24.
Scegliere l'interfaccia di un metodo

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

Interfaccia, prototipo e firma

- Per **interfaccia** di un metodo si intende la facciata che il metodo offre ai suoi chiamanti
  - Nome del metodo
  - Lista dei parametri formali
  - Tipo di ritorno
  - Modificatori (visibilità, *static*, *final*, etc.)
  - In Java, l'eventuale clausola "*throws* ..."

- Alcuni autori usano il termine "*method header*" per riferirsi all'interfaccia di un metodo
- In C++, si usa il termine **prototipo** per indicare l'intestazione di un metodo o funzione

- Per **firma** (*signature*) di un metodo si intende invece il nome del metodo e la lista dei parametri formali
- La firma contiene le uniche informazioni che sono rilevanti ai fini del binding dinamico

L'importanza dell'interfaccia

- In questa lezione discuteremo dei criteri da prendere in considerazione quando si voglia **scegliere oculatamente l'interfaccia migliore per un metodo** che deve svolgere un dato compito (contratto)

- L'argomento è particolarmente rilevante in alcuni contesti, tra cui:
  - metodi che manipolano **collezioni**
  - metodi che appartengono a **librerie** destinate ad un ampio uso

- Le collezioni si presentano in una ricca gerarchia di classi ed interfacce parametriche, rendendo complessa la scelta dell'interfaccia migliore per un metodo

- Per il successo di una libreria, è fondamentale che le interfacce dei metodi, e più in generale l'interfaccia pubblica della libreria, sia ben progettata, in modo da essere utile in quanti più contesti è possibile

Scelta dei parametri formali

- La scelta del tipo di parametri formali dovrebbe **rispecchiare il più fedelmente possibile la pre-condizione** del metodo in questione

- Quindi, il tipo scelto dovrebbe:
  - accettare tutti i valori che soddisfano la pre-condizione (*completezza*)
  - rifiutare tutti gli altri valori (*correttezza*)

- Purtroppo, spesso i limiti del linguaggio di programmazione usato impediscono di ottenere contemporaneamente correttezza e completezza

Scelta dei parametri formali: esempi

- Esempio di pre-condizione 1: il metodo accetta una **lista non vuota**
  - Non esiste in Java un tipo specifico per una lista non vuota
  - Ci accontenteremo di accettare una lista qualsiasi, per poi sollevare un'eccezione a run-time se la lista risulta vuota
  - La firma ottenuta è *completa* ma non *corretta*

- Esempio di pre-condizione 2: il metodo accetta un **intero non negativo**
  - In Java non esiste un tipo specifico per un intero non negativo
  - Ci accontenteremo di accettare un intero qualsiasi, per poi sollevare un'eccezione se è negativo
  - La firma ottenuta è *completa* ma non *corretta*
  - In C++, sarebbe possibile ottenere una firma corretta e completa usando il tipo “unsigned int”

Correttezza VS Completezza

- Quando non esiste una scelta corretta e completa, si deve scegliere tra due opzioni:
  A) Scartare tutti i valori non validi ed anche alcuni valori validi (firma corretta ma non completa)
  B) Accettare tutti i valori validi ed anche alcuni valori non validi (firma completa ma non corretta)

- Di norma, si preferisce la scelta B, in quanto i valori non validi che vengono accettati possono essere successivamente scartati a run-time (lanciando un'eccezione)
  - Quindi, in linea di massima assegneremo alla completezza una priorità più alta della correttezza

- In un contesto in cui sia prioritaria la robustezza e quindi si voglia minimizzare il rischio di errori a run-time, potrebbe essere preferibile la scelta A

Violare la **completezza**

- Il criterio di completezza o generalità richiede che il metodo accetti tutti i valori che soddisfano la pre-condizione
- Violare la completezza rende il metodo **meno utile**, in quanto non è applicabile a tutti i valori previsti dal contratto
- In altri termini, una firma non completa sta effettivamente **restringendo la pre-condizione**

Violare la **correttezza**

- Scegliere una firma che viola la correttezza comporta che il compilatore consentirà ai chiamanti di passare al metodo in questione dei valori non validi

- In tal modo, si indebolisce la capacità del compilatore di verificare, tramite il type-checking, il rispetto delle pre-condizioni

- La parte della pre-condizione che non viene espressa nella firma dovrà essere verificata a run-time e potrà portare ad errori a tempo di esecuzione (lancio di eccezioni)
  - in tal modo, anche se la firma viola la correttezza, non si sta modificando (cioè indebolendo) la pre-condizione

Esempio

- Consideriamo un metodo con la seguente **pre-condizione:**

  accetta una collezione priva di duplicati

- In Java, possiamo interpretare la pre-condizione in vari modi:

1) `void f(`**`Set`**`<?> s)`

   Corretta ma non completa. Anche una lista può essere priva di duplicati.

