Ereditarietà e Concorrenza nei Linguaggi OO e Concorrenti

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Outline

Concurrent OO Languages (COOL)

Nozioni Storico-Generali.

Concurrent OO Languages (COOL).

Sin dall'esordio della metodologia OO si è tentato di sfruttarla nella costruzione di sistemi distribuiti/concorrenti.

Questa esigenza si è concretizzata nei COOL, linguaggi di programmazione che vorrebbero fondere le due metodologie.

- metà anni 80 grande entusiamo relativo ai COOL che si è però rivelato un settore di nicchia;
- metà anni 90, Java e C# riportano i COOL in auge.

Comunque la convivenza tra OOP e concorrenza rimane difficile.

Anomalia dell'Ereditarietà

Nozioni Generali e Caratteristiche.

Sincronizzazione ed ereditarietà hanno caratteristiche conflittuali che ostacolano il loro uso simultaneo senza rompere l'encapsulation degli oggetti.

In un sistema distribuito, l'insieme dei messaggi che un oggetto può gestire:

- non è uniforme rispetto al tempo, ma
- dipende dallo stato corrente degli oggetti stessi.

Inoltre, per assicurare i vincoli di sincronizzazione è necessario scrivere il relativo codice di sincronizzazione.

Caso di Studio: il Bounded Buffer.

Si consideri la classica implementazione di un buffer limitato e condiviso in ambiente distribuito:

```
public class BoundedBuffer {
  protected int top = 0;
  protected final int MAX = 20;
  protected int buffer[] = new int[MAX];

  public synchronized void put(int n) ... {
    while (top >= MAX) wait();
    buffer[top++] = n;
    notifyAll();
  }
  public synchronized int get() ... {
    while (top <= 0) wait();
    int result = buffer[--top];
    notifyAll();
    return result;
  }
}
```

- due metodi: `get()` e `put()`;
- `put()` non può essere chiamato se il buffer è pieno.
- `get()` non può essere chiamato se il buffer è vuoto;

I vincoli devono essere fatti rispettare dal buffer stesso.

Caso di Studio: il Bounded Buffer.

Nel caso sequenziale è possibile separare la gestione dei vincoli dal comportamento vero e proprio, perché:

- lo stato dell'oggetto è responsabilità dell'unico thread che lo usa;

Nel distribuito questo non è più vero:

- ci sono tanti client che accedono al servizio più o meno contemporaneamente (problema race-condition);
- tendenzialmente i client NON devono essere a conoscenza dello stato dell'oggetto .

Pertanto i vincoli di sincronizzazione devono essere gestiti direttamente dall'oggetto il cui accesso è sottoposto a vincoli.

Come vedremo, mischiare il codice funzionale con il codice relativo alla sincronizzazione non è una prassi che dà buoni frutti.

Anomalia dell'Ereditarietà

Nozioni Generali e Caratteristiche.

Sincronizzazione e ereditarietà hanno caratteristiche conflittuali che ostacolano il loro uso simultaneo senza rompere l'encapsulation degli oggetti.

In sostanza: mantenere i vincoli di sincronizzazione dei metodi di una classe durante la definizione di una classe derivata impone la necessità di ridefinire i metodi stessi inficiando le potenzialità proprie dell'ereditarietà.

Storicamente, il danno creato dalla coesistenza di ereditarietà e concorrenza è considerato così grave da suggerire di rimuovere l'ereditarietà dai COOL.

Il termine **anomalia dell'ereditarietà** è stato coniato per referenziare i problemi legati alla coesistenza dell'ereditarietà e della concorrenza.

Anomalia dell'Ereditarietà

Nozioni Generali e Caratteristiche.

Vediamo nel dettaglio in cosa consiste l'anomalia dell'ereditarietà, allo scopo utilizzeremo l'esempio del buffer.

È universalmente riconosciuto che l'anomalia della ereditarietà non si presenta sempre nella stessa forma e che dipende dal meccanismo per la sincronizzazione fornito dal COOL.

Matsuoka e Yonezawa hanno stilato una tassonomia del problema classificando le occorrenze dell'anomalia in tre classi:

- sensibilità alla storia degli stati accettabili;
- partizionamento degli stati; e
- modifica degli stati accettabili

Anomalia dell'Ereditarietà
Sensibilità alla Storia degli Stati Accettabili.

Questa forma di anomalia si può manifestare quando i vincoli di sincronizzazione sono imposti sotto forma di guardie ai metodi.

```
class BoundedBuffer {
  void put(int v) when "not full" { ... }
  int get() when "not empty" { ... }
  ...
}
```

Guardia: espressione booleana che deve valere vero per poter attivare il metodo.

Estendo la classe aggiungendo gget() che non può essere attivato se il buffer è vuoto e dopo una chiamata a get().

```
class HistoryBuffer extends BoundedBuffer {
  void put(int v) when "not full" { after-get=false; ... }
  int get() when "not empty" { after-get=true; ... }
  int gget() when "not empty" and "not after-get" {...}
}
```

Ovviamente implica la ridefinizione dei metodi get() e put() per tenere traccia della storia delle attivazioni.

Anomalia dell'Ereditarietà
Modifica degli Stati Accettabili.

Un'altra manifestazione dell'anomalia dell'ereditarietà si ha quando si estende una classe con un metodo che influenza gli stati possibili e relativi messaggi accettabili di un oggetto.

