viene gestita a livello di bytecode. I costruttori richiamano i costruttori della classe genitore; spesso all'interno del codice viene sfruttata l'ereditarietà della nuova classe.

```
import punto;

public class rettangolo extends figura
{
```

```
	// Costruttori
	public rettangolo()
	{
		super(0,0,1,1);
	}

	public rettangolo(int x1, int y1, int x2, int y2)
	{
		super(x1,y1,x2,y2);
	}

	public rettangolo(punto pntP1,punto pntP2)
	{
		super(pntP1,pntP2);
	}
	//Fine dei Costruttori

	//Riscrittura metodo astratto
	public int iCalcolaArea()
	{
		return(super.geth()*super.getl());
	}

	public int iCalcolaPerimetro()
	{
		return(super.geth()*2+super.getl()*2);
	}

	private void test()
	{
		super.test(5);
	}



	public void test(char test_var)
	{
		char a;
		rettangolo b;
		b= new rettangolo();
		//Voglio invocare la funzione test della classe padre
		b.test(5);
		test();
		a=test_var;
	}

	private void test(int a,int b)
	{
	if (a>0 && b==0)
		test('b');
	}

}
```

```
Compiled from rettangolo.java
public class rettangolo extends figura {
   public rettangolo();
   public rettangolo(int,int,int,int);
   public rettangolo(punto,punto);
   public int iCalcolaArea();
   public int iCalcolaPerimetro();
   public void test(char);
}


Method rettangolo()
  0 aload_0
  1 iconst_0
  2 iconst_0
  3 iconst_1
  4 iconst_1
  5 invokespecial #1 <Method figura(int,int,int,int)>
  8 return

Method rettangolo(int,int,int,int)
  0 aload_0
  1 iload_1
  2 iload_2
  3 iload_3
  4 iload 4
  6 invokespecial #1 <Method figura(int,int,int,int)>
  9 return

Method rettangolo(punto,punto)
  0 aload_0
  1 aload_1
  2 aload_2
  3 invokespecial #2 <Method figura(punto,punto)>
  6 return

Method int iCalcolaArea()
  0 aload_0
  1 invokespecial #3 <Method int geth()>  richiama il metodo geth() suo (in realtà il metodo
è ereditato)
  4 aload_0
  5 invokespecial #4 <Method int getl()>
  8 imul
  9 ireturn

Method int iCalcolaPerimetro()
  0 aload_0
  1 invokespecial #3 <Method int geth()> prende l'altezza
  4 iconst_2
  5 imul                    Moltiplica per 2
  6 aload_0
  7 invokespecial #4 <Method int getl()> prende la larghezza
 10 iconst_2
 11 imul                    moltiplica per 2
 12 iadd                    Somma (le due moltiplicazioni si trovavano già nell'OP)
 13 ireturn                 restituisce quello che si trova sull'OP
```

```
Method void test()
   0 aload_0
   1 iconst_5
   2 invokespecial #5 <Method void test(int)>          Invoca il metodo passandogli 5
   5 return

Method void test(char)
   0 new #6 <Class rettangolo>                 Nuova allocazione
   3 dup
   4 invokespecial #7 <Method rettangolo()>    Chiamata del costruttore
   7 astore_3                                  Il ref viene salvato nella variabile 3
   8 aload_3
   9 iconst_5
  10 invokevirtual #5 <Method void test(int)>          di quell rettangolo viene richiamato il
metodo test
  13 aload_0
  14 invokespecial #8 <Method void test()>             il metodo test della classe stessa
(this)
  17 iload_1
  18 istore_2
  19 return

Method void test(int, int)
   0 iload_1
   1 ifle 14
   4 iload_2
   5 ifne 14
   8 aload_0
   9 bipush 98                 Carattere ascii
  11 invokevirtual #9 <Method void test(char)>          chiamata a this.test(98)
  14 return
```

Abbiamo verificato come le condizioni vengano spezzate in più parti; questo può rendere più semplice la valutazione dei predicati.
Abbiamo verificato come viene risolta l'ereditarietà. Questa modalità potrebbe potrebbe far nascere delle difficoltà in fase di esecuzione simbolica, ma il fatto che ci siano diversi tipi di chiamate potrebbe semplificare il problema.
Finora non abbiamo incontrato grossi problemi per la realizzazzione di un esecutore simbolico per il bytecode.

Polimorfismo.
Le classi che seguono sono state scritte al fine di verificare come vengono risolte le chiamate polimorfe in bytecode.

Abbiamo una classe astratta e una classe che ne eredita I metodi. Un metodo della classe derivata accetta come parametro un'oggetto del tipo della classe padre.

```
public abstract class astratta
{
```

```
	private int x;
	public int y;

	public astratta()
	{
		x=0;
		y=0;
	}

	public astratta(int x,int y)
	{
		this.x=x;
		this.y=y;
	}

	public abstract void metodo1(int a);
	public abstract void metodo1(char a);
	public abstract int metodo2();
	public abstract int metodo2(char a);
}

public class a extends astratta
{
	private int x;
	public int y;

	public a()
	{
		x=0;
		y=0;
	}

	public a(int x,int y)
	{
		this.x=x;
		this.y=y;
	}

	public void metodo1(int a)
	{
		x+=a;
	}

	public void metodo1(char a)
	{
		if (a=='a')
			x=y;
	}

	public int metodo2()
	{
		return(x);
	}


	public int metodo2(char a)
	{
```

```
                if (a=='a')
                        return(x+1);
                return(0);
        }

        public void test(astratta h)
        {
        h.metodo1(5);
        }
}
```

nella funzione test la risoluzione del binding della funzione è dinamico e a livello del bytecode non si può ricavare nessun tipo di informazione. Evidentemente tutte le informazioni utili alla risoluzione di questo problema stanno nella rappresentazione che è stata scelta per l'oggetto allocato in memoria. Per questo motivo sarà necessario progettare la memoria al fine di riuscire sempre a capire con quali oggetti si stia lavorando.

```
Compiled from a.java
public class a extends astratta {
   public int y;
   public a();
   public a(int,int);
   public void metodo1(int);
   public void metodo1(char);
   public int metodo2();
   public int metodo2(char);
   public void test(astratta);
}

Method a()
  0 aload_0
  1 invokespecial #1 <Method astratta()>
  4 aload_0
  5 iconst_0
  6 putfield #2 <Field int x>
  9 aload_0
 10 iconst_0
 11 putfield #3 <Field int y>
 14 return

Method a(int,int)
  0 aload_0
  1 invokespecial #1 <Method astratta()>
  4 aload_0
  5 iload_1
  6 putfield #2 <Field int x>
  9 aload_0
 10 iload_2
 11 putfield #3 <Field int y>
 14 return

Method void metodo1(int)
  0 aload_0
  1 dup
```

```
  2 getfield #2 <Field int x>
  5 iload_1
  6 iadd
  7 putfield #2 <Field int x>
 10 return

Method void metodo1(char)
  0 iload_1
  1 bipush 97
  3 if_icmpne 14
  6 aload_0
  7 aload_0
  8 getfield #3 <Field int y>
 11 putfield #2 <Field int x>
 14 return

Method int metodo2()
  0 aload_0
  1 getfield #2 <Field int x>
  4 ireturn

Method int metodo2(char)
  0 iload_1
  1 bipush 97
  3 if_icmpne 13
  6 aload_0
  7 getfield #2 <Field int x>
 10 iconst_1
 11 iadd
 12 ireturn
 13 iconst_0
 14 ireturn

Method void test(astratta)
  0 aload_1
  1 iconst_5
  2 invokevirtual #4 <Method void metodo1(int)>-> chiamo il metodo 1 dell'oggeto passato
per parametro
  5 return
```

Solamente vedendo la chiamata a questo metodo possiamo sapere di quale oggetto stiamo chiamando il metodo.

Eccezioni

Questa classe viene introdotta per verificare come vengono generate e gestite le eccezioni.

In questa classe viene inoltre affrontata l'implementazione di array, sia di tipi primitivi che ti tipi complessi come oggetti.

```
public class ex
{
        private int x;
        public ex(int x) throws Exception
```

```
        {
        Exception e=new Exception();
                if (x==0) throw e;
        this.x=x;
        }

        public void divide(int a)
        {
        int result1,result2;
        result1=a/x; //non puo' dare eccezioni
        try
                {
                result2=x/a;
                }
        catch (Exception e)
                {
                System.out.println("Divisione per 0");
                }
        }

        public static void main(String args[])
        {
        int a[];
        ex array[];
        a= new int[2];
        a[0]=1;
        a[1]=2;
        array=new ex[2];
        try
                {
                array[1]=new ex(a[0]);
                array[0]=new ex(0);
                }
        catch (Exception ie)
                {
                System.out.println("Errore");
                }
        }
}

Compiled from ex.java
public class ex extends java.lang.Object {
   public ex(int) throws java.lang.Exception;
   public void divide(int);
   public static void main(java.lang.String[]);
}

Method ex(int)
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 new #2 <Class java.lang.Exception>
  7 dup
  8 invokespecial #3 <Method java.lang.Exception()> -> Oggetto Exception creato
 11 astore_2
 12 iload_1         -> parametro
 13 ifne 18         -> se diverso da 0 salto a 18
 16 aload_2         -> carica eccezione
```

```
 17 athrow          -> lancia eccezione
 18 aload_0
 19 iload_1
 20 putfield #4 <Field int x>
 23 return

Method void divide(int)
   0 iload_1
   1 aload_0
   2 getfield #4 <Field int x>
   5 idiv             -> divido i due membri
   6 istore_2         -> salva risultato
   7 aload_0
   8 getfield #4 <Field int x>          -> carico X
  11 iload_1                    -> carico parametro
  12 idiv                       -> divido
  13 istore_3                   -> salva risultato
  14 goto 27                    -> salto a 27
  17 astore 4
  19 getstatic #5 <Field java.io.PrintStream out>
  22 ldc #6 <String "Divisione per 0">
  24 invokevirtual #7 <Method void println(java.lang.String)>
  27 return
Exception table:
   from   to  target type
     7    14    17   <Class java.lang.Exception>
```

Tabella delle eccezioni dice quali parti del codice sono coperte da clausole try e catch dice che il codice da 7 a 14 è della clausola try e che il codice del catch parte dalla riga 17

```
Method void main(java.lang.String[])
   0 iconst_2
   1 newarray int               -> nuovo array 2 elementi
   3 astore_1                   -> nella variabile 1
   4 aload_1                    -> indirizzo della variabile 1
   5 iconst_0                   -> costante 0
   6 iconst_1                   -> costante 1
   7 iastore                    -> store nell'array registra 1 nella casella 0 dell'indirizzo di base
dell'array
   8 aload_1                    ->idem
   9 iconst_1
  10 iconst_2                   -> var1[1]=2
  11 iastore
  12 iconst_2                           -> 2
  13 anewarray class #8 <Class ex> -> nuovo array di puntatori a classe EX di 2 elementi
  16 astore_2                           -> nella var2
  17 aload_2                            -> var2 (servirà per la aastore alla riga 29)
  18 iconst_1                           -> 1(servirà per la aastore alla riga 29)
  19 new #8 <Class ex>                  -> nuovo classe ref nello stack
  22 dup                                -> ref dupplicato sullo stack
  23 aload_1                            -> var1
  24 iconst_0                           -> 0
  25 iaload                             -> carica var1[0]
```

```
 26 invokespecial #9 <Method ex(int)>          -> ex( var1[0])  va a togliere ref di riga 22 e
parametro costruito in 25
 29 aastore                          -> registra nel campo var2[1]
 30 aload_2                          -> var2
 31 iconst_0                         -> 0
 32 new #8 <Class ex>                -> stesso procedimento
 35 dup
 36 iconst_0
 37 invokespecial #9 <Method ex(int)>
 40 aastore                          -> fine allocazione array di oggetti
 41 goto 53                          ->fine
 44 astore_3
 45 getstatic #5 <Field java.io.PrintStream out>
 48 ldc #10 <String "Errore">
 50 invokevirtual #7 <Method void println(java.lang.String)>
 53 return
Exception table:
  from   to  target type
   17    41    44   <Class java.lang.Exception>
```

Stesso procedimento: nella tabella vengono indicate le parti di codice che si occupano della gestione delle eccezioni.

Appendice B

Analisi e test di prestazioni strumento JABA

JABA(Java Architecture for Bytecode Analysis) è un insieme di librerie per l'analisi di programmi scritti in Java a livello del bytecode. JABA si compone di 15 package, ognuno di essi ha un compito differente.

jaba.classfile: offre tutte le classi necessarie per il recupero di informazioni da file ".class".
Offre inoltre tutte le strutture dati atte all'immagazzinamento delle informazioni recuperate. Permette la lettura organizzata di file ".class". Ricostruisce esattamente la struttura definita nelle specifiche della Java Virtual Machine.

jaba.constants: contiene tutte le costanti necessarie agli altri package. In particolare contiene le costanti utili alla lettura del bytecode: tutti i modificatori di visibilità, gli opcode di tutte le istruzioni di bytecode.

jaba.dbHandlers: offre le classi per l'interfacciamento con database. In particolare permette di colloquiare con database per l'immagazzinamento di informazioni ottenute dalla costruzione di vari grafi.

jaba.du: presenta le classi necessarie alla rappresentazione di una definizione oppure di un uso per ogni costrutto di Java.

jaba.graph.*: questi sei package offrono i servizi necessari alla costruzione di grafi con informazioni ricavate dal bytecode. In particolare possono essere costruiti grafi generici oppure specifici. I grafi specifici previsti sono: Augmented Control-Flow Graph, Control Dependence Graph, Control-Flow Graph, Intermethod Control-Flow Graph.

jaba.gui: contiene le classi necessarie alla creazione dell'interfaccia grafica che viene fornita assieme alle librerie. L'interfaccia grafica non è indispensabile per

l’utilizzo di JABA, può risultare utile se si vuole usare JABA come strumento a sé stante.

jaba.instruction: presenta le classi necessarie a rappesentare tutte le istruzioni del bytecode.
Per ogni istruzione è prevista una classe che la rappresenta. Ogni classe permette di ottenere informazioni riguardo le singole istruzioni. Le informazioni che vengono fornite, per ogni istruzione, riguardano gli usi, le definizioni, gli operandi, e le eccezioni che essa può generare.

jaba.instruction.visitor: offre i servizi per la visita delle istruzioni di bytecode sotto esame, permette di passare in rassegna tutto il codice e di fornirne una rappresentazione simbolica.

jaba.main: package di interfacciamento offerto assieme alle librerie. Prevede alcune modalità di utilizzo standard di JABA. In particolare viene offerto un driver, già scritto, per la creazione dei diversi grafi.

jaba.sym: presenta tutti i servizi per la costruzione di una struttura dati contenente tutte le informazioni del codice sotto esame. Offre i servizi per analizzare tutte le classi direttamente o indirettamente usate dal programma analizzato. Permette di avere una struttura da cui partire per un analisi approfondita del bytecode.