2) `void f(`**`Collection`**`<?> s)`

   Completa ma non corretta. Il metodo può verificare a tempo di esecuzione che la collezione non contenga duplicati.

Vantaggi di 1: la pre-condizione è verificata dal compilatore. E' impossibile per il chiamante sbagliare e violare la pre-condizione. Diminuisce il rischio di errori a tempo di esecuzione.

Vantaggi di 2: il metodo offre un servizio più ampio, cioè accetta anche le liste.

Esempio *addAll*

- Ad esempio, consideriamo un metodo *addAll* che accetta due collezioni e aggiunge tutti gli elementi della prima alla seconda

- Se andiamo a precisare questo contratto, otteniamo:
  - **Pre-cond:** accetta due collezioni, tali che gli elementi della prima possono essere inseriti nella seconda
  - **Post-cond:** alla fine dell'esecuzione, la seconda collezione conterrà gli elementi che conteneva all'inizio, più tutti quelli della prima collezione; la prima collezione non verrà modificata

- Il fatto che gli elementi della prima collezione possano essere inseriti nella seconda si traduce in:

  Il tipo degli elementi della prima collezione è sottotipo del tipo degli elementi della seconda collezione

Esempio *addAll*: una possibile firma

Esaminiamo un'interfaccia possibile per il metodo a*ddAll*:

1) void addAll(Collection<?> a, Collection<?> b)

Esempio *addAll*: una possibile firma

Esaminiamo un'interfaccia possibile per il metodo a*ddAll*:

1) void addAll(Collection<?> a, Collection<?> b)

È completa ma **non è corretta**, perché accetta dei valori come a=Collection<String> e b=Collection<Employee>, che non soddisfano la pre-condizione, perché non è possibile inserire delle stringhe in una collezione di Employee.

Negli esempi introduttivi, abbiamo accettato altre firme complete, ma non corrette.
In questo caso, due problemi ci impediscono di accettare questa firma.

Per prima cosa, a causa delle limitazioni dei tipi generici in Java (si veda la lezione sulle conseguenze dell'*erasure*), non sarebbe possibile verificare a tempo di esecuzione che le due collezioni fossero compatibili tra loro, perché i parametri effettivi di tipo delle due collezioni vengono persi durante la compilazione.

Cosa più importante, la firma in questione non consente di scrivere un corpo corretto per il metodo, in quanto **non è possibile invocare il metodo "add"** della collezione "b"!

Quindi, dobbiamo aggiungere alla nostra discussione un **altro criterio** di valutazione, che chiameremo **funzionalità.**

Quest'interfaccia per addAll **non è funzionale.**

- Per quanto sia comune scegliere una firma che accetta un tipo più generale di quello richiesto dalla pre-condizione, la firma non deve mai accettare un tipo così generico da impedire di portare a termine il compito assegnato al metodo

- Quindi, è possibile scegliere una firma che violi la correttezza, a patto di preservare la **funzionalità** del metodo stesso
- Cioè, il tipo scelto deve contenere le informazioni (campi) e le funzionalità (metodi) necessarie a svolgere il compito previsto

- Nota: nel valutare la funzionalità assumiamo di non voler ricorrere a conversioni forzate (cast)

Esempio *addAll*: altre firme

Esaminiamo altre interfacce possibili per il metodo *addAll*:

2) <T> void addAll(Collection<T> a, Collection<T> b)

È funzionale, corretta ma **non completa**, perché non accetta valori validi come a=Collection<Manager> e b=Collection<Employee>.

3) <S, T extends S> void addAll(Collection<T> a, Collection<S> b)

È funzionale, corretta e completa.

4) <T> void addAll(Collection<T> a, Collection<? super T> b)

È funzionale, corretta e completa.

5) <T> void addAll(Collection<? extends T> a, Collection<T> b)

È funzionale, corretta e completa.

6) <T> void addAll(Collection<? extends T> a, Collection<? super T> b)

È funzionale, corretta e completa.