Consideriamo ad es. la seguente interfaccia:

```
interface Lock {
  void lock(...);
  void unlock(...);
}
```

che introduce la possibilità di inibire la ricezione dei messaggi.

```
class LockedBoundedBuffer extends BoundedBuffer implements Lock {
  void put(int v) when "not full" and "not locked" { ... }
  int get() when "not empty" and "not locked" { ... }
  void lock() when "not locked" { locked = true; ... }
  void unlock when "locked" { locked = false; ... }
}
```


Anomalia dell'Ereditarietà
Partizionamento degli Stati Accettabili.

Insiemi dei Messaggi Accettabili

Un meccanismo alternativo consiste nell'associare i messaggi accettabili ai possibili stati di un oggetto e nell'esplicitare nel codice la transizione tra stati.

```
class BoundedBuffer {
      ...
  behavior: empty = {put}; full = {get}; partial = {get, put};
      ...
  void put(int v) {
    if (top == MAX-1) become full;
    else become partial;
      ...
  }
  int get() {
    if (top == 1) become empty;
    else become partial;
      ...
  }
}
```

Stato e insieme dei messaggi accettabili in quello stato.

become forza il passaggio da uno stato all'altro.

Anomalia dell'Ereditarietà
Partizionamento degli Stati Accettabili (Segue).

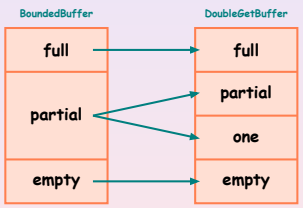
Si presenta l'anomalia quando si estende la classe introducendo nuovi stati, ad es., introducendo get2() che toglie due elementi dal buffer in un colpo solo.

```
class DoubleGetBuffer extends BoundedBuffer {
      ...
  behavior: empty = {put}; full = {get, get2};
            one = {get, put}; partial = {get, put, get2};
      ...
  void put(int v) {
    if (top == MAX-1) become full;
    else if (top == 0) become one;
    else become partial;
      ...
  }
  int get() {
    if (top == 1) become empty;
    else if (top == 2) become one;
    else become partial;
      ...
  }
```

```
  int get2() {
    if (top == 2) become empty;
    else become partial;
          ...
  }
}
```

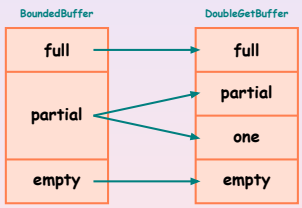

Anomalia dell'Ereditarietà
Specchietto Riassuntivo.

Partizionamento degli Stati

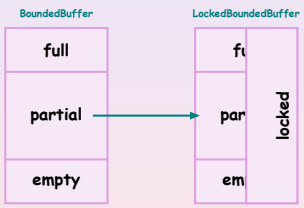


Anomalia dell'Ereditarietà
Specchietto Riassuntivo.

Partizionamento degli Stati



Modifica degli Stati



Anomalia dell'Ereditarietà
COOL di Nuova Generazione: Il Caso Java.

Le applicazioni possono essere multi-thread e le regioni critiche sono regolate da un monitor.

```
public class BoundedBuffer {
  protected int top = 0;
  protected final int MAX = 20;
  protected int buffer[] = new int[MAX];

  public synchronized void put(int n) ... {
    while (top >= MAX) wait();
    buffer[top++] = n;
    notifyAll();
  }
  public synchronized int get() ... {
    while (top <= 0) wait();
    int result = buffer[--top];
    notifyAll();
    return result;
  }
}
```

```
class HistoryBuffer extends BoundedBuffer {
  protected boolean afterGet = false;
  public synchronized void put(int n) ... {
    super.put(n);
    afterGet = false;
  }
  public synchronized int get() ... {
    int result = super.get();
    afterGet = true;
    return result;
  }
  public synchronized int gget() ... {
    while ((top<=0)||(afterGet)) wait();
    afterGet = false;
    return super.get();
  }
}
```

I monitor sono una variante delle guardie e soffrono dell'anomalia legata alla storia.

Anomalia dell'Ereditarietà
Soluzioni Proposte/Considerate.

Reflection.

Matsuoka e Yonezawa [2] hanno proposto l'uso della riflessione per incapsulare, e quindi coordinare, nel meta-livello i vincoli di sincronizzazione.

Aspect-Oriented Programming.

Tecniche più recenti[1] prevedono la separazione dei vincoli di sincronizzazione tramite la fattorizzazione in aspetti e la successiva combinazione tramite aspect-oriented programming.

[1] Ad es., C. Videira Lopes and K. J. Lieberherr. Abstracting Process-to-Function Relation in Concurrent Object-Oriented Application. In *Proceedings ECOOP'94*, LNCS 821:81-99. Springer, Dec. 1994.

Riferimenti Bibliografici

- **Mohamed E. Fayad and Rachid Guerraoui.**
  OO Distributed Programming Is Not Distributed OO Programming.
  Communications of the ACM, 42(4):101–104, April 1999.
- **Satoshi Matsuoka and Akinori Yonezawa.**
  Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Language.
  In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 109–126. MIT Press, 1993.
- **Giuseppe Milicia and Vladimiro Sassone.**
  The Inheritance Anomaly: Ten Years After.
  In *Proceedings of the 9th Annual ACM Symposium on Applied Computing (SAC'04)*, pages 1267–1274, Nicosia, Cyprus, on 14th-17th of March 2004. ACM Press.