Il test di JABA è stato affrontato utilizzando diversi driver appositamente scritti per mettere sotto esame i diversi servizi offerti. Il test è partito dalla semplice lettura di un class file per arrivare all`analisi completa di un package con costruzione dei relativi grafi. Il test è stato eseguito su file e package di diverse dimensioni e su piattaforme diverse.

I quattro package che sono stati analizzati, di dimensioni diverse, sono identificati dai relativi file rc:
-punto.rc composto solamente dalla classe punto
-test.rc composto dai due file forniti assieme a JABA Test.class e User.class
-rettangolo.rc composto dai file punto.class figura.class rettangolo.class
-giavamachia.rc composto da 15 file per una lunghezza complessiva di circa 1500 righe di codice

Successivamente è stato testato un driver di lettura di un solo class file sui filedi dimensioni diverse(MYJABA1):

-punto.class
-Test.class
-Robot.class
-Arena.class

Il test era basato sulla rilevazione del tempo di esecuzione e sulla quantita' di memoria utilizzata.
Le tre macchine utilizzate per il test sono:
-AMD Athlon 1400MHz, 640MB Ram, S.O: Linux Mandrake 8.0
-AMD Duron 800Mhz, 256MB Ram 133Mhz SO: Linux Mandrake 8.0
-Intel Celeron 333MHz, 128MB Ram, S.O: Linux Debian

I risultati del test hanno evidenziato che la lettura di uno o più file ".class" effettuata tramite il package jaba.classfile è veloce, infatti non ha mai occupato i vari processori per più di due secondi, e non è troppo costosa in quantità di memoria, infatti la memoria occupata varia tra i 7 e gli 8 Mb, anche per package molto grandi.
Al contrario l'analisi approfondita e la costruzione dei grafi risultano molto costosi. Per effettuare l'analisi approfondita viene costruita la symbol table che risulta molto pesante in termini di tempo, il tempo minimo per la costruzione della symbol table è circa 7 secondi su una piattaforma di nuova generazione, fino ad arrivare a passare il minuto su piattaforme meno potenti. Lo stesso discorso vale in termini di memoria, la costruzione della symbol table occupa almeno 30MB.
Questo dispendio di risorse nasce dal fatto che la symbol table viene costruita analizzando tutti i riferimenti possibili cioè andando a costruire una rappresentazione simbolica di tutte le classi collegate. Nei test veniva costruita andando ad analizzare anche tutte le librerie del JDK direttamente o indirettamente collegate.

Abbiamo successivamente analizzato approfonditamente i package per verificare il possibile utilizzo dei componenti nella fase di progettazione e realizzazione dell'esecutore simbolico.

I package per la costruzione di grafi non sono stati presi in considerazione perché non utili al fine della realizzazione di un esecutore simbolico, lo stesso per il package dell'interfaccia grafica.
Il package jaba.instruction offre tutte le classi necessarie a rappresentare simbolicamente il byte code dei programmi java. Per ogni istruzione abbiamo una

classe che la rappresenta, e ogni classe permette di sapere per ogni istruzione i suoi usi, le sue definizioni, i suoi operandi, e le eccezioni che esso può generare.
Tutte le classi offerte vengono create e inizializzate all'interno della classe InstructionLoadingVisitor del package jaba.instruction.visitor. La classe viene a sua volta viene inizializzata tramite il costruttore. Il costruttore necessita il passaggio del bytecode da analizzare, l'informazione sul punto di entrata all'interno del bytecode, richiede il passaggio della CondtantPoolMap e della LocalVariableMap (classi definite all'interno del package jaba.sym).
Il metodo principale di questa classe e' java.util.Vector iterate() che visita il codice ,probabilmente solo di un metodo, dato che il byte code e' diviso per metodi all'interno dei file ".class". Costruisce delle istruzioni visitate una rappresentazione simbolica. Questo metodo restituisce un array di oggetti definiti in jaba.instruction. Ci ritroveremmo perciò dopo la chiamata a iterate a lavorare con un array di "istruzioni" che ci potrebbe fornire informazioni sulla loro natura: definizioni e usi eccezioni generate e operandi.
L'uso di questi due package è molto interessante, purtroppo porta con se il peso dell'inizializzazione della symbol table creata da Program() di jaba.sym. I nostri test hanno infatti verificato che gran parte del costo in memoria e in tempi è causata dalla creazione della tabella simbolica all'interno del metodo load() della classe program.
D'altro canto la symbol table risulta essere molto interessante e parte fondamentale di JABAinfatti, quando JABA analizza un programma, crea un oggetto di tipo jaba.sym.Program che legge il bytecode, costruisce un CFG per il programma, simula il codice, per simulare lo stack degli operandi e identificare correttamente definizioni e usi locali di variabili, e salva tutte le informazioni raccolte in opportune strutture dati (Class objects, Method objects, attributi, etc.). A partire da Program, si può quindi accedere a qualsiasi informazione relativa al programma e all'analisi dello stesso.
Il package jaba.classfile, risulta molto interessante perchè permette di leggere i file ."class" e recuperare le informazioni rilevanti in essi contenuti. Il package è indipendente dagli altri e può essere utilizzato senza dipendenze da altri package.

Concludendo, nella progettazione e realizzazione di un esecutore simbolico per Java che opera su bytecode, sono interessanti in particolare i package: jaba.classfile, jaba.instruction e jaba.instruction.visitor. Mentre jaba.classfile è utilizzabile senza gli altrie non crea quindi grossi inconvenienti, jaba.instruction.visitor necessitano la creazione della symbol table, che a livello di prestazioni risulta molto dispendiosa. D'altra parte sarebbe possibile sfruttare solo il package jaba.instruction, che non richiede l'uso della symbol table di JABA, e

riscrivere un servizio per la visita del byte code con la sua rappresentazione.

Nel seguito sono riportati i driver utilizzati per le interazioni con lo strumento JABA e i risultati dei test di prestazione eseguiti.

Driver JABA

Driver fornito con lo strumento, dato un package di classi da analizzare, crea un oggetto jaba.sym.program e lo carica, andando a leggere tutti i file class e costruendone una rappresentazione simbolica(Symbol Table), da questo oggetto vengono poi costruiti due tipi diversi di grafi: ICFG, ACFG.
Utilizzato per capire le potenzialita` complessive dello strumento.

```
package jaba.main;

import jaba.sym.Program;
import jaba.sym.Class;
import jaba.sym.Method;
import jaba.graph.icfg.ICFG;
import jaba.graph.acfg.ACFG;

/**
 * Main driver class for the java analysis system.
 * @author Jim Jones and Saurabh Sinha -- <i>Created.  April 1, 1999</i>
 */
public class JABA
{
  /**
   * Main driver for the java analysis system.  Reads in a resource file
   * specifying all of the class files to be analyzed, loads those classfiles,
   * parses them, and performs analysis on them.
   * @param argv  Command line arguments.  There should be only one command
   *              line argument specifying the resouce file.
   */
  public static void main( String[] argv )
    {
      /* ensure that there is 1 and only 1 command line argument, if not,
            output usage statement. */
      if (argv.length != 1)
            {
              usage();
              return;
            }
      /* read in the resource file */
      ResourceFile rcFile = new ResourceFile( argv[0] );
      /* create and load program object */
      Program program = new Program();
      program.load( rcFile );
      /* the following code in this method is completely optional and
            presented here for demonstration purposes */
      System.out.println( "Symbol table loaded..." );
```

```
        /* construct the ICFG for program */
      ICFG icfg = (ICFG)program.getAttributeOfType( "jaba.graph.icfg.ICFG" );
      System.out.println( "ICFG constructed..." );
      /* construct ACFG for each method in program */
      Class[] classes = program.getClasses();
      for ( int i = 0; i < classes.length; i++ ) {
            if ( classes[i].isSummarized() ) continue;
         Method[] methods = classes[i].getMethods();
         for ( int j = 0; j < methods.length; j++ ) {
            ACFG acfg = (ACFG)methods[j].getAttributeOfType(
               "jaba.graph.acfg.ACFG" );
         }
      }
      System.out.println( "ACFGs constructed..." );
      /* output a message stating analysis completed successfully */
      System.out.println( "\nAnalysis completed successfully" );
    }
  /**
   * Outputs a usage statement to stderr.
   */
  public static void usage()
    {
      System.err.println( "Error: must provide one command-line argument:\n" +
                          "\tthe filename of a valid resource file." );
    }
}
```

Driver MYJABA

Dato un package di classi si occupa della lettura dei file .class e della successiva costruzione della Symbol table ancora l'oggetto jaba.sym.program.
Utilizzato per capire quanto incide la costruzione dei grafi sull'analisi complessiva.

```
package jaba.main;
import jaba.sym.Program;


public class MYJABA
{
  public static void main( String[] argv )
    {
      /* ensure that there is 1 and only 1 command line argument, if not,
            output usage statement. */
      if (argv.length != 1)
            {
              usage();
              return;
            }
      /* read in the resource file */
      ResourceFile rcFile = new ResourceFile( argv[0] );
      /* create and load program object */
      Program program = new Program();
      program.load( rcFile );
      /* the following code in this method is completely optional and
            presented here for demonstration purposes */
       System.out.println( "Symbol table loaded..." );
        /* output a message stating analysis completed successfully */
       System.out.println( "\nAnalysis completed successfully" );
    }
  /**
   * Outputs a usage statement to stderr.
   */
  public static void usage()
    {
      System.err.println( "Error: must provide one command-line argument:\n" +
                                  "\tthe filename of a valid resource file." );
    }
}
```

Driver MYJABA1

Dato un singolo file ".class" lo legge e ricava alcune informazioni: campi e constant pool. Stampa le informazioni ricavate a video.
Utilizzato per capire le prestazioni di JABA nel solo compito di recupero informazioni. Utilizzato anche per capire la differenza di prestazioni in base alla mole di lavoro fornita in ingresso.

```
package jaba.main;
import jaba.classfile.ClassFile;
import jaba.classfile.ConstantPool;
import jaba.classfile.ConstantPoolInfo;

public class MYJABA1
{
  public static void main( String[] argv )
    {
      /* ensure that there is 1 and only 1 command line argument, if not,
            output usage statement. */
      if (argv.length != 1)
            {
              usage();
              return;
            }

            ClassFile jcr=new ClassFile(argv[0]);
            ConstantPool cp;
            ConstantPoolInfo cpInfo;
            try{jcr.Load();
            }
            catch (Exception e)
            {System.out.println("Non e' un file .class");}

            cp=jcr.getConstantPool();
            cpInfo=cp.getConstantPoolEntry(5);

            System.out.println("Nome classe:");
            System.out.println(jcr.GetClassName());
            System.out.println("i field sono:");
            System.out.println(jcr.getFieldsCount());
            System.out.println("Constant Pool Count");
            System.out.println(cp.getConstantPoolCount());
            System.out.println("Entry 5 Constant Pool:");
    System.out.println(cpInfo.getString());
      /* output a message stating analysis completed successfully */
      System.out.println( "\nAnalysis completed successfully" );
```

```
    }
  /**
   * Outputs a usage statement to stderr.
   */
  public static void usage()
    {
      System.err.println( "Error: must provide one command-line argument:\n" +
                          "\tthe filename of a valid resource file." );
    }
}
```

Driver MYJABA2

Dato in ingresso un package di classi apre e legge tutti i file .class.
Utilizzato per testare le potenzialità di jaba.classfile, e capire la differenza di prestazioni in base alla mole di lavoro in ingresso.

```
package jaba.main;
import jaba.classfile.ClassFile;
import java.io.IOException;

public class MYJABA2
{
  public static void main( String[] argv )
    {
      /* ensure that there is 1 and only 1 command line argument, if not,
            output usage statement. */
      if (argv.length != 1)
            {
              usage();
              return;
            }

      /* read in the resource file */
    ResourceFile rcFile = new ResourceFile( argv[0] );
    String[] arrayClass;
          ClassFile[] arrayClassOpened;

      /* create and load Symbol Table */
                    arrayClass=rcFile.getClassFilePathNames();
                    arrayClassOpened=new ClassFile[arrayClass.length];
                    for(int i=0;i<arrayClass.length;i++)
                    {
                              arrayClassOpened[i]=new ClassFile(arrayClass[i]);
                              try
                              {
                                        arrayClassOpened[i].Load();
                              }catch(IOException e)
                              {System.out.println("Errore di caricamento class file");}
                    }
      System.out.println( "Symbol table loaded..." );
          try{
          System.in.read();
          }catch(IOException e)
          {System.out.println("Errore di IO");}
         /* output a message stating analysis completed successfully */
```

```
      System.out.println( "\nAnalysis completed successfully" );
    }
  /**
   * Outputs a usage statement to stderr.
   */
  public static void usage()
    {
      System.err.println( "Error: must provide one command-line argument:\n" +
                          "\tthe filename of a valid resource file." );
    }
}
```

