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

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 (\textless\textgreater) from Class A.
- Instances a of A, b of B.
- Re-usability of classes is substitution:
  - B subtype (\textless\textgreater) 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 (\textless\textgreater);
  - B \textless\textgreater A without redefining features of A.
- Inheritance anomaly:
  \[ \exists B \prec A \land (\neg \exists C_1, \ldots, C_n. B \prec_1 C_1 \prec \ldots \prec_1 C_n \prec A)) \]
- Depends on the notion of type and on the inheritance mechanism.
First Example

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

Other Notations with Similar Problem

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

Behavior Abstractions: The Good

```java
class QUEUE2 extends QUEUE1 {
  interface : public QUEUE1

  enqueue(x: T): E

  enqueue(x: T) {
    if (size < capacity) {
      queue[size++] = x;
      size += 1;
      return;
    }
    return;
  }
}
```

Behavior Abstractions: The Not-So-Good

```java
class QUEUE2 extends QUEUE1 {
  interface : public QUEUE1

  enqueue(x: T): E

  enqueue(x: T) {
    if (size < capacity) {
      queue[size++] = x;
      size += 1;
      return;
    }
    return;
  }
}
```

Method Guards: The Good

```java
class BUFFER is
  interface : public BUFFER2

  put(): E

  put(): E

  put(): E

  implement:
    size = size + 1;
    queue[size] = x;
    return;
end;
```

The previous anomaly...

is usually called:

the **Partition Refinement Anomaly**.
Method Guards: The Not-So-Good (1)

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

```java
class GGET BUFFER extends BUFFER {
    int count = 0;
    
guards: if (afterPut = false and not isEmpty())
    implementation: 
        if (count == 0) { // both put and get need re-definition!
            count = count + 1;
        } else { 
            // ... 
        }
    
    put(): count = count + 1; afterPut = true; 
    get(): count = count + 1; afterGet = true; 
}
```

- 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.