INGEGNERIA DEL SOFTWARE

Prof. TO DO
10 CFU

Tommaso Fontana
Valentina Deda

Lecture Notes
Year 2017/2018

Triennale Ingegneria Informatica
Politecnico di Milano
Italy
3 gennaio 2018

Indice

1 JML

1.1 Dominio

In JML posso usare **SOLO**:

- \result
- I parametri formali
- Attributi pubblici
- Metodi Puri

1.2 Pre-Condizioni

Introdotte da “requires”

Sono condizioni dei parametri sotto le quali vale la specifica.

Se non vengono rispettate il metodo puo’ avere comportamento indefinito (spetta al **chiamante** fare i controlli, non al metodo!).

1.3 Post-Condizioni

Introdotte da “ensure”

Effetto garantito **AL TERMINE** del metodo. Cioe’ devo specificare che vincolirispetta il risultato (= se ho un certo risultato cosa deve risultare vero, cosa deve aver fatto il metodo, ecc...).

1.4 Eccezioni

Introdotte da “signals(*TipoEccezione* e”

Dice che cosa deve essere vero **SE** il metodo ha sollevato l’eccezione (= se ho chiamato eccezzione **ALLORA** so che deve valere quello che il signals indica).

1.5 Sintassi

Le specifiche JML vanno messe prima della dichiarazione del metodo.

Sono contenute in un commento ed ogni riga inizia con una @.

Esempi:

```
// @ ...
/*
   @ ...
   @ ...
   @ ...
*/
```

1.6 \result

E' il valore ritornato dal metodo.

1.7 Omissione

Omettere pre o post condizione equivale a scrivere rispettivamente "require true" o "ensure true"

1.8 Commenti

Alcune specifice sono piu' comprensibli in linguaggio naturale. Vanno espresse come (*...*) con la specifica in linguaggio naturale tra gli asterischi.

1.9 Operatori JML

- \old(x): restituisce il valore del parametro x **AL MOMENTO DELLA CHIAMATA**
- assigable: indica che un **PARAMETRO** puo' essere modificato, va messo **prima** della pre-condizione.
- **Metodi pubblici dele collezioni**: equals,contains,containsAll,get,subList...
- Quantificatori:
  - \forall
  - \exist
  - \sum, \product, \min, \max
  - \num-of: restituisce il nimero di volte per cui un parametro valgono le condizioni.

1.10 Astrazioni

La specifica definisce loggetto astratto , ad esempio "insieme di numeri interi" l'implementazione definisce l'oggetto concreto, ad esempio "array di int".

1.11 OAT

Se ho bisogno di specificare metodi non puri o comunque mi serve conoscere la struttura del implementazione uso l'OAT.

Questo significa che in una riga JML scrivero' *tipo del OAT nel implementazione* oat

Esempio:

```
1   // @ spec_public List<Int> oat;
```

e poi un commento che ridescrive le caratteristiche.

Cosi posso usare metodi che, se non conoscessi l'implementazione non potrei usare, tipo subList per una lista.

1.12 Public Invariant

Definisco propriet' che devo essere sempre vere per l'oggetto Astratto, posso usare **SOLO** metodi e attributi pubblici.

1.13 Verificare gli invariant

Uso il metodo di induzione:

- Verifico che valga **al termine** del costruttore.
- **Suppongo** che valga **alla chiamata** di ogni modificatore, devo dimostrare che vale anche al termine

1.14 Rep

E' Struttura dati che rappresenta i valori degli oggetti dell' ADT.

E' insieme di variabili **private** e delle operazioni che l'oggetto svolfe (**Implementate**)

1.15 Funzione di Astrazione(AF)

Specifica il legame fra stato concreto ed astratto.

Si usa un **Private Invariant** per descriverla. In particolare nell'invariant descrivo la relazione fra parti private ed osservatori.

Deve mettere in evidenza il legame fra rappresentatore ed oggetto, descrive "l'implementazione" del Rep.