Driver MYJABA3

Dato in ingresso un package di classi legge tutti i file .class e costruisce la System Table, in questo caso costruendo direttamente l`oggetto jaba.sym.SymbolTable. Utilizzato per capire il vero peso in prestazioni della Symbol Table sull'analisi.

```
package jaba.main;
import jaba.classfile.ClassFile;
import java.io.IOException;
import jaba.sym.SymbolTable;
import jaba.sym.SymbolTableEntry;
import jaba.sym.Program;

public class MYJABA3
{
  public static void main( String[] argv )
    {

      /* ensure that there is 1 and only 1 command line argument, if not,
            output usage statement. */
      if (argv.length != 1)
            {
              usage();
              return;
            }

     /* read in the resource file */
    ResourceFile rcFile = new ResourceFile( argv[0] );
    String[] arrayClass;
          ClassFile[] arrayClassOpened;
          SymbolTable sTable= new SymbolTable();
          Program p= new Program();
          SymbolTableEntry[] sTableEntry;
     /* create and load Symbol Table */
                    arrayClass=rcFile.getClassFilePathNames();
                    arrayClassOpened=new ClassFile[arrayClass.length];
                    for(int i=0;i<arrayClass.length;i++)
                    {
                              arrayClassOpened[i]=new ClassFile(arrayClass[i]);
                              try{
                              arrayClassOpened[i].Load();
                              }catch(IOException e)
                              {System.out.println("ERRORE di IO");}
                    }
          sTable.load(p,arrayClassOpened,rcFile.getClassPaths());
          sTableEntry=sTable.getSymbols();
```

```
            System.out.println("Symbol table");
            System.out.println(sTableEntry[0].toString());
                System.out.println( "Symbol table loaded..." );
            try{
            System.in.read();
            }catch(IOException e)
            {System.out.println("ERRORE IO");}
        /* output a message stating analysis completed successfully */
        System.out.println( "\nAnalysis completed successfully" );
  }
   /**
    * Outputs a usage statement to stderr.
    */
   public static void usage()
     {
       System.err.println( "Error: must provide one command-line argument:\n" +
                                  "\tthe filename of a valid resource file." );
     }
}
```

Prestazioni

AMD Athlon 1,4Ghz
640 MB Ram 133Mhz

	JABA	MYJABA	MYJABA2	MYJABA3	MYJABA1	
Punto	6,40	6,05	0,30	6,03	0,3	Punto
Test	6,45	6,05	0,31	5,99	0,3	Test
Rettangolo	6,52	6,05	0,33	6,04	0,36	Arena
GiavaMachia	14,10	13,25	0,42	13,35	0,35	Robot

AMD Duron 800Mhz
256 MB Ram 133Mhz

	JABA	MYJABA	MYJABA2	MYJABA3	MYJABA1	
Test	11,90	10,90	0,60	11,00	0,6	Test
Punto	11,90	10,90	0,60	11,00	0,6	Punto
Rettangolo	11,80	10,90	0,70	10,90	0,68	Arena
GiavaMachia	26,70	24,30	0,80	24,40	0,6	Robot

Intel Celeron 333Mhz
128 MB RAM

	JABA	MYJABA	MYJABA2	MYJABA3	MYJABA1	
Test	24,80	23,40	1,10	23,40	1	Test
Punto	24,70	23,40	1,10	23,30	1	Punto
Rettangolo	24,70	23,10	1,15	23,30	1,13	Arena
GiavaMachia	65,40	52,00	1,40	61,00	1,13	Robot

Occupazione in memoria.

DRIVER	SORGENTE	N CLASSI	LINEE	DIMENSIONE	RSS	MEM
JABA	punto	1	67	994B	29.3MB	25MB
	test	2	45+x	1.57KB	29.3MB	25MB
	rettangolo	3	206	3.8KB	29.5MB	25MB
	giavamachia	15	1494	30.7KB	57.8MB	54MB
MYJABA	punto	1	67	994B	27.9MB	23MB
	test	2	45+x	1.57KB	28.4MB	24MB
	rettangolo	3	206	3.8KB	28.3MB	24MB
	giavamachia	15	1494	30.7KB	54.6MB	50MB
MYJABA2	punto	1	67	994B	7.2MB	3MB
	test	2	45+x	1.57KB	7.7MB	3MB
	rettangolo	3	206	3.8KB	7.7MB	3MB
	giavamachia	15	1494	30.7KB	8MB	4MB
MYJABA3	punto	1	67	994B	28.4MB	24MB
	test	2	45+x	1.57KB	28.4MB	24MB
	rettangolo	3	206	3.8KB	28.4MB	24MB
	giavamachia	15	1494	30.7KB	55MB	51MB
MYJABA1	punto	1	67	994B	7.2MB	3MB
	Test	1	45	840B	7.3MB	3MB
	Arena	1	228	6.2KB	7.6MB	3MB
	Robot	1	373	7.3KB	7.6MB	4MB

Appendice C
Documentazione fase di progettazione

Di seguito viene presentata tutta la documentazione della fase di progettazione fornita dallo strumento utilizzato: Together® 6.0

SymbolicExecutor

Table Of Contents

Root Package

Diagrams

diagram Symbolic execution

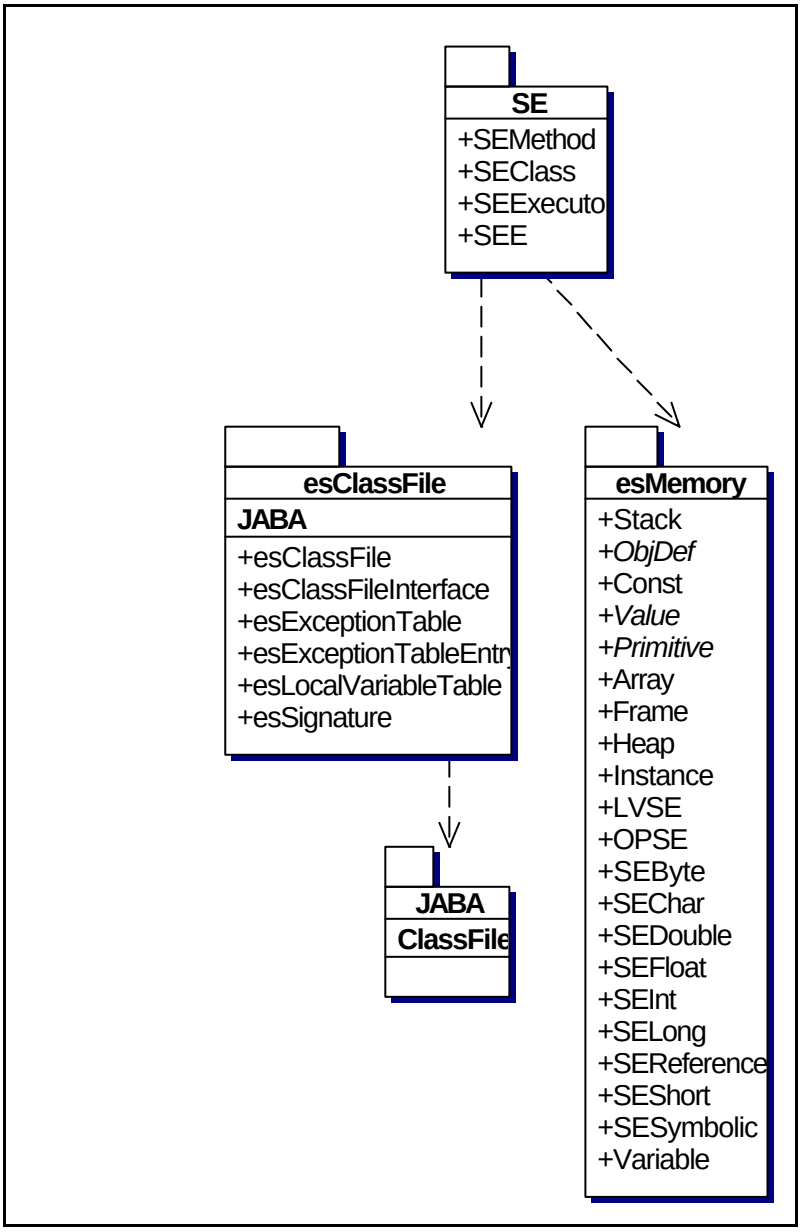
Subpackages

package esClassFile
package esMemory
package JABA
package SE

Class Diagrams

Diagram *Symbolic execution*

package: <default>



Package Nodes

esClassFile
esMemory
JABA
SE

Package Node Detail

Package *esMemory*

Package *esClassFile*

Dependency Links

to **ClassDiagram** JABA

Package *JABA*

Package *SE*

Package dell'esecutore simbolico

Dependency Links

to **ClassDiagram** esClassFile

to **ClassDiagram** esMemory

Package *esClassFile*

Class Diagrams

diagram esClassFile

Classes

class esClassFile
class esClassFileInterface
class esExceptionTable
class esExceptionTableEntry
class esLocalVariableTable
class esSignature

Class Diagrams

Class Diagram *esClassFile*

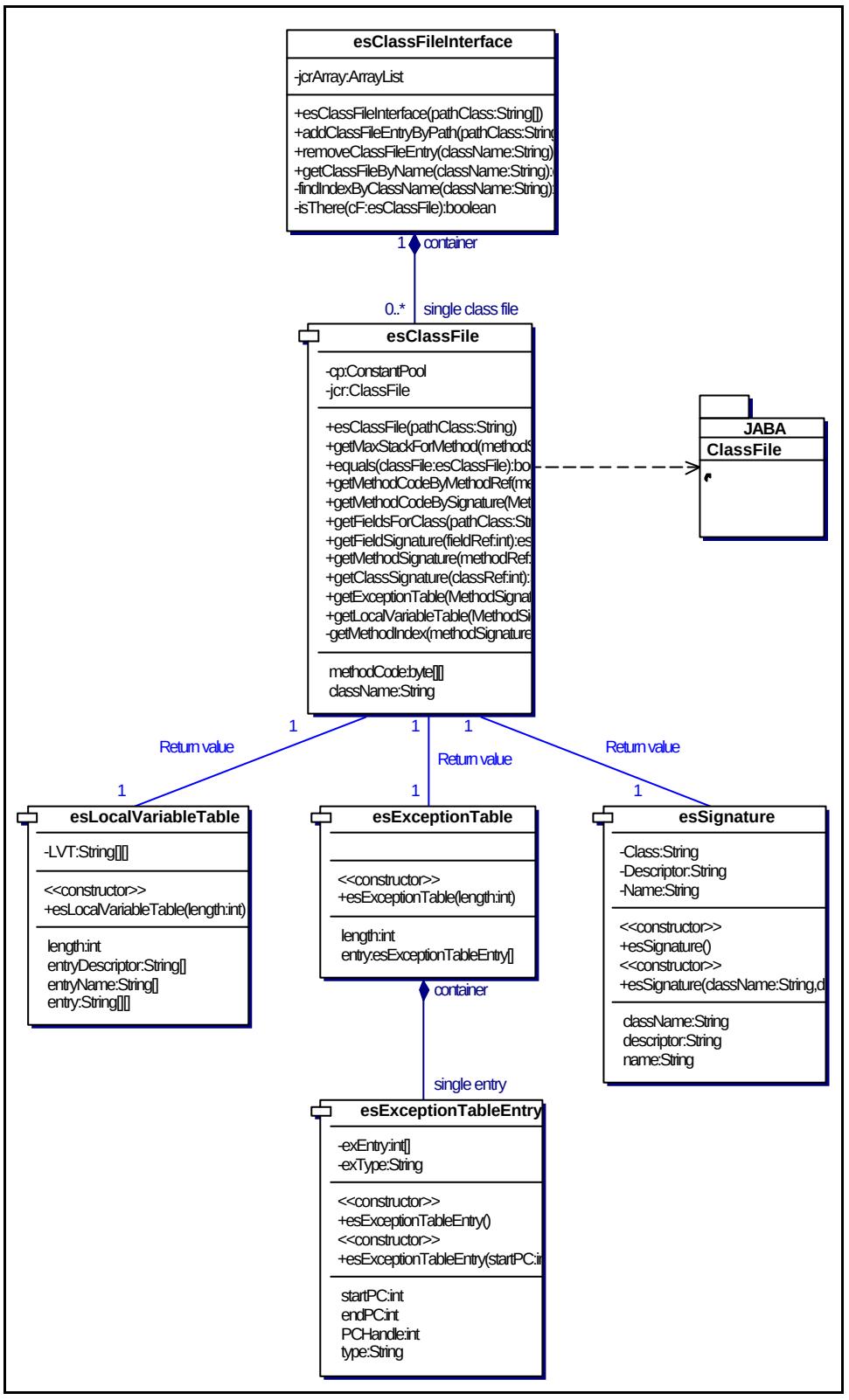
package: esClassFile

Class Nodes

esClassFile
esClassFileInterface
esExceptionTable
esExceptionTableEntry
esLocalVariableTable
esSignature

Shortcuts to Diagrams

ClassDiagram JABA



Class Detail

Class *esClassFile*

package: **esClassFile**

```
public class esClassFile
```

Interface for dialog with classfile analyzer (JABA.classfile) for retrieve information for a single class file

Field Summary	
private ConstantPool	**cp**
private ClassFile	**jcr**
private esLocalVariableTable	**lnkesLocalVariableTable**

Constructor Summary	
public	**esClassFile**(String pathClass) Constructor, initialize the structure to retrieve information from a file .class

Method Summary	
public boolean	**equals**(esClassFile classFile) Test the equality of class file structure, the test is on the name of the class
public String	**getClassName**() Returns the name of the class
public String	**getClassSignature**(int classRef) Given an index of the constant table of CONSTANT_Class type, returns the signature of the class
public esExceptionTable	**getExceptionTable**(esSignature MethodSignature) Given the signature of a method, returns the Exception table of a method.
public esSignature	**getFieldsForClass**(String pathClass) Given a classpath returns all the fields of that class
public esSignature	**getFieldSignature**(int fieldRef) Given an index of the constant pool of CONSTANT_FieldRef type, returns the signature of the Field
public esLocalVariableTable	**getLocalVariableTable**(esSignature MethodSignature) Given the signature of a method, returns the Local Variable table information for that method.
public int	**getMaxStackForMethod**(esSignature methodSignature) Given the signature of a method, the method returns the maximum length needed for the operand stack for that method
private byte[]	**getMethodCode**(int index)
public byte[]	**getMethodCodeByMethodRef**(int methodRef) Given a Methodref index of the constant pool, returns the of bytecode
public byte[]	**getMethodCodeBySignature**(esSignature MethodSignature) Given the signature of a method, returns the byte code of the method.
private int	**getMethodIndex**(esSignature methodSignature)
public esSignature	**getMethodSignature**(int methodRef) Given an index of the constant pool of CONSTANT_Methodref type, returns the signature of the Method

Field Detail

cp

private ConstantPool cp

---

jcr

private ClassFile jcr

---

lnkesLocalVariableTable

private esLocalVariableTable lnkesLocalVariableTable

Constructor Detail

esClassFile

```
public esClassFile(String pathClass)
```

Constructor, initialize the structure to retrieve information from a file .class

Throws:
Exception

Parameter doc:
pathClass The path for the class file, the path is absolute without extension

Method Detail

equals

```
public boolean equals(esClassFile classFile)
```

Test the equality of class file structure, the test is on the name of the class

Parameter doc:
classFile the class needed for the evaluation of the test

---

getClassName

```
public String getClassName()
```

Returns the name of the class

Return doc:
String Is the name of the class.