Ci sono criteri per confrontare le interfacce 3, 4, 5 e 6?

Quale delle quattro interfacce è preferibile?

Altri criteri: esprimere ulteriori garanzie

- In alcuni casi, è possibile esprimere nel tipo dei parametri formali delle **garanzie offerte dalla post-condizione**, come il fatto che un dato parametro non sarà modificato

- In C++, il modificatore *const* serve proprio ad esprimere questo vincolo
- Ad esempio, se Person è una classe, un parametro di tipo "const Person&" è un riferimento ad un oggetto Person che non può essere utilizzato per modificare l'oggetto
  - Tecnicamente, il riferimento in questione può essere usato solo per invocare metodi che siano a loro volta dichiarati const

- In Java, il tipo jolly può esprimere una proprietà simile, nel caso delle collezioni
- Un parametro di tipo "Collection<? extends Employee>" sostanzialmente non può essere utilizzato per **modificare** la collezione
  - Più precisamente, non è possibile invocare il metodo **add**, se non con argomento *null*
  - Tuttavia, è possibile invocare il metodo remove, perché il suo argomento è di tipo Object
- Analogamente, un parametro di tipo "Collection<? super Employee>" non può essere utilizzato per **leggere** il contenuto della collezione
  - Più precisamente, si può accedere agli oggetti contenuti soltanto come se fossero degli *Object*

Altri criteri: semplicità

- Infine, un ultimo criterio che vale la pena menzionare riguarda la **semplicità dell'interfaccia**
- A parità degli altri criteri, è preferibile un'interfaccia che sia più semplice da leggere e comprendere

- Nel caso dei metodi parametrici, questo criterio implica che, a parità delle altre caratteristiche, è preferibile un'interfaccia che utilizza **meno parametri di tipo**

Esempio *addAll*: la scelta migliore

Riesaminiamo le ultime 4 firme per il metodo *addAll*, tutte funzionali, corrette e complete:

3) <S, T extends S> void addAll(Collection<T> a, Collection<S> b)

4) <T> void addAll(Collection<T> a, Collection<? super T> b)

È più semplice della 3 perché usa un solo parametro di tipo.

Inoltre, garantisce che la collezione b non sarà letta.

5) <T> void addAll(Collection<? extends T> a, Collection<T> b)

Usa un solo parametro di tipo e quindi è più semplice della 3.

Inoltre, garantisce che la prima collezione non sarà modificata (almeno, non tramite il metodo *add*).

6) <T> void addAll(Collection<? extends T> a, Collection<? super T> b)

Usa un solo parametro di tipo e garantisce che la prima collezione non sarà modificata e la seconda non sarà letta.

Quindi, scegliamo l'interfaccia 6.

Infatti, nella classe Collections della API Java troviamo:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

Scelta dei parametri formali

- **Riassumendo**, nella scelta del tipo di parametri formali si possono individuare le seguenti forze in gioco, elencate in ordine di **importanza decrescente**:

1) Funzionalità (il tipo prescelto deve offrire le funzionalità necessarie a svolgere il compito prefissato, cioè a realizzare la post-condizione)
2) Completezza (il metodo dovrebbe accettare *tutti* i valori che soddisfano la pre-condizione)
3) **Correttezza** (il metodo dovrebbe accettare *soltanto* valori che soddisfano la pre-condizione)
4) **Ulteriori garanzie** (il tipo dei parametri dovrebbe esprimere eventuali garanzie previste dalla post-condizione)
5) **Semplicità** (a parità degli altri criteri, la firma dovrebbe essere più semplice possibile)

- Il criterio 1 è obbligatorio, mentre gli altri vanno considerati come qualità da massimizzare, ma che non sempre è possibile soddisfare a pieno, anche a causa delle limitazioni imposte dal linguaggio di programmazione utilizzato

Scelta del tipo di ritorno

- Nel caso del tipo di ritorno, solitamente l'unica decisione da prendere riguarda la sua specificità
- A questo proposito, due forze spingono in direzioni opposte:

1) Un tipo di ritorno **più specifico** è più utile per il chiamante, perché esprime più informazioni sul valore restituito
2) Un tipo di ritorno **meno specifico** nasconde l'implementazione interna del metodo (incapsulamento) e quindi favorisce l'evoluzione futura del software