In sostanza **Associa ad ogni oggetto concreto il suo oggetto Astratto**.

1.16 Rep Invariant(RI)

Specifica le proprieta' che **la Rep deve rispettare**. Come a AF e' un private invariant.

NB il RI puo' diventare falso durante il metodo, basta che sia **Vero alla chiamata ed al ritorno**.

Si puo' **Implementare** ad esempio un repOk un metodo che verifica che RI sia rispettato. Si definisce il metodo repOk con tutti i controlli per l'RI, poi (o si chiama nei mutatori e costrutti tramite le **Asserzioni**(= asserzioni sono condizioni che se non sono verificate a RUNTIME danno **AssertionError** e bloccano l'esecuzione.))

NB: Non posso fornire 'accesso esterno a parti mutabili del rep, altrimenti **ESPLODE TUTTO**. Quindi devo evitare:

- di restituire come ritorno il riferimento ad una componente mutabile del rep.
- di mettere nel rep una componente mutabile che ha un riferimento esterno all'oggetto.
- dichiarare attributi pubblici.

1.17 Specifiche totali

le specifiche totali hanno:

- requires true
- precondizioni nell' ensure
- violazione delle pre-condizioni nel signal

2 Estensioni

2.1 Estensione

E' un altro modo per dire ereditarieta'. Un' esntensione e' **pura** se non modifica la specifica dei metodi.

2.2 Rappresentazione per estensioni

Di solito la sottoclasse mantiene la AF della superclasse. Non modifica **MAi** l'RI della superclassse ed aggiunge il proprio RI.

2.3 Verificare validita' delle estensioni

- **Principio di sostituzione di Liskov**: "ogni modulo che usa un oggetto della superclasse deve poter usare un oggetto della sottoclasse senza accorgersi della differenza"
- **Regola della segnatura**: "La sottoclasse deve avere tutti i metodi della superclasse con la stessa intestazione ma puo' avere **meno** eccezioni'
- **Regola dei metodi**: "La sottoclasse deve avere la stessa specifica della superclasse." pero' posso:
  - **Indebolire la pre-condizione** = aggiungere altre pre-condizioni in **OR** a quelle della superclasse.
  - **Rafforzare la post-condizione** = aggiugnere altre post-condizioni in **AND** a quelle della superclasse.
- **Regola delle proprieta'** la sottoclasse deve conservare tutti i public invariant dela superclasse, e anche le sue proprieta' evolutive.

2.4 Sintassi JML

Introduco nuove condizioni con un

```
1 // @ also ...
```

3 Testing

3.1 Tipi di Testing

- White Box: si generano casi di test basandosi sulla specifica.
- Black Box: si generano casi di test basandosi sul codice.

3.2 BlackBox / Funzionale

- Genero casi di test per ogni “categoria” dei parametri
- Genero casi per ogni corner case fra “categorie”

3.3 Tipid i test Black Box

Test combinatori:

- Identifico attributi che possono cambiare durante lesecuzione
- Si generano combinazioni dei possibili valori degli attributi
- Le combinazioni devono contemplare:
  - “valori standard”
  - Corner case (= ad esempio se ho valori in un intervallo prendo gli estremi)
  - valori errati

3.4 WhiteBox / Strutturale

Genero casi di test per “percorrere” tutto il programma Quindi percorrere tutti i cammini possibli. I cammini sono le strade che il programma puo’ prendere, cioe’ tutti **i modi in cui puo’ arrivare dall’ inizio alla fine**. Possono anche vederli come la sequenza di righe di codice che eseguo per arrivare alla fine.

3.5 Tipi di test White-Box

- **Copertura dei cammini**: Genero casi che portano ad eseguire tutte le psosibili combinazioni di istruzioni. Devo contare le iterazioni dei cicli ed i metodi in cui posso uscire da essi per avere l numero totale dei cammini.
- **Copertura delle istruzioni**: Genero casi per cui eseguo ogni istruzione **Almeno** una volta.
- **Copertura delle Diramazioni**: Genero casi che mi permettono di coprire tutte le diramazioni, cioe' immaginidnando il programma come diagramma di flusso, devo poter toccare ogni nodo.
- **Copertura delle condizioni**: genero casi che coproono tutti i possibili **Risultati** delle condizioni dei cicli e degli if.