---

getClassSignature

```
public String getClassSignature(int classRef)
```

Given an index of the constant table of CONSTANT_Class type, returns the signature of the class

Parameter doc:
classRef Is the CONSTANT_Classref of searched class

Return doc:
String Is the string that contains the signature of a Class, for a class the signature is only the name.

---

getExceptionTable

```
public esExceptionTable getExceptionTable(esSignature MethodSignature)
```

Given the signature of a method, returns the Exception table of a method.

Throws:
RuntimeException

Parameter doc:
MethodSignature Is the structure that contains the signature of a method.

Return doc:
esExceptionTable Is a structure that contains the exception table.

Exception doc:
Exception if the method signature is wrong

---

getFieldsForClass

```
public esSignature getFieldsForClass(String pathClass)
```

Given a classpath returns all the fields of that class

Parameter doc:
String containing the absolute path of the class

Return doc:
Array of signature each element for each field of the class

---

getFieldSignature

```
public esSignature getFieldSignature(int fieldRef)
```

Given an index of the constant pool of CONSTANT_FieldRef type, returns the signature of the Field

Parameter doc:
fieldRef Is the CONSTANT_Fieldref of searched field

Return doc:
Signature Is the structure that contains the signature of a field.

---

getLocalVariableTable

```
public esLocalVariableTable getLocalVariableTable(esSignature MethodSignature)
```

Given the signature of a method, returns the Local Variable table information for that method.

Throws:
RuntimeException

Parameter doc:
MethodSignature Is the structure that contains the signature of a method.

Return doc:
esLocalVariableTable Is the structure that contains the Local Variable Table.

Exception doc:
RuntimeException if the method signature is wrong

---

getMaxStackForMethod

```
public int getMaxStackForMethod(esSignature methodSignature)
```

Given the signature of a method, the method returns the maximum length needed for the operand stack for that method

Throws:
RuntimeException

Parameter doc:
methodSignature The signature for the searched method

Return doc:
int The max length of the operand stack

Exception doc:
RuntimeException if the method signature is wrong

---

getMethodCode

```
private byte[] getMethodCode(int index)
```

Throws:
RuntimeException

Exception doc:
RuntimeExecption if the method signature is wrong

---

getMethodCodeByMethodRef

```
public byte[] getMethodCodeByMethodRef(int methodRef)
```

Given a Methodref index of the constant pool, returns the of bytecode

Parameter doc:
methodRef Is the Constant_methodref of searched method

Return doc:
byte[] Array of byte that contains the byte code

---

getMethodCodeBySignature

```
public byte[] getMethodCodeBySignature(esSignature MethodSignature)
```

Given the signature of a method, returns the byte code of the method.

Parameter doc:
MethodSignature Is the structure that contains the signature of a method.

Return doc:
byte[] Array of byte that contains the byte code

---

getMethodIndex

```
private int getMethodIndex(esSignature methodSignature)
```

Throws:
RuntimeException

---

getMethodSignature

```
public esSignature getMethodSignature(int methodRef)
```

Given an index of the constant pool of CONSTANT_Methodref type, returns the signature of the Method

Parameter doc:
methodRef Is the CONSTANT_Methodref of searched field

Return doc:
Signature Is the structure that contains the signature of a method.

Class *esClassFileInterface*

package: **esClassFile**

```
public class esClassFileInterface
```

Class that permitt to retrieve information from a number of classfile, we take assumption that there are't two classfile with same name

Field Summary	
private ArrayList	**jcrArray**
private esClassFile	**lnkesClassFile**

Constructor Summary	
public	**esClassFileInterface**(String pathClass) Constructor, given an array of classpath build the structure to retrieve information from the class files

Method Summary	
public void	**addClassFileEntryByPath**(String pathClass) Given a class file insert it at the end of array.
private int	**findIndexByClassName**(String className)
public esClassFile	**getClassFileByName**(String className) Given a class name returns the correspondent classFile
private boolean	**isThere**(esClassFile cF)
public esClassFile	**removeClassFileEntry**(String className) Given a class file name remove it from the array.

Field Detail

jcrArray

private ArrayList jcrArray

---

lnkesClassFile

private esClassFile lnkesClassFile

Constructor Detail

esClassFileInterface

```
public esClassFileInterface(String pathClass)
```

Constructor, given an array of classpath build the structure to retrieve information from the class files

Throws:
RuntimeException
Exception

Parameter doc:
pathClass array of classpath where to find the file .class to open

Method Detail

addClassFileEntryByPath

```
public void addClassFileEntryByPath(String pathClass)
```

Given a class file insert it at the end of array.

Throws:
RuntimeException
Exception

Parameter doc:
String containing the path of the classFile to insert

---

findIndexByClassName

```
private int findIndexByClassName(String className)
```

Throws:
RuntimeException

---

getClassFileByName

```
public esClassFile getClassFileByName(String className)
```

Given a class name returns the correspondent classFile

Parameter doc:
className The searched class

Return doc:
esClassFile The classFile structure of the correspondent class

---

isThere

```
private boolean isThere(esClassFile cF)
```

---

removeClassFileEntry

```
public esClassFile removeClassFileEntry(String className)
```

Given a class file name remove it from the array.

Throws:
RuntimeException

Parameter doc:
className name of the class file to remove

Class *esExceptionTable*

package: **esClassFile**

```
public class esExceptionTable
```

Class that represents an Exception table

Field Summary	
private esClassFile	**lnkesClassFile**
private esExceptionTableEntry	**lnkesExceptionTableEntry**

Constructor Summary	
public	**esExceptionTable**(int length) Constructor, Initialize the structure that will contain the exception table

Method Summary	
public esExceptionTableEntry	**getEntry**(int index) Given an index of exception table, returns an entry
public int	**getLength**() Returns the number of element of the exception table
public void	**setEntry**(int index, esExceptionTableEntry entry) Set an entry of the exception table.

Field Detail

lnkesClassFile

private esClassFile lnkesClassFile

---

lnkesExceptionTableEntry

private esExceptionTableEntry lnkesExceptionTableEntry

Constructor Detail

esExceptionTable

```
public esExceptionTable(int length)
```

Constructor, Initialize the structure that will contain the exception table

Parameter doc:
length number of element of exception table

Stereotype:
constructor

Method Detail

getEntry

```
public esExceptionTableEntry getEntry(int index)
```

Given an index of exception table, returns an entry

Return doc:
esExceptionTableEntry an entry in the exception table

---

getLength

```
public int getLength()
```

Returns the number of element of the exception table

Return doc:
int the number of element of the exception table

---

setEntry

```
public void setEntry(int index, esExceptionTableEntry entry)
```

Set an entry of the exception table.

Parameter doc:
index The position where insert the entry
entry The entry to insert

Class *esExceptionTableEntry*

package: **esClassFile**

```
public class esExceptionTableEntry
```

Class that represents an entry of the exception table

Field Summary	
private int[]	**exEntry**
private String	**exType**

Constructor Summary	
public	**esExceptionTableEntry**() Constructor, initialize the structure with null value
public	**esExceptionTableEntry**(int startPC, int endPC, int PCHandle, String type) Constructor, Initialize the structure for the Entry

Method Summary	
public int	**getEndPC**() Return the Program Counter of the end of try
public int	**getPCHandle**() Return the Program Counter of the start of catch
public int	**getStartPC**() Return the Program Counter of the start of try
public String	**getType**() Return the type of exception
public void	**setEndPC**(int endPC) Set the Program Counter of the end of try
public void	**setPCHandle**(int pcHandle) Set the Program Counter of the start of catch
public void	**setStartPC**(int startPC) Set the Program Counter of the start of try
public void	**setType**(String type) Set the type of exception

Field Detail

exEntry

private int[] exEntry

---

exType

private String exType

Constructor Detail

esExceptionTableEntry

```
public esExceptionTableEntry()
```

Constructor, initialize the structure with null value

Stereotype:
constructor

---

esExceptionTableEntry

```
public esExceptionTableEntry(int startPC, int endPC, int PCHandle, String type)
```

Constructor, Initialize the structure for the Entry

Parameter doc:
startPC Program counter of the start of try
endPC Program counter of the end of try
PCHandle Program Counter of the start of catch
type Type of the Exception

Stereotype:
constructor

Method Detail

getEndPC

```
public int getEndPC()
```

Return the Program Counter of the end of try

Return doc:
int The Program Counter of the end of try statement

---

getPCHandle

```
public int getPCHandle()
```

Return the Program Counter of the start of catch

Return doc:
int The Program Counter of the start of catch statement

---

getStartPC

```
public int getStartPC()
```

Return the Program Counter of the start of try

Return doc:
int The Program Counter of the start of try statement

---

getType

```
public String getType()
```

Return the type of exception

Return doc:
String The type of exception

---

setEndPC

```
public void setEndPC(int endPC)
```

Set the Program Counter of the end of try

Parameter doc:
endPC The Program Counter of the end of try statement

---

setPCHandle

```
public void setPCHandle(int pcHandle)
```

Set the Program Counter of the start of catch

Parameter doc:
pcHandle The Program Counter of the start of catch statement

---

setStartPC

```
public void setStartPC(int startPC)
```

Set the Program Counter of the start of try

Parameter doc:
startPc The Program Counter of the start of try statement

---

setType

```
public void setType(String type)
```

Set the type of exception

Parameter doc:
String The type of exception

Class *esLocalVariableTable*

package: **esClassFile**

```
public class esLocalVariableTable
```

Class that represents a local variable table

Field Summary	
private String	**LVT**

Constructor Summary	
public	**esLocalVariableTable**(int length) Constructor, initialize the object

Method Summary	
public String	**getEntry**(int index) Given an index of LVT, returns an entry
public int	**getLength**() Return the length of the local variable table
public void	**setEntry**(int index, String entry) Set a new entry in the LVT at indicated position.
public void	**setEntryDescriptor**(int index, String descriptor) Set an entry of the LVT with a new descriptor
public void	**setEntryName**(int index, String name) Set an entry of the LVT with a new name

Field Detail

LVT

private String LVT

Constructor Detail

esLocalVariableTable

```
public esLocalVariableTable(int length)
```

Constructor, initialize the object

Parameter doc:
length The length for the local table

Stereotype:
constructor

Method Detail

getEntry

```
public String getEntry(int index)
```

Given an index of LVT, returns an entry

Parameter doc:
index The index of the searched variable

Return doc:
String[] An entry of the local variable table, is composed of two part, the descriptor(String[0]) and the name(String[1])

---

getLength

```
public int getLength()
```

Return the length of the local variable table

Return doc:
int length

---

setEntry

```
public void setEntry(int index, String entry)
```

Set a new entry in the LVT at indicated position.

Parameter doc:
index position for the entry
String[] An entry of the local variable table, is composed of two part, the descriptor(String[0]) and the name(String[1])

---

setEntryDescriptor

```
public void setEntryDescriptor(int index, String descriptor)
```

Set an entry of the LVT with a new descriptor

Parameter doc:
index position of the variable, which will be changed, in the LVT
descriptor new descriptor for the variable

---

setEntryName

```
public void setEntryName(int index, String name)
```

Set an entry of the LVT with a new name

Parameter doc:
index position of the variable, which will be changed, in the LVT
name new name for the variable

Class *esSignature*

package: **esClassFile**

```
public class esSignature
```

Class that represents the signature of method or a field

Field Summary	
private String	**Class**
private String	**Descriptor**
private esClassFile	**lnkesClassFile**
private String	**Name**

Constructor Summary	
public	**esSignature**() Constructor, Create an instance of the class that contains all null string
public	**esSignature**(String className, String descriptor, String name) Constructor, Given the class name , the descriptor and the name create an instance of the class

Method Summary	
public String	**getClassName**() Return the name of the class
public String	**getDescriptor**() Return the descriptor
public String	**getName**() Return the name
Public void	**setClassName**(String className) Set the Class name
Public void	**setDescriptor**(String descriptor) Set the descriptor
Public void	**setName**(String name) Set the name

Field Detail

Class

private String Class

Descriptor

private String Descriptor

InkesClassFile

private esClassFile InkesClassFile

Name

private String Name

Constructor Detail

esSignature

public esSignature()

Constructor, Create an instance of the class that contains all null string

Stereotype:
constructor

---

esSignature

public esSignature(String className, String descriptor, String name)

Constructor, Given the class name , the descriptor and the name create an instance of the class

Parameter doc:
className The name of the class of the method or field
descriptor The descriptor of method or field
name The name of the method or field

Stereotype:
constructor

Method Detail

getClassName

public String getClassName()

Return the name of the class

Return doc:
String the name of the class

---

getDescriptor

```
public String getDescriptor()
```

Return the descriptor

Return doc:
String The descriptor returned

---

getName

```
public String getName()
```

Return the name

Return doc:
String The name returned

---

setClassName

```
public void setClassName(String className)
```

Set the Class name

Parameter doc:
className the name of class to give

---

setDescriptor

```
public void setDescriptor(String descriptor)
```

Set the descriptor

Parameter doc:
descriptor the descriptor to give

---

setName

```
public void setName(String name)
```

Set the name

Parameter doc:
name The name to give

Package *esMemory*

Class Diagrams

diagram esMemory
diagram Spezzato1
diagram Spezzato2
diagram Spezzato3

Classes

class Array
class Const
class Frame
class Heap
class Instance
class LVSE
class OPSE
class Primitive
class SEByte
class SEChar
class SEDouble
class SEFloat
class SEInt
class SELong
class SEReference
class SEShort
class SESymbolic
class Stack
class Value
class Variable