- Il progettista dovrà trovare un **compromesso** tra queste due forze

- Il contesto suggerirà se è più importante l'utilità per il chiamante o la futura evoluzione del software in questione

Esempi di scelta del tipo di ritorno

- Ad esempio, tra List<?>, List<Employee> e List<Manager>, è più utile restituire List<Manager> perché fornisce più informazioni degli altri tipi sul valore restituito
  - Questo vale anche se List<Manager> non è sottotipo di List<Employee>
  - Se il chiamante non è interessato alle sottoclassi di Employee, può sempre assegnare il risultato a List<? extends Employee>

- Invece, tra List<Manager> e LinkedList<Manager> di solito si preferisce la prima, per lasciare la possibilità di cambiare implementazione di lista in futuro
  - ad es., per poter usare ArrayList invece di LinkedList

Best practice

Scegliere il tipo di ritorno in modo che conservi l'informazione di tipo presente nei parametri formali, ma nasconda i dettagli implementativi del metodo

Tipo di ritorno e parametri formali

- A volte, la specificità del tipo di ritorno può essere in contrasto con i criteri 4 e 5 (ulteriori garanzie e semplicità) della scelta dei parametri formali
  - come nell'esempio "intersezione insiemistica", nel seguito

- In tal caso, solitamente si preferisce dare più importanza al tipo di ritorno

Esempio: intersezione insiemistica

- Consideriamo il problema di scegliere l'interfaccia di un metodo che accetta due insiemi (Set) e **restituisce l'intersezione dei due**

- Prima di esaminare alcune interfacce possibili, è opportuno **chiarire il contratto** del metodo ed in particolare la sua pre-condizione

- Quali restrizioni devono valere sui due insiemi passati come argomenti?

- Idealmente, dovremmo trarre ispirazione dalla definizione matematica di intersezione, che definisce l'operazione su *qualsiasi coppia di insiemi*
- Quindi, se è possibile dovremmo accettare come argomenti due insiemi di **qualsiasi tipo**

Sul contratto dell'intersezione insiemistica

- Può sorprendere che si vogliano accettare come argomenti anche due insiemi di tipi non "imparentati"

- In alcuni casi, come un *Set<Employee>* e un *Set<String>*, sappiamo a priori che gli insiemi non possono contenere elementi in comune
  - Non ha senso calcolare l'intersezione, perché è sicuramente **vuota**
  - Potremmo scegliere di **scartare** questo tipo di argomenti

- Tuttavia, si consideri il caso di un *Set<Cloneable>* e un *Set<Comparable<?>>*
- Anche se le interfacce Cloneable e Comparable non hanno alcun collegamento, niente impedisce che tali insiemi abbiano degli elementi in comune (oggetti che implementano entrambe le interfacce)
- Quindi, dovremmo accettare questo tipo di argomenti

Esempio: intersezione insiemistica

Consideriamo diverse interfacce:

1) <T> Set<T> intersection(Set<T> a, Set<T> b)

2) <T> Set<T> intersection(Set<T> a, Set<? extends T> b)

3) Set<?> intersection(Set<?> a, Set<?> b)

4) <S,T> Set<S> intersection(Set<S> a, Set<T> b)

5) <S> Set<S> intersection(Set<S> a, Set<?> b)

Esempio: intersezione insiemistica

Consideriamo diverse interfacce:

1) <T> Set<T> intersection(Set<T> a, Set<T> b)

È funzionale, corretta ma **non completa**. Rifiuta insiemi di tipo diverso.

2) <T> Set<T> intersection(Set<T> a, Set<? extends T> b)

È funzionale, corretta ma **non completa**. Rifiuta insiemi di tipo non correlato, come Set<Cloneable> e Set<Comparable<?>>.

3) Set<?> intersection(Set<?> a, Set<?> b)

È funzionale, completa e corretta. Inoltre, esprime forti garanzie su entrambi i suoi argomenti.

Però, il tipo di ritorno non ha alcuna relazione con quello dei due argomenti ed è poco utile al chiamante.

4) <S,T> Set<S> intersection(Set<S> a, Set<T> b)

Come la 3, è funzionale, completa e corretta. Non esprime garanzie sugli argomenti.

Il tipo di ritorno è più specifico, quindi migliore, rispetto a quello della 3, in quanto conserva il tipo del primo argomento.

5) <S> Set<S> intersection(Set<S> a, Set<?> b)

È simile alla 4, però usa un parametro di tipo in meno ed esprime ulteriori garanzie su b.