3.6 Altri tipi di test

- **Test di unita'**: testo ogni modulo separatamente.
- **Test di integrazione**: testo sottinsieme a mano a mano piu' grandi di moduli vedendo come ingeragiscono i moduli nei sottinsiemi.
- **Test di sistema**: testo il programma completo.
- **Test di regressione**: se il programma viene aggiornato, testo l'aggiornamento.

3.7 ScaffHolding

: E' composto da:

- Driver ⟶ Componente che simula la parte di programma che invoca il modulo da testare. Prepara l'ambiente, i parametr, fa chiamate accessorie e verifica che vadano a buon fine.
- Stub ⟶ Componente che simula la parte chaimata dal modulo (= cioe' esegue il modulo da testare, verigica l'ambiente creato dal driver e i parametri) restituisce i risultati richiesti dalle specifiche.

Di base lo scaffholding serve per automatizzare i Test.

4 Pattern

4.1 Design Pattern

Sono dele soluzioni “riutilizzabli” di cui esiste gia’ uno schema da seguire e che puo’ adattare a varie situazioni.

4.2 Tipi di Pattern

- Creazionali
- Strutturai
- Comportamentali

4.3 Alcuni pattern

- **Factory Method**: e’ un metodo **NON COSTRUTTORE** che restituisce oggetti della sua classe, si implementa come metodo statico.Per usarlo devo dichiarare il costruttore come protected o private.
- **Singleton**: e0 un tipo di classe per cui puo’ esistere **UNA SOLA** istanza. Devo dichiarare il costruttore protected o private. Devo implementare un metodo che permetta di avere una sola istanza:
  - Dichiaro un attributo statico istanza.
  - Dichiaro un metodo che verifica che “istanza” sia null, se e’ un nul ritorna una nuova istanza altrimenti ritorna l’istanza gia’ esistente senza crearne di nuove.
- **Adapter**: E’ un interfaccia che si frappone fra due diverse interfacce,L’adapter richiama i metodi dell’oggetot da adattare e li “passa” all’interfaccia che deve uitlizzare sotto forma di utilizzabile.
- **Proxy**: Se ho oggetti “pesanti” posso sostiutirli con un piu’ leggero proxy che ne fa le veci. Il proxy “passa’ le richieste che gi vengono fatte all’oggetto pesante oppure se ne e’ in grado risponde direttamente ad esse (per esempio attraverso caching).
- **Observer**: e’ un oggetto che segnala quando un altro oggetto cambia stato.
- **Strategy**: Se ho diversi algoritmi possibili (e intercambiabili) per raggiungere un risultato, Strategy permette di selezionare a **RUNTIME** il migliore.
- **Comparator**: Esiste gia’ l’interfaccia comparator in java.util, la impemento affinche’ possa confrontare oggetti del tipo che mi interessa confrontare.
- **Decorator**:Permette di aggiungere nuove funzionalita’ agli oggetti a runtime. e’ un alternativa all’ereditarieta’. Si implementa passando come parametro l’oggetto da arricchire al costtruttore del decorator.

- **State**: Gli oggetti cambiano dinamicamente caratteristiche a seconda dello stato in cui si trovano, Si usa una classe astratta STATE e poi clsassi concrete che la impementano per ogni diverso stato.
- **Model-View-Controller**: COmposto da 3 elementi:
  - Model : fornisce metodi per accesso ai dati
  - View : visualizza i dati forniti da model e si occupa di gestire i dati a seconda dei comandi dell'utente.
  - Controler: riceve comandi dal utente esterno e modifcia lo stato di model e view di conseguenza.