Interfaces

interface ObjDef

Class Diagrams

Class Diagram *esMemory*

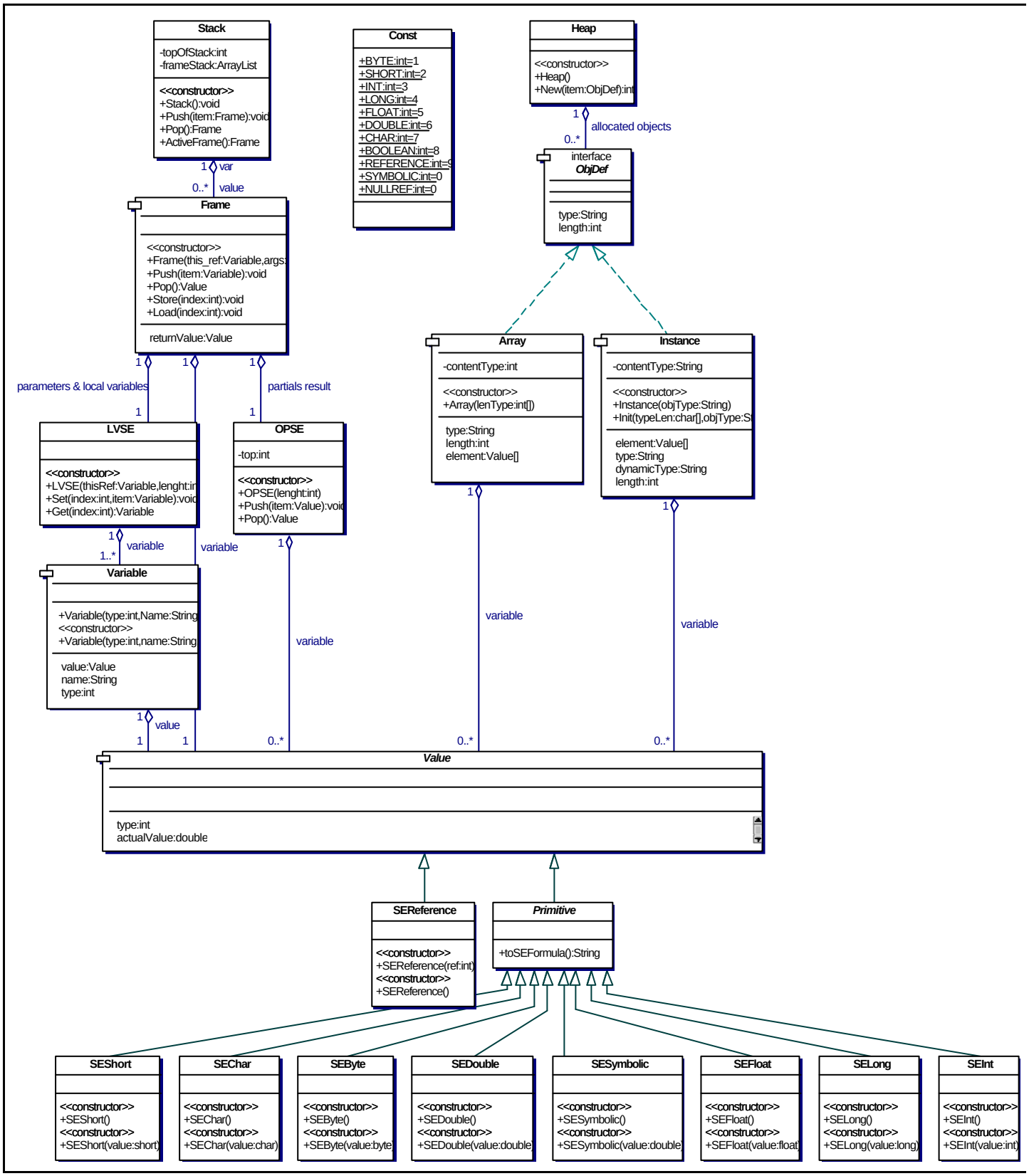
package: esMemory

Class Nodes

Array
Const
Frame
Heap
Instance
LVSE
OPSE
Primitive
SEByte
SEChar
SEDouble
SEFloat
SEInt
SELong
SEReference
SEShort
SESymbolic
Stack,Value,Variable

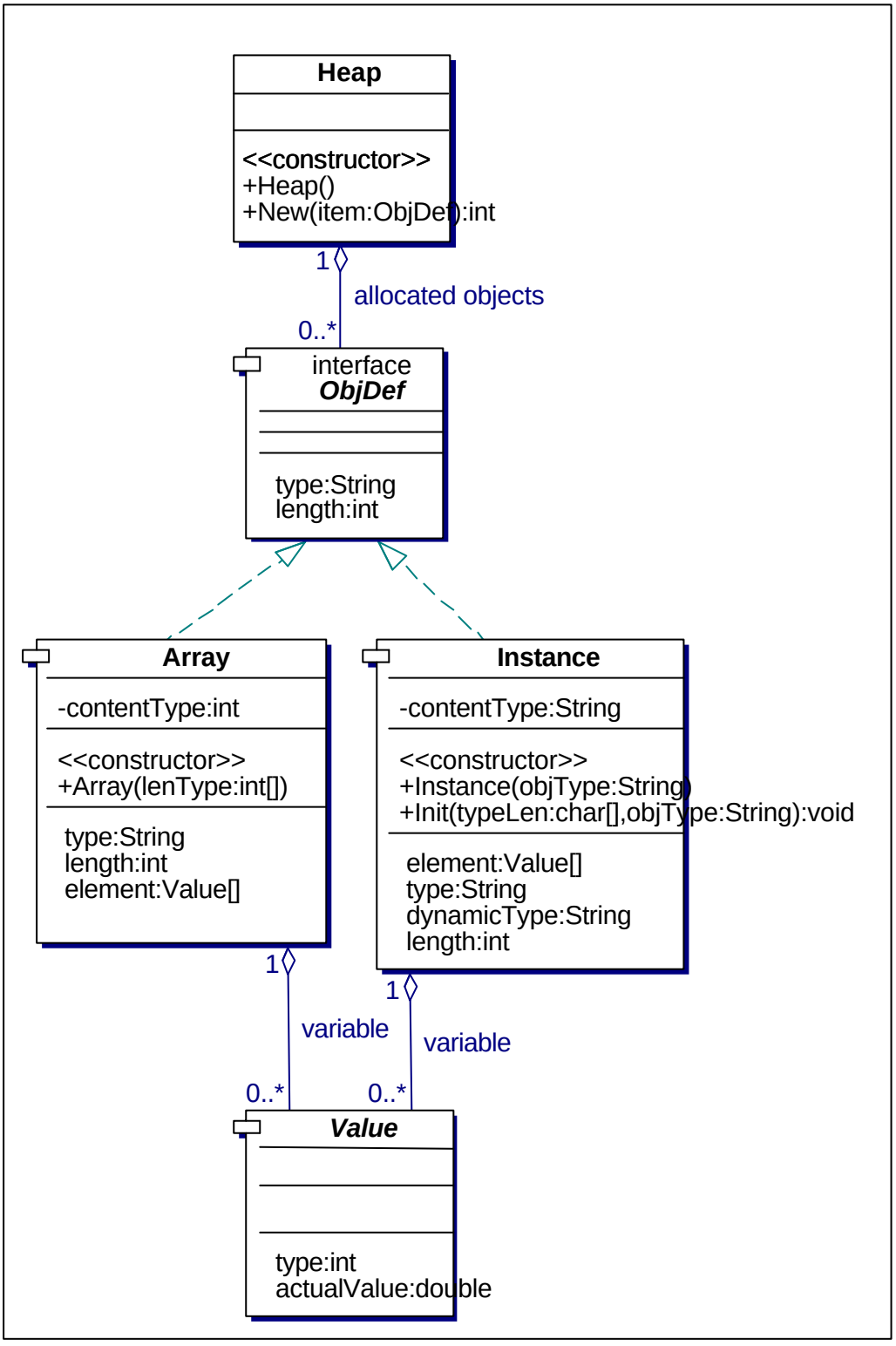
Interface Nodes

ObjDef



Class Diagram *Spezzato1*

package: esMemory



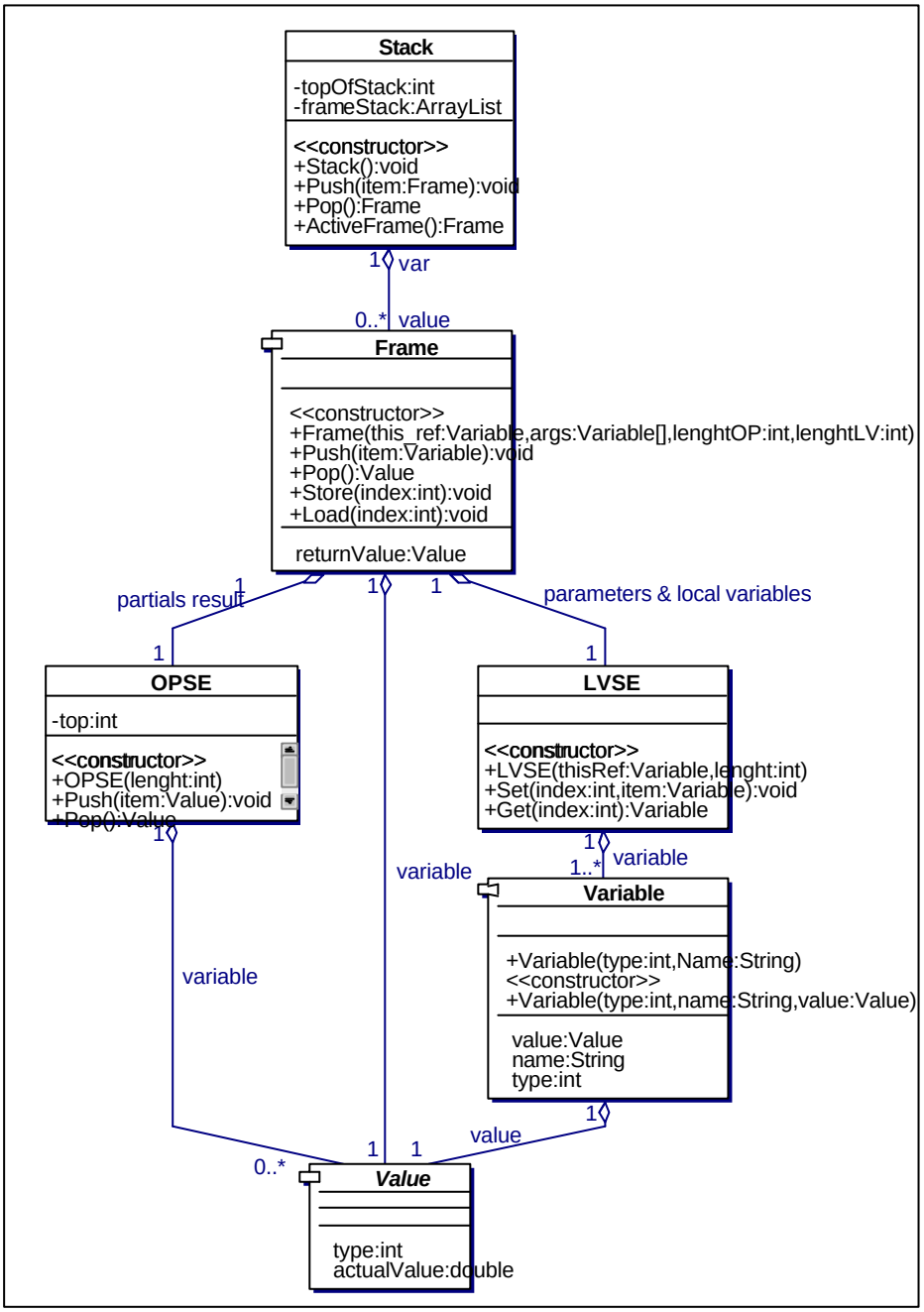
Shortcuts to Elements

of ***ClassDiagram*** *esMemory*

- **Class** Memory.Array
- **Class** Memory.Heap
- **Class** Memory.Instance
- **Interface** Memory.ObjDef
- **Class** Memory.Value

Class Diagram *Spezzato2*

package: esMemory



Shortcuts to Elements

of ***ClassDiagram*** *esMemory*

- **Class** Memory.Frame
- **Class** Memory.LVSE
- **Class** Memory.OPSE
- **Class** Memory.Stack
- **Class** Memory.Value
- **Class** Memory.Variable

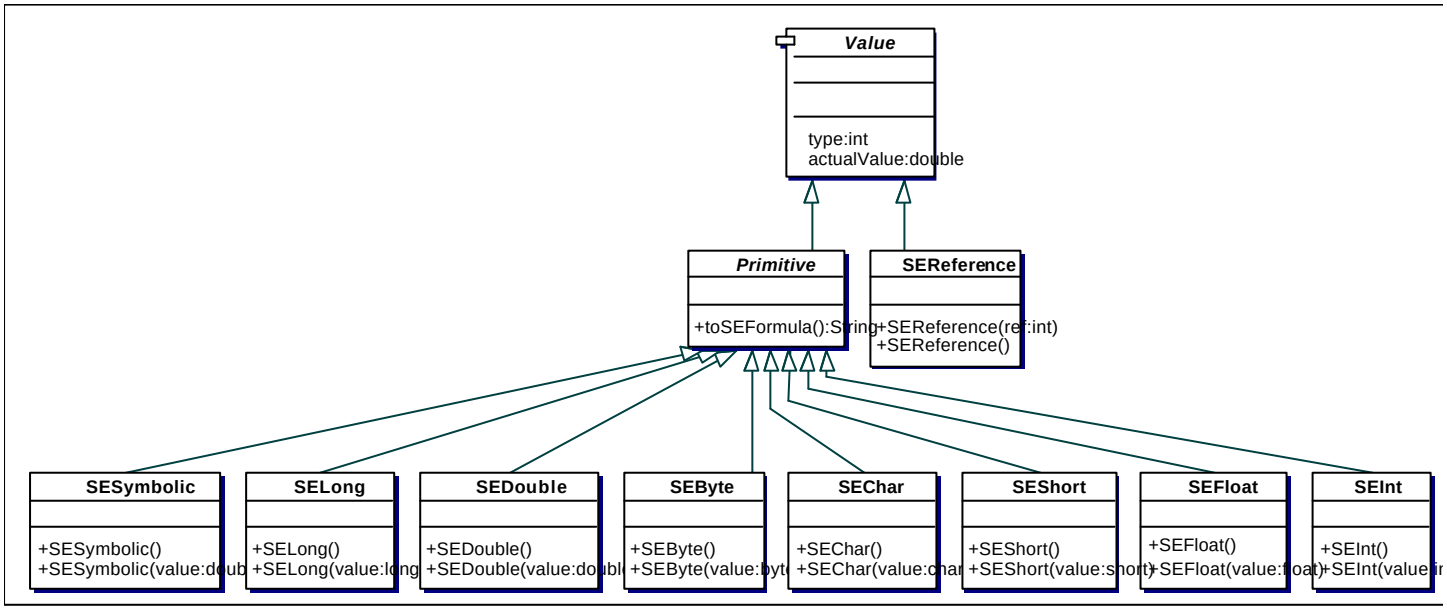
Class Diagram *Spezzato3*

package: esMemory

Shortcuts to Elements

of ***ClassDiagram*** *esMemory*

- **Class** Memory.Primitive
- **Class** Memory.SEByte
- **Class** Memory.SEChar
- **Class** Memory.SEDouble
- **Class** Memory.SEFloat
- **Class** Memory.SEInt
- **Class** Memory.SELong
- **Class** Memory.SEReference
- **Class** Memory.SEShort
- **Class** Memory.SESymbolic
- **Class** Memory.Value



Class Detail

Class *Array*

package: **esMemory**

```
public class Array
```

Implements:

Memory.ObjDef

Class that represent an array in memory.

