Concurrent Object-Oriented Programming
Arnaud Bailly, Bertrand Meyer and Volkan Arslan

Lecture 5: Inheritance Anomaly.

In a Sequential World...

- A class is an implementation pattern.
- One is interested in functional behavior.
- A type contains operation signatures for a class.
- Objects offer uniform services.
In a Concurrent World...

- Classes are still implementation patterns.
- One is interested in interactive behavior:
  - the sequence of requests sent to an object,
  - the sequence of requests sent by an object.
- A type is a state machine.
- Objects offer naturally non-uniform services.

Inheritance and Subtyping

- Class B inherits (\(<\)) from Class A.
- Instances a of A, b of B.
- Re-usability of classes is substitution:
  - B subtype (\(<\)) of A meaning
  - any object c can use b as if it were a.
- Generally, in a sequential world:
  - B inherits from A implies B subtype of A.
  - B subtype of A implies B inherits from A.
- This really not true in the concurrent world!

Inheritance Anomaly

- Incremental inheritance (\(<\)):
  - \(B < A\) without redefining features of A.
- Inheritance anomaly:
  \[\exists \, B < A \land (\neg \exists C_1, \ldots, C_n. B < ^\ldots < C_i < \ldots C_n < A)\]
  - Depends on the notion of type and on the inheritance mechanism.
First Example

- Live routine as in POOL.
- Queue2 with a q2() method taking 2 elements.
- Queue2 is a subtype of Queue1!
- Queue2 has to redefine BODY.

```plaintext
CLASS Queue...
BODY DO
IF empty THEN ANSWER(empl)
ELSIF full THEN ANSWER(queue)
ELSE ANSWER ANY FIFO YDDB
END CLASS Queue...
```

Other Notations with Similar Problem

- When synchronization is:
  - interwoven (monitors, delay queues), or
  - isolated but not separable (path expressions).
- Can be further studied:
  - Behavior abstractions,
  - Enable sets,
  - Method guards.

Behavior Abstractions: The Good

```plaintext
class BUFFER LAST inherits BUFFER is
public interface ...
// added method Last
behavior:
empty  = renaming empty;
full   = (put, get, last) redefines full;
implementation:
  ... // inherited, not modified
end
```

```plaintext
... OBJECT: last () is ...
```

```plaintext
end BUFFER;
```
Behavior Abstractions: The Not-So-Good

```java
class BUFFER3 extends BUFFER is
  public interface: .. // as before
  behavior:
    empty_ = .. renames empty;
    one_ = .. (put, get) .. renames partial;
    fullL = .. (get, get2) .. renames full;
  implementation:
    Boolean isZero: // added to isFull, isFull
      put (OBJECT) is ..
        if (isFull) then become fullL;
        if one_ then become one_; else become partialL;
    end; // similar redefinition is necessary for get()
    Couple get2 () is .. // returns the two elements on top
      if (isFull) then become empty_;
      if one_ then become one_; else become partialL;
    end;
  end BUFFER3;
```

The previous anomaly...

is usually called:
the Partition Refinement Anomaly.

Method Guards: The Good

```java
class BUFFER is
  public interface: .. // as before
  guards:
    put; isFull()
  get; isEmpt()
  implementation:
    int in, out, buffsize;
    Boolean in = false; .. (put) ..
    Boolean empt() = .. (out) ..
    BUFFER (i: int) is size = i .. end;
    put (OBJECT) is .. in := in + 1; end;
    OBJECT get = .. out := out + 1; end;
  end BUFFER;

class BUFFER2 inherits BUFFER is
  guards: get(); plusOne()
  implementation:
    Boolean plusOne = .. in := out + 2; end;
    Couple get2 () is .. in := in + 2; end;
  end;
```
Method Guards: The Not-So-Good (1)

- Method `get()` may execute only after method `get()`.
- This is called **history-only sensitivity**.
- The guards are not re-defined but the bodies are.

```java
class BUFFER implements BUFFER is...
guards: get: (afterPut = false and not isEmpty())
implementation:
  Boolean afterPut := false;
  Object getObj() is .. out := out + 1; afterPut := false; end;
  // both put and get need re-definition!
  put: (Obj) is .. in := in + 1; afterPut := true; end;
  Object getObj() is .. in := in + 1; afterPut := false; end;
end;
```

Method Guards: The Not-So-Good (2)

```java
class LOCKER is...
guards: 
  lock: (not locked)
implementation:
  Boolean locked := false
  lock() is locked := true; end;
  unlock() is locked := false; end;
end;
class LOADED_BUFFER inherits BUFFER, LOCKER is...
guards: / need to redefine all the guards from BUFFER!
  put: (not locked and not isFull())
  get: (not locked and not isEmpty())
implementation:
  // nothing changes..
end;
```

The previous anomaly...

is usually called:

the **Modification of Acceptable States**.
Other Remedies

- Either:
  - Based on reflective mechanisms, or
  - Based on automatic parallelization.
- Isolate the code completely.
- Try to reduce (suppress) its total amount.
- When code is left, still poor on separability.
- Example: Enable Sets.

References


Next

- Formal models and logics for concurrency.