Esempio: intersezione insiemistica

In base ai criteri presentati, l'interfaccia migliore risulta la **5**, in quanto è l'unica che:

- è funzionale, completa e corretta
- preserva nel tipo di ritorno una parte dell'informazione di tipo presente negli argomenti
- esprime la garanzia che il secondo argomento non verrà né letto né scritto
- usa un solo parametro di tipo

Esempio: intersezione insiemistica

Per conferma, esaminiamo il metodo di intersezione fornito dalla libreria **Google Guava**, nella classe Sets:

```
public static <E> Sets.SetView<E> intersection(Set<E> a, Set<?> b)
```

Nota: Sets.SetView è una interfaccia che estende Set e rappresenta una vista su un insieme.

La struttura è proprio quella della nostra firma 5.

L'**API Java** offre invece il seguente metodo nell'interfaccia Collection<E>:

```
public boolean retainAll(Collection<?> other)
```

Il metodo modifica questa (this) collezione, lasciando solo gli elementi presenti nella collezione other. Quindi, effettua l'intersezione tra this e other.

Anche in questo caso, viene accettato un insieme di qualunque tipo.

Esempio negativo

Nel seguente brano, tratto da un manuale di programmazione Android, una delle tre firme proposte non è funzionale:

Supponiamo di voler creare un metodo di ordinamento di oggetti di tipo T contenuti in una lista attraverso questo tipo di Comparator. La firma di questo metodo […] sarà del tipo:

```
public <T> void sort(List<T> l, Comparator<T> c)
```

[…] una List<Dog> potrà essere ordinata solo se si ha a disposizione un Comparator<Dog>. In realtà sappiamo che un Dog è un Animal per cui potremmo ordinare la lista di Dog anche se avessimo a disposizione un Comparator<Animal> o un Comparator<Object>. Nel caso in cui decidessimo di permettere questa cosa, la firma del metodo diventerebbe la seguente:

```
public <T> void sort(List<T> l, Comparator<? super T> c)
```

Se poi volessimo ampliare ulteriormente, potremmo scrivere lo stesso metodo nel seguente modo:

```
public <T> void sort(List<? extends T> l, Comparator<? super T> c)
```

Esercizio (esame 3/7/2014, #4)

Implementare il metodo *subMap*
che accetta una mappa e una collezione e restituisce una nuova mappa
ottenuta restringendo la prima alle sole chiavi che compaiono nella collezione.
Il metodo non modifica i suoi argomenti.

Valutare le seguenti intestazioni per il metodo *subMap*,
in base ai criteri di funzionalità, completezza, correttezza,
fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.
Infine, scegliere l'intestazione migliore oppure proporne un'altra.

(a) <K> Map<K,?> subMap(Map<K,?> m, Collection<K> c)
(b) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<?> c)
(c) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<? super K> c)
(d) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<? extends K> c)
(e) <K,V> Map<K,V> subMap(Map<K,V> m, Set<K> c)
(f) <K,V,K2 extends K> Map<K,V> subMap(Map<K,V> m, Collection<K2> c)

Esercizio (esame 31/1/2014, #4)

Il metodo *isMax* accetta un oggetto *x*, un comparatore ed un insieme di oggetti,
e restituisce *true* se, in base al comparatore,
x è maggiore o uguale di tutti gli oggetti contenuti nell'insieme.
Altrimenti, il metodo restituisce *false*.

Valutare ciascuna delle seguenti intestazioni per il metodo *isMax*,
in base ai criteri di funzionalità, completezza, correttezza,
fornitura di ulteriori garanzie e semplicità.
Infine, scegliere l'intestazione migliore oppure proporne un'altra,
motivando brevemente la propria scelta.

(a) boolean isMax(Object x, Comparator<Object> c, Set<Object> s)
(b) <T> boolean isMax(T x, Comparator<T> c, Set<T> s)
(c) <T> boolean isMax(T x, Comparator<? super T> c, Set<T> s)
(d) <T> boolean isMax(T x, Comparator<? extends T> c, Set<? super T> s)
(e) <T> boolean isMax(T x, Comparator<? super T> c, Set<? super T> s)
(f) <S,T extends S> boolean isMax(T x, Comparator<? super S> c, Set<S> s)

How To Design A Good API and Why it Matters

Un seminario di Joshua Bloch, disponibile in video (YouTube) e come sequenza di slide