Field Summary	
private int	**contentType**
private int	**length**

Field Summary	
private Value	**lnkValue**
private String	**type**

Constructor Summary	
public	**Array**(int[] lenType) Creates a new array.

Method Summary	
public Value	**getElement**(int index) Return an element of array
public int	**getLength**() return the number of element of array
public String	**getType**() Return the type of the object
public void	**setElement**(int index, Value item) Modify an element of array

Field Detail

contentType

private int contentType

---

length

private int length

---

lnkValue

private Value lnkValue

---

type

private String type

Constructor Detail

Array

public Array(int[] lenType)

Creates a new array.

Stereotype:
constructor

Parameter doc:
array of integer where the first element is the dimension of the new array and the second one is the type of each element

Method Detail

getElement

public Value getElement(int index)

Return an element of array

Parameter doc:
index: Position of the searched element.

Return doc:
Value: Returned element.

---

getLength

```
public int getLength()
```

return the number of element of array

Return doc:
Length of the array

---

getType

```
public String getType()
```

Return the type of the object

Return doc:
Type of the array

---

setElement

```
public void setElement(int index, Value item)
```

Modify an element of array

Parameter doc:
index: Position of element to modify.
item: New value of the element.

Class *Const*

package: **esMemory**

```
public final class Const
```

Class that contain costants

Field Summary	
public final static int	**BOOLEAN**
public final static int	**BYTE**
public final static int	**CHAR**
public final static int	**DOUBLE**
public final static int	**FLOAT**
public final static int	**INT**
public final static int	**LONG**
public final static int	**NULLREF**
public final static int	**REFERENCE**
public final static int	**SHORT**
public final static int	**SYMBOLIC**

Field Detail

BOOLEAN

public final static int BOOLEAN = 8

BYTE

public final static int BYTE = 1

CHAR

public final static int CHAR = 7

DOUBLE

public final static int DOUBLE = 6

FLOAT

public final static int FLOAT = 5

INT

public final static int INT = 3

LONG

public final static int LONG = 4

NULLREF

public final static int NULLREF = 0

REFERENCE

public final static int REFERENCE = 9

SHORT

public final static int SHORT = 2

SYMBOLIC

public final static int SYMBOLIC = 0

Class *Frame*

package: **esMemory**

```
public class Frame
```

Class that rapresent activation record of a single method.

Field Summary	
private LVSE	**lnkLVSE**
private OPSE	**lnkOPSE**
private SEE	**lnkSEE**
private Value	**lnkValue**

Constructor Summary	
public	**Frame**(Variable this_ref, Variable args, int lenghtOP, int lenghtLV) Frame constructor

Method Summary	
public Value	**getReturnValue**() Return the return value, the return value is in the frame at the end of the execution of a method.
public void	**Load**(int index) Move a variable from local variable to operand stack.

Method Summary	
public Value	**Pop**() Return and delete the variable from the top of the operand stack.
public void	**Push**(Variable item) Put the passed variable on the top of the operand stack.
public void	**setReturnValue**(Value returnValue)
public void	**Store**(int index) Pop a variable from operand stack and stores it into the local variable.

Field Detail

lnkLVSE

private LVSE lnkLVSE

---

lnkOPSE

private OPSE lnkOPSE

---

lnkSEE

private SEE lnkSEE

---

lnkValue

private Value lnkValue

Constructor Detail

Frame

```
public Frame(Variable this_ref, Variable args, int lenghtOP, int lenghtLV)
```

Frame constructor

Parameter doc:
This: Reference to the owner of the frame
Args: List of Variable type object that represent the parameters to the method.
OP_Lenght: Integer that define the dimension of the operand stack.
LV_Lenght: Integer that define the dimension of the local variable array.

Stereotype:
constructor

Method Detail

getReturnValue

```
public Value getReturnValue()
```

Return the return value, the return value is in the frame at the end of the execution of a method.

Return doc:
Return value of the method.

Load

```
public void Load(int index)
```

Move a variable from local variable to operand stack.

Parameter doc:
Index: Index of the local variable table where get the Variable to store in the operand stack.

Pop

public Value Pop()

Return and delete the variable from the top of the operand stack.

Return doc:
Variable: represent the data on the top of operand stack.

Push

public void Push(Variable item)

Put the passed variable on the top of the operand stack.

Parameter doc:
item: Variable to put on the top of operand stack.

---

setReturnValue

public void setReturnValue(Value returnValue)

Store

public void Store(int index)

Pop a variable from operand stack and stores it into the local variable.

Parameter doc:
Index: Index of local variable table where the method stores the Variable which is popped from the operand stack.

Class *Heap*

package: **esMemory**

public final class Heap

Class that offers the same services of the heap in the JVM's memory.

Field Summary	
private ObjDef	**lnkObjDef**
private SEE	**lnkSEE**

Constructor Summary	
public	**Heap**() Constructor of an Heap structure.

Method Summary	
public int	**New**(ObjDef item) Allocate new space to store the new object into the heap.

Field Detail

lnkObjDef

private ObjDef lnkObjDef

---

lnkSEE

private SEE lnkSEE

Constructor Detail

Heap

```
public Heap()
```

Constructor of an Heap structure.

Stereotype:
constructor

Method Detail

New

```
public int New(ObjDef item)
```

Allocate new space to store the new object into the heap.

Parameter doc:
item: Object to create in the heap.

Return doc:
Handle of the object allocated

Class *Instance*

package: **esMemory**

```
public class Instance
```

Implements:
Memory.ObjDef

Class that represent an instance of an object in memory.

It contain a reference to the Heap

Field Summary	
private String	**contentType**
private int	**length**
private Value	**lnkValue**
private String	**type**

Constructor Summary	
public	**Instance**(String objType) Constructor for an instance of an object.Initialize only the static type of the object.

Method Summary	
public String	**getDynamicType**() Return the dynamic type of the object.
public Value	**getElement**(int index) Get a field of object.
public int	**getLength**() Return the number of field.
public String	**getType**() Return the static type of the object.
public void	**Init**(char[] typeLen, String objType) Initialize the object with dynamic type and field information.
public void	**setElement**(int index, Value item) Set a field of object.

Field Detail

contentType

private String contentType

---

length

private int length

---

lnkValue

private Value lnkValue

---

type

private String type

Constructor Detail

Instance

```
public Instance(String objType)
```

Constructor for an instance of an object.Initialize only the static type of the object.

Parameter doc:
objType: Static type of object.

Stereotype:
constructor

Method Detail

getDynamicType

```
public String getDynamicType()
```

Return the dynamic type of the object.

---

getElement

```
public Value getElement(int index)
```

Get a field of object.

Parameter doc:
index: Position of element to get from the array of field.

---

getLength

```
public int getLength()
```

Return the number of field.

---

getType

```
public String getType()
```

Return the static type of the object.

Init

```
public void Init(char[] typeLen, String objType)
```

Initialize the object with dynamic type and field information.

Parameter doc:
typeLen: Types and number of content fields, the number is the length of the array.
objType: Dynamic type of object.

---

setElement

```
public void setElement(int index, Value item)
```

Set a field of object.

Parameter doc:
index: Position of element to set in the array of field.
item: New value to give to field.

Class *LVSE*

package: **esMemory**

```
public class LVSE
```

Class that rapresent the local variable table of a method.

Field Summary	
private Variable	**lnkVariable**

Constructor Summary	
public	**LVSE**(Variable thisRef, int lenght) Constructor of the local variable table array.

Method Summary	
public Variable	**Get**(int index) Return the value of an element of the local variable table.
public void	**Set**(int index, Variable item) Store a value into a specific index of the local variable table array.

Field Detail

lnkVariable

private Variable lnkVariable

Constructor Detail

LVSE

```
public LVSE(Variable thisRef, int lenght)
```

Constructor of the local variable table array.

Parameter doc:
thisRef: Parameter of Variable type
it's the reference of the owner of the frame: the instance of object which invokes the method.

Stereotype:
constructor

Method Detail

Get

```
public Variable Get(int index)
```

Return the value of an element of the local variable table.

Return doc:
Object of type Variable at index position.

Parameter doc:
index: Index in the local variable table.

Set

```
public void Set(int index, Variable item)
```

Store a value into a specific index of the local variable table array.

Parameter doc:
Index : Index of the local variable table where the variable is stored.

Class *OPSE*

package: **esMemory**

```
public class OPSE
```

Class that represent the JVM's operand stack.

Field Summary	
private Value	**lnkValue**
private int	**top**

Constructor Summary	
public	**OPSE**(int lenght) Constructor of operand stack.

Method Summary	
public Value	**Pop**() Return and delete the element from the top of the operand stack.
public void	**Push**(Value item) Put the argument on the top of the operand stack.

Field Detail

lnkValue

private Value lnkValue

---

top

private int top

Constructor Detail

OPSE

```
public OPSE(int lenght)
```

Constructor of operand stack.

Parameter doc:
lenght: Size of the operand stack to allocate.

Stereotype:
constructor

Method Detail

Pop

public `Value` `Pop()`

Return and delete the element from the top of the operand stack.

Return doc:
Value: value on the top of the operand stack.

Push

public `void` `Push(Value item)`

Put the argument on the top of the operand stack.

Parameter doc:
item : Value to put on the top of operand stack.

Class *Primitive*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
```

`public abstract class Primitive`

Extends:
Memory.Value

Class that represent a primitive type.

Method Summary	
public `String`	**`toSEFormula`**`()` Method that return a string wich represent a symbolic formula

Method Detail

toSEFormula

public `String` `toSEFormula()`

Method that return a string wich represent a symbolic formula

Return doc:
String: Symbolic formula

Class *SEByte*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SEByte
```

`public class SEByte`

Extends:
Memory.Primitive

Constructor Summary	
public	**SEByte**() Constructor that initialize the value to 0
public	**SEByte**(byte value) Constructor that initialize the value to a passed one

Constructor Detail

SEByte

```
public SEByte()
```

Constructor that initialize the value to 0

Stereotype:
constructor

SEByte

```
public SEByte(byte value)
```

Constructor that initialize the value to a passed one

Parameter doc:
value: passed value

Stereotype:
constructor

Class *SEChar*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SEChar
```

```
public class SEChar
```

Extends:
Memory.Primitive

class that represent a character type

Constructor Summary	
public	**SEChar**() Constructor that initialize the value of the c haracter to 0.
public	**SEChar**(char value) Constructor that initialize the value of the character to a passed value.

Constructor Detail

SEChar

```
public SEChar()
```

Constructor that initialize the value of the character to 0.

Stereotype:
constructor

SEChar

```
public SEChar(char value)
```

Constructor that initialize the value of the character to a passed value.

Parameter doc:
value: Initial value to give to the character

Stereotype:
constructor

Class *SEDouble*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SEDouble
```

```
public class SEDouble
```

Extends:
Memory.Primitive

Constructor Summary	
public	**SEDouble**() Constructor that initialize the value to 0
public	**SEDouble**(double value) Constructor that initialize the value to a passed one

Constructor Detail

SEDouble

```
public SEDouble()
```

Constructor that initialize the value to 0

Stereotype:
constructor

SEDouble

```
public SEDouble(double value)
```

Constructor that initialize the value to a passed one

Parameter doc:
value: passed value

Stereotype:
constructor

Class *SEFloat*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SEFloat
```

```
public class SEFloat
```

Extends:
Memory.Primitive

Constructor Summary	
public	**SEFloat**() Constructor for float type that initialize the value to 0
public	**SEFloat**(float value) Constructor that initialize the value to a passed one

Constructor Detail

SEFloat

public SEFloat()

Constructor for float type that initialize the value to 0

Stereotype:
constructor

SEFloat

public SEFloat(float value)

Constructor that initialize the value to a passed one

Parameter doc:
value: passed value

Stereotype:
constructor

Class *SEInt*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SEInt
```

public class SEInt

Extends:
Memory.Primitive

class that represent an integer Type.

Constructor Summary	
public	**SEInt**() Constructor that initialize the value of the integer to 0.
public	**SEInt**(int value) Constructor that initialize the value of the integer to a passed value.

Constructor Detail

SEInt

public SEInt()

Constructor that initialize the value of the integer to 0.

Stereotype:
constructor

SEInt

```
public SEInt(int value)
```

Constructor that initialize the value of the integer to a passed value.

Parameter doc:
value: Initial value to give to the integer

Stereotype:
constructor

Class *SELong*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SELong
```

```
public class SELong
```

Extends:
Memory.Primitive

Constructor Summary	
public	**SELong**() Constructor of a long type that initialize the value to 0
public	**SELong**(long value) Constructor that initialize the value to a passed one

Constructor Detail

SELong

```
public SELong()
```

Constructor of a long type that initialize the value to 0

Stereotype:
constructor

SELong

```
public SELong(long value)
```

Constructor that initialize the value to a passed one

Parameter doc:
value: passed value

Stereotype:
Constructor

Class *SEReference*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.SEReference
```

```
public class SEReference
```

Extends:

Memory.Value

Class that represent a reference to the heap.

Constructor Summary	
public	**SEReference**() Constructor that initialize the value of the reference to Null=0.
public	**SEReference**(int ref) Constructor of the class.

Constructor Detail

SEReference

```
public SEReference()
```

Constructor that initialize the value of the reference to Null=0.

Stereotype:
constructor

SEReference

```
public SEReference(int ref)
```

Constructor of the class.

Parameter doc:
ref: initial value of the reference.

Stereotype:
constructor

Class *SEShort*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SEShort
```

```
public class SEShort
```

Extends:

Memory.Primitive

Constructor Summary	
public	**SEShort**() Constructor of a short type that initialize the value to 0
public	**SEShort**(short value) Constructor that initialize the value to a passed one

Constructor Detail

SEShort

```
public SEShort()
```

Constructor of a short type that initialize the value to 0

Stereotype:
constructor

SEShort

```
public SEShort(short value)
```

Constructor that initialize the value to a passed one

Parameter doc:
value: passed value

Stereotype:
constructor

Class *SESymbolic*

package: **esMemory**

```
Memory.Value
  |
  +--Memory.Primitive
       |
       +--Memory.SESymbolic
```

```
public class SESymbolic
```

Extends:
Memory.Primitive

Class that represent a symbolic expression.

Constructor Summary	
public	**SESymbolic**() Constructor that initialize the value of the Symbolic expression to Null.
public	**SESymbolic**(double value) Constructor that initialize the value of the symbolic expression to a passed value.

Constructor Detail

SESymbolic

```
public SESymbolic()
```

Constructor that initialize the value of the Symbolic expression to Null.

Stereotype:
constructor

SESymbolic

```
public SESymbolic(double value)
```

Constructor that initialize the value of the symbolic expression to a passed value.

Parameter doc:
value: Initial value to give to the symbolic expression

Stereotype:
constructor

Class *Stack*

package: **esMemory**

```
public class Stack
```

Class that rapresents the thread stack, which contains activation's record of methods.

Classe che rappresenta il thread stack che contiene i record di attivazione dei metodi.

Field Summary	
private ArrayList	**frameStack**
private Frame	**lnkFrame**
private int	**topOfStack**

Method Summary	
public Frame	**ActiveFrame**() Return the copy of the active frame, which is on the top of the stack , the method don't deallocate it.
public Frame	**Pop**() Return the frame on the top of the stack and delete it.
public void	**Push**(Frame item) Put a frame on the top of the stack.
public void	**Stack**() Constructor of stack.

Field Detail

frameStack

private ArrayList frameStack

---

lnkFrame

private Frame lnkFrame

---

topOfStack

private int topOfStack

Method Detail

ActiveFrame

public Frame ActiveFrame()

Return the copy of the active frame, which is on the top of the stack , the method don't deallocate it.

Return doc:
Frame: Copy of the active frame.

Pop

public Frame Pop()

Return the frame on the top of the stack and delete it.

Return doc:
Frame: Frame deallocated from the stack.

Push

```
public void Push(Frame item)
```

Put a frame on the top of the stack.

Parameter doc:
item: Frame that must be put on the top of the stack, it is the frame that it will become the active frame, it must be initialized before being passed.

Stack

```
public void Stack()
```

Constructor of stack.

Stereotype:
constructor

Class *Value*

package: **esMemory**

```
public abstract class Value
```

Class that represent data of reference type, primitive type and simbolic type in memory.

Method Summary	
public double	**getActualValue**() return the value
public int	**getType**() Return the type of the Value
public void	**setActualValue**(double actualValue) Set the value of the Value
public void	**setType**(int type) Set the type of the value, this is the dynamic type

Method Detail

getActualValue

```
public double getActualValue()
```

return the value

---

getType

```
public int getType()
```

Return the type of the Value

---

setActualValue

```
public void setActualValue(double actualValue)
```

Set the value of the Value

---

setType

```
public void setType(int type)
```

Set the type of the value, this is the dynamic type

Class *Variable*

package: **esMemory**

```
public class Variable
```

Abstract class that represent a variable into the memory. Collect name and type of the variable.

Field Summary	
private Value	**lnkValue**
private String	**name** Name of the variable.
private int	**type** Type of the variable.

Constructor Summary	
public	**Variable**(int type, String Name) Constructor of class Variable
public	**Variable**(int type, String name, Value value)

Method Summary	
public String	**getName**()
public int	**getType**()
public Value	**getValue**()
public void	**setValue**(Value value)

Field Detail

lnkValue

private Value lnkValue

---

name

private String name

Name of the variable.

---

type

private int type

Type of the variable.

Constructor Detail

Variable

```
public Variable(int type, String Name)
```

Constructor of class Variable

Parameter doc:
type: type of variable
name: name of variable

Variable

```
public Variable(int type, String name, Value value)
```

Stereotype:
constructor

Parameter doc:
Type of the variable
Name of the variable
initial value of the variable

Method Detail

getName

```
public String getName()
```

Return doc:
Name of the variable

---

getType

```
public int getType()
```

Return doc:
Type of the variable

---

getValue

```
public Value getValue()
```

Return doc:
Value of the variable

---

setValue

```
public void setValue(Value value)
```

Return doc:
Value of the variable

Interface Detail

Interface *ObjDef*

package: **esMemory**

All Known Implementing Classes:
Array, Instance

```
public interface ObjDef
```

Interface for instance class and and array class.

Method Summary	
public int	**getLength**() Return the Number of elements that the object holds.
public String	**getType**() Return the Static Type identifier.

Method Detail

getLength

```
public int getLength()
```

Return the Number of elements that the object holds.

---

getType

```
public String getType()
```

Return the Static Type identifier.

Package *JABA*

Class Diagrams

diagram JABA

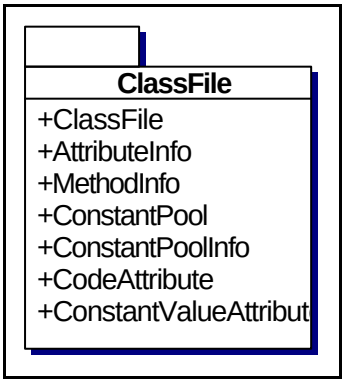
Subpackages

package JABA.ClassFile

Class Diagrams

Class Diagram *JABA*

package: JABA



Package Nodes

ClassFile

Package Node Detail

Package *JABA.ClassFile*

Package *JABA.ClassFile*

Class Diagrams

diagram ClassFile

Classes

class AttributeInfo
class ClassFile
class CodeAttribute
class ConstantPool
class ConstantPoolInfo
class ConstantValueAttribute
class MethodInfo

Package *SE*

Package dell'esecutore simbolico

Class Diagrams

diagram Relationship1
diagram Relationship2
diagram SE

Interaction Diagrams

diagram Sequence

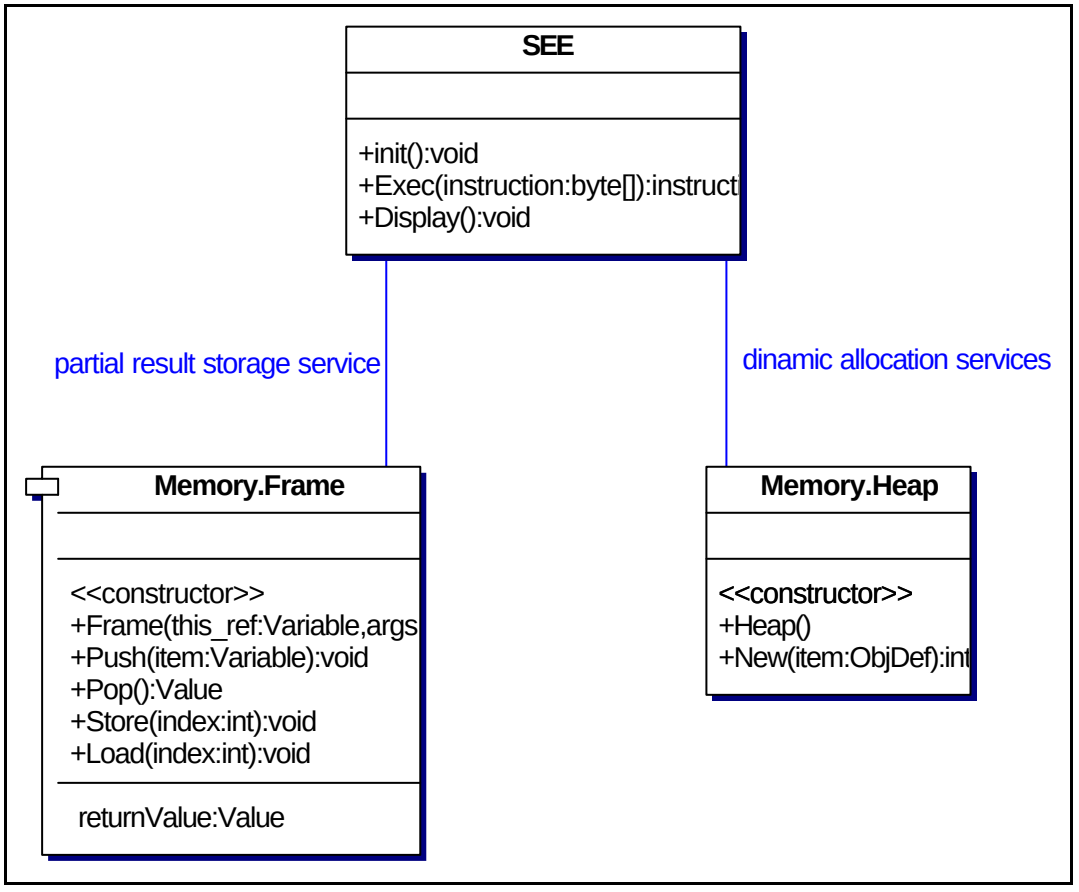
Classes

class SEClass
class SEE
class SEExecutor
class SEMethod

Class Diagrams

Class Diagram *Relationship1*

package: SE



Shortcuts to Elements

of ***ClassDiagram*** *esMemory*

Class Memory.Frame

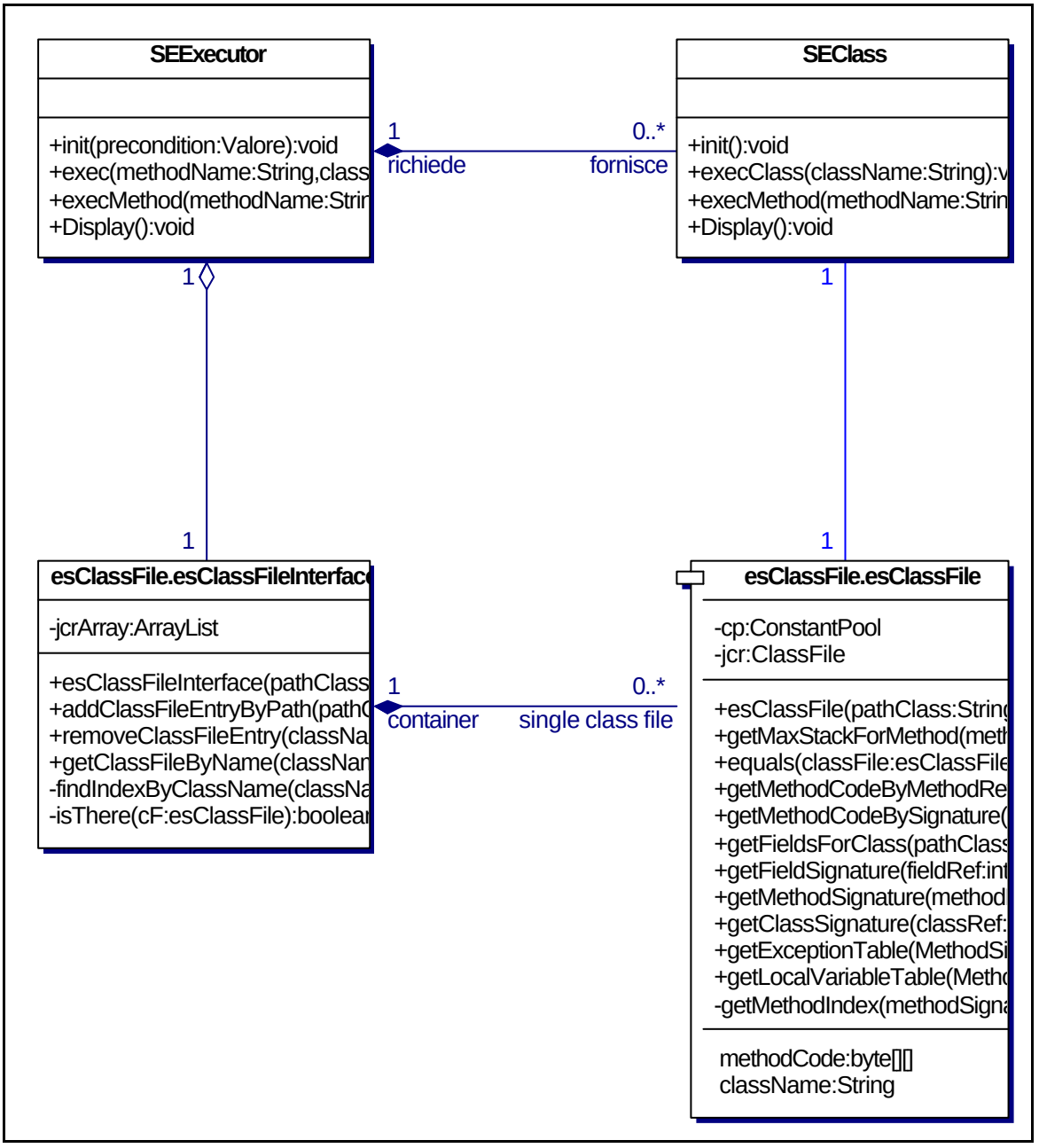
Class Memory.Heap

of ***ClassDiagram*** *SE*

Class SE.SEE

Class Diagram *Relationship2*

package: SE



Shortcuts to Elements

of ***ClassDiagram*** *esClassFile*

Class esClassFile.esClassFile

Class esClassFile.esClassFileInterface

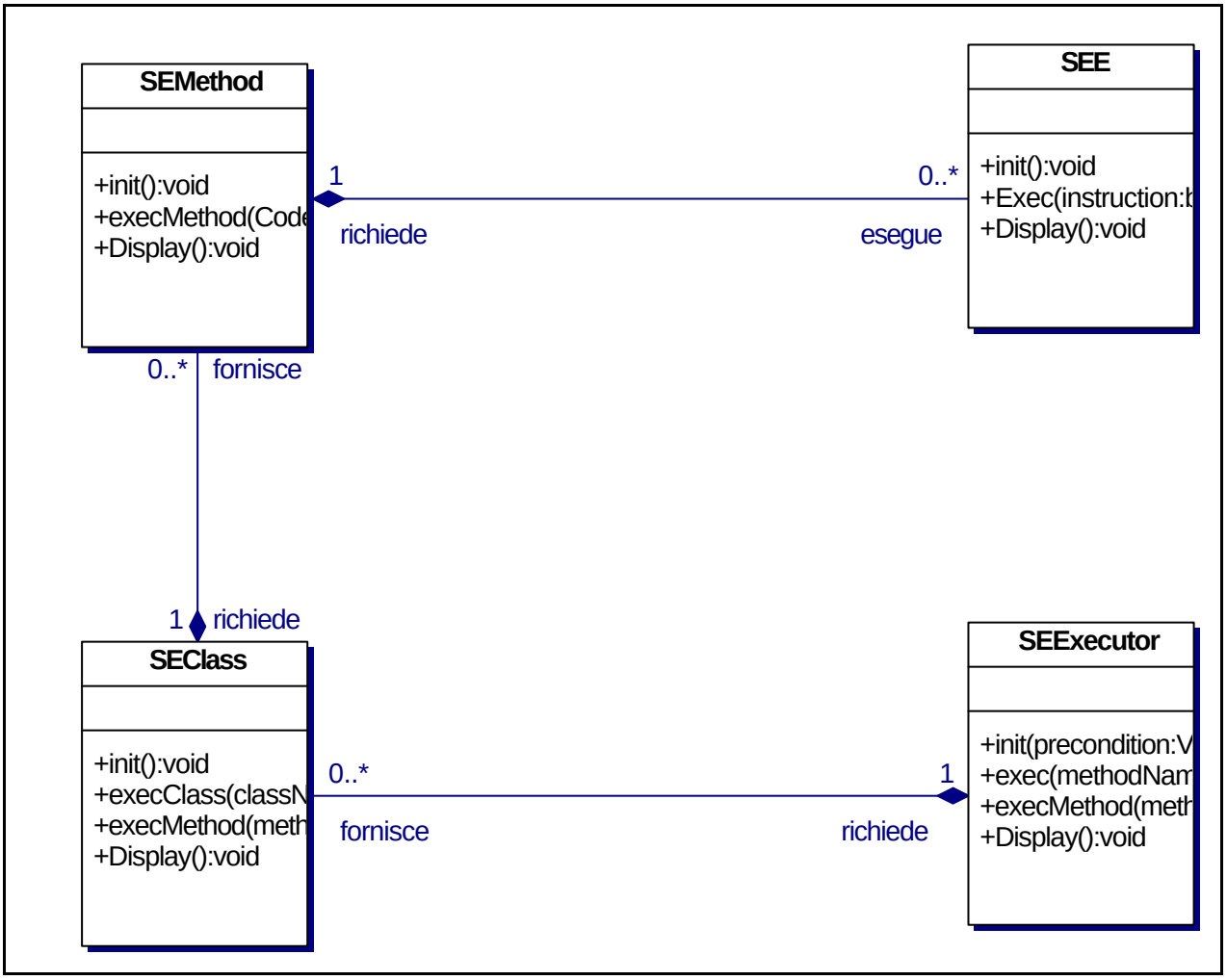
of ***ClassDiagram*** *SE*

Class SE.SEClass

Class SE.SEExecutor

Class Diagram *SE*

package: SE



Package dell'esecutore simbolico

Class Nodes

SEClass
SEE
SEExecutor
SEMethod

Class Detail

Class *SEClass*

package: **SE**

```
public class SEClass
```

Ask the execution of a method of a class, create active frame, resolve interaction of method interclass.

Field Summary	
private esClassFile	**lnkesClassFile**
private Facciata_SEE_SI	**lnkFacciata_SEE_SI**
private SEMethod	**lnkSEMethod**

Method Summary	
public void	**Display**() Display the current symbolic state.
public void	**execClass**(String className) Simbolically executes all the methods of a class.
public void	**execMethod**(String methodName) Simbolically executes an entire method.
public void	**init**() Initializer of simbolic state.

Field Detail

lnkesClassFile

private esClassFile lnkesClassFile

---

lnkFacciata_SEE_SI

private Facciata_SEE_SI lnkFacciata_SEE_SI

---

lnkSEMethod

private SEMethod lnkSEMethod

Method Detail

Display

public void Display()

Display the current symbolic state.

---

execClass

public void execClass(String className)

Simbolically executes all the methods of a class.

Parameter doc:
ClassName: The name of the class to be opened

---

execMethod

public void execMethod(String methodName)

Simbolically executes an entire method.

Parameter doc:
ClassName: Name of the class to be analized
MethodName: Name of the method to be executed

---

init

public void init()

Initializer of simbolic state.

Class *SEE*

package: **SE**

public class SEE

Engine of simbolic execution, implements the algorithms of simbolic execution, it executes an instruction to the time, and modifies the state of the machine.

Field Summary	
private Facciata_ClassFile	**lnkFacciata_ClassFile**
private Facciata_SEE_SI	**lnkFacciata_SEE_SI**

Method Summary	
public void	**Display**() Display the current symbolic state.
public instruction	**Exec**(byte[] instruction) Simbolically executes an instruction and returns the next one to be executed.
public void	**init**() Initializer of simbolic state.

Field Detail

lnkFacciata_ClassFile

private Facciata_ClassFile lnkFacciata_ClassFile

---

lnkFacciata_SEE_SI

private Facciata_SEE_SI lnkFacciata_SEE_SI

Method Detail

Display

public void Display()

Display the current symbolic state.

Exec

public instruction Exec(byte[] instruction)

Simbolically executes an instruction and returns the next one to be executed.

---

init

public void init()

Initializer of simbolic state.

Class *SEExecutor*

package: **SE**

public class SEExecutor

Ask the execution of a method to a class or creation of a class, resolve the intraclasses interactions (runtime constant pool) selecting the class to create.

Field Summary	
private esClassFileInterface	**lnkesClassFileInterface**
private SEClass	**lnkSEClass**

Method Summary	
public void	**Display**() Display the current symbolic state.
public void	**exec**(String methodName, String className, int line) Simbolically executes a method like debugging, starting from a specific point.
public void	**execMethod**(String methodName, String className) Simbolically executes an entire method.
public void	**init**(Valore precondition) Initialize simbolic executer with precondition.

Field Detail

lnkesClassFileInterface

private esClassFileInterface lnkesClassFileInterface

---

lnkSEClass

private SEClass lnkSEClass

Method Detail

Display

public void Display()

Display the current symbolic state.

---

exec

public void exec(String methodName, String className, int line)

Simbolically executes a method like debugging, starting from a specific point.

---

execMethod

public void execMethod(String methodName, String className)

Simbolically executes an entire method.

---

init

public void init(Valore precondition)

Initialize simbolic executer with precondition.

Class *SEMethod*

package: **SE**

public class SEMethod

Select the instruction to execute according to the analysis previously carried out, pass the instruction to the engine of simbolic execution.

Field Summary	
private SEE	**lnkSEE**

Method Summary	
public void	**Display**() Display the current symbolic state.

Method Summary	
public void	**execMethod**(byte[] Code) Simbolically executes an entire method.
public void	**init**() Initializer of simbolic state.

Field Detail

lnkSEE

private SEE lnkSEE

Method Detail

Display

public void Display()

Display the current symbolic state.

---

execMethod

public void execMethod(byte[] Code)

Simbolically executes an entire method.

---

init

public void init()

Initializer of simbolic state.

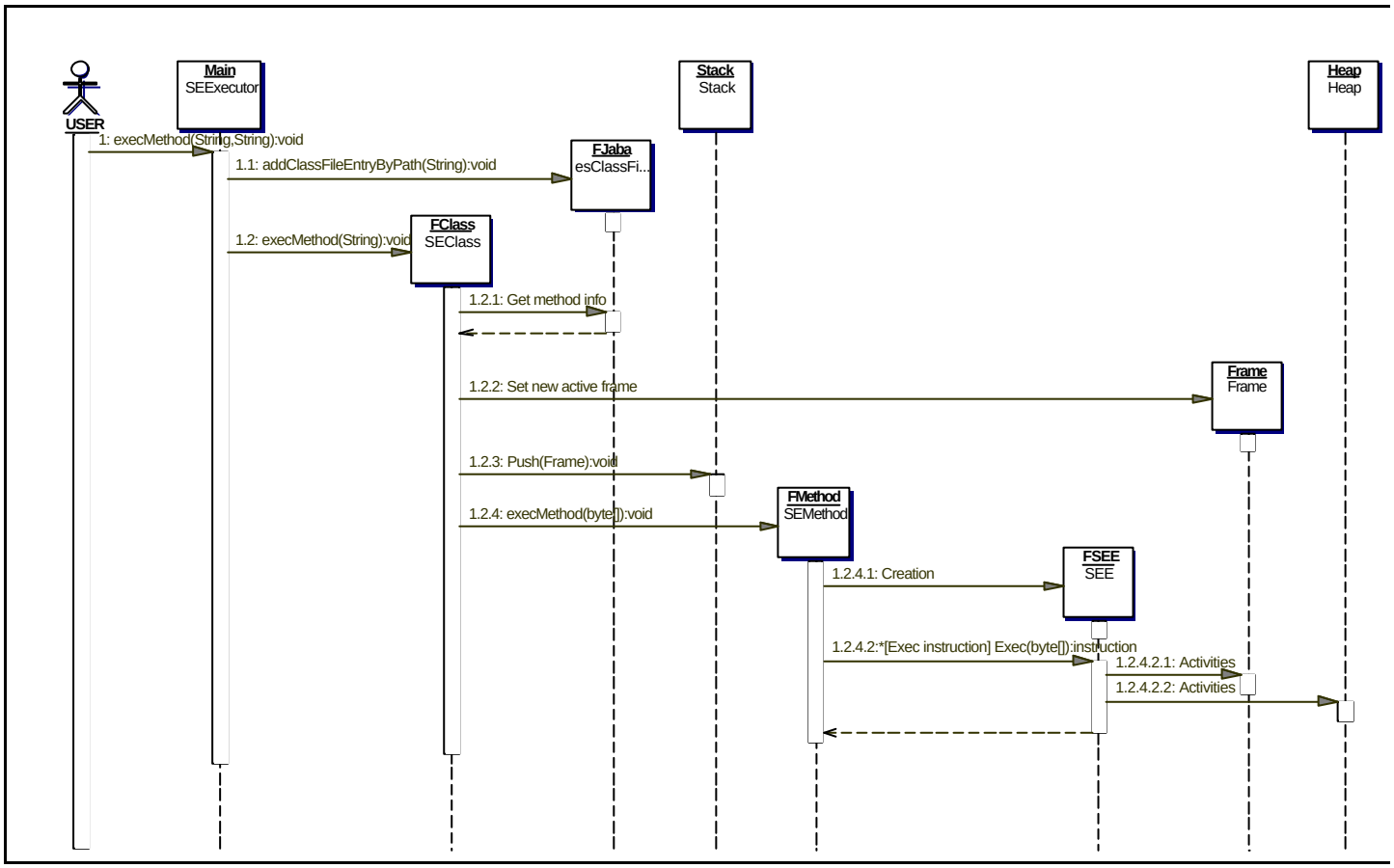
Interaction Diagrams

Sequence Diagram *Sequence*

package: SE

Object Summary

- FClass
- FJaba
- FMethod
- Frame
- FSEE
- Heap
- Main
- Stack
- USER



Object Detail

Object *FClass*

Instantiates:
SE.SEClass

created:
true

Message Detail

Get method info to **Object** FJaba

Number:
1.2.1

Synchronization:
call

sendingInstant:
265

processingDuration:
20

return arrow

Set new active frame to **Object** Frame

Number:
1.2.2

creation message

Synchronization:
call

sendingInstant:
347

processingDuration:
50

to **Object** Stack

Number:
1.2.3

Synchronization:
call

sendingInstant:
418

processingDuration:
20

Operation:
Memory.Stack.Push(Memory.Frame)

operationNameAsText:
'Push(Frame):void'

to **Object** FMethod

Number:
1.2.4

creation message

Synchronization:
call

sendingInstant:
466

processingDuration:
204

Operation:
SE.SEMethod.execMethod(byte[])

operationNameAsText:
'execMethod(String,String):void'

Object *FJaba*

Instantiates:
esClassFile.esClassFileInterface

created:
true

Object *FMethod*

Instantiates:
SE.SEMethod

created:
true

Message Detail

to **Object** FSEE

Number:
1.2.4.2

Synchronization:
call

sendingInstant:
592

processingDuration:
68

Operation:
SE.SEE.Exec(byte[])

operationNameAsText:
'Exec():instruction'

return arrow

Iteration:
Exec instruction

Creation to **Object** FSEE

Number:
1.2.4.1

creation message

Synchronization:
call

sendingInstant:
522

processingDuration:
50

Object *Frame*

Instantiates:
Memory.Frame

created:
true

Object *FSEE*

Instantiates:
SE.SEE

created:
true

Message Detail

Activities to **Object** Frame

Number:
1.2.4.2.1

Synchronization:
call

sendingInstant:
605

processingDuration:
20

Activities to **Object** Heap

Number:
1.2.4.2.2

Synchronization:
call

sendingInstant:
630

processingDuration:
20

Object *Heap*

Instantiates:
Memory.Heap

Object *Main*

Instantiates:
SE.SEExecutor

Message Detail

to **Object** FJaba

Number:
1.1

creation message

Synchronization:
call

sendingInstant:
140

processingDuration:
50

Operation:
esClassFile.esClassFileInterface.addClassFileEntryByPath(java.lang.String)

operationNameAsText:
'addClassFileEntryByPath(String):void'

to **Object** FClass

Number:
1.2

creation message

Synchronization:
call

sendingInstant:
210

processingDuration:
470

Operation:
SE.SEClass.execMethod(java.lang.String)

operationNameAsText:
'execMethod(String,String):void'

Object *Stack*

Instantiates:
Memory.Stack

Object *USER*

Stereotype:
actor

Message Detail

to **Object** Main

Number:
1

Synchronization:
call

sendingInstant:
115

processingDuration:
575

Operation:
SE.SEExecutor.execMethod(java.lang.String,java.lang.String)

operationNameAsText:
'execMethod(String,String):void'