Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski
Lecture 6: Objects and concurrency
Outline

- Actors
- Active Objects
- Inheritance Anomaly
Refresher: towards OO

1. **Mutex, Semaphore**
   Mutual exclusion, no implicit scoping
   No condition synchronization

2. **CCR**
   Scoped mutual exclusion
   Condition synchronization (outside critical section)

3. **Monitors**
   Mutual exclusion, routine body as critical section
   Condition synchronization (inside critical section), wait/signal
   No atomicity guarantee

4. **Rendezvous, protected objects**
   Mutual exclusion (if necessary)
   Condition synchronization (outside critical section)
   Atomic execution of features
Can we merge objects and tasks?

- "Passive" (sequential) objects
  - React to incoming calls
  - Objects are **passive**
    - Features are executed by task(s)
    - No "internal" activity
  - Single task is implicitly created upon program start
    - Root creation procedure
    - **main**

- "Active" objects
  - Autonomous behavior
  - Own associated task
    - Possibly synchronizes with tasks calling object
Actors

- Well-described model (Hewitt 73, Agha 93)

- Originated from AI
  - Applied to functional programming first
  - Spawned a large family of concurrent object-oriented languages

- A computational entity that
  - has a unique name (an e-mail address)
  - has a set of potential behaviors
  - communicates through asynchronous messages
  - is reactive, i.e. it only responds to received messages

- Everything is an actor
  - Just like “everything is an object” but actors are inherently concurrent
Actors

- Actor’s behavior is deterministic
  - response to message is uniquely determined by message contents

- Basic actions on message reception
  - create a finite number of new actors (with fresh names)
  - send a finite number of messages
  - switch to a different behavior
    - specify a replacement which is essentially another actor that takes the place of the actor that creates it

- All actions performed on receiving a message done in parallel (no order imposed)
Actor model

1 | 2 | ... (mail queue) ... | n | n+1

create actors

\( x_n \)

specify replacement

\( x_{n+1} \)

\( y_1 \)

1 | 2 | ... (mail queue) ... | n | n+1
Evaluation

- Four no's
  - No sequentiaity
  - No buffering
  - No synchrony
  - No fixed topology

- Rather rudimentary mechanism
  - Explicit message passing not unified with object/feature model
  - No inheritance
Evaluation

- Asynchronous message passing
  - How to implement queries, i.e. invocations with replies?
    Especially with FIFO ordering?
  - Unbounded queues?

- How to reason about code in sequential manner?
  - Requires additional mechanism on top of basic actor paradigm
  - "Serializer" actor

- Used in several languages
  - ABCL family (Yonezawa)
  - Erlang
  - Io, Salsa (distribution)
  - JoCaml (join-calculus, functional)
"Actors are the real thing of which object-oriented programming is the caricature. When people say that OO programming is based on an 'intuitive' modelling of the world as objects in interaction, the intuition they invoke is the Actor model, but their actual model is a less-expressive version where only one object is active at a time, with a linear flow of control: objects are crippled Actors deprived of independent activity".

(Wikipedia)
Active objects

- In OO world, active objects have emerged
  - Largely inspired by actors
  - Can be viewed as mixture of tasks and protected objects in Ada’95

- An active object has an associated task (body)
  - Performs autonomously
  - May synchronize with other tasks through methods invoked on the object, cf. accept, or invoked on other objects
  - Thus represents “state machine”
Example in Synchronous Java

```java
public active class BoundedIntBuffer {
  private int[] a;
  private count = 0;
  private final int max;
  public BoundedIntBuffer(int max)
    { this.max = max; this.a = int[max]; } 
  public void put(int val)
    { a[count++] = val; }
  public int get()
    { return a[count--]; }
  public void run {
    for(;;)
    select {
      case
      when (count < max) accept put;
      case
      when (count > 0) accept get;
    }
  }
}
```
Active objects in Java

- Classes of active objects inherit from Thread
  - Implement `run()` method
  - `start()` method launches separate thread
  - Thread can synchronize with incoming calls through synchronization primitives
    - `wait()`, `notify()`, `notifyAll()`

- Tedious description of state machine
  - No Ada’95-like `select`
  - Variants of Java exist for that purpose, eg. Synchronous Java (EPFL)
Active vs passive objects

Different levels of passivity/activity:
- Simple passive objects (typically self-contained)
  - Handle a single call at a time
- Smarter passive objects
  - Can handle several calls simultaneously
  - Possibly communicate with objects acting in other tasks
  - Possibly create/spawn active objects/tasks

...  
- Simple active objects
  - Single associated task

...
Sequential objects

- A **class** is an implementation pattern
  - For **objects** of a kind
  - Modelling facility

- One is interested in **functional** behavior
  - Behavior
  - State

- A **type** contains operation descriptions for objects
  - Protocol for interacting with instances
  - Describes available features
Subtyping and inheritance

- Inheritance
  - Mechanism for decomposing functional behavior
    - Modularity
    - Extensibility
    - Conciseness
    - Reusability
    - ...

- Static typing
  - Used to ensure data type safety at compilation
  - Sufficient to guarantee safety in sequential world
Substitution principle

- Suppose class B inherits from class A

- Subtype substitution
  - B subtype of A means
    - Wherever instance of A is expected, an instance of B can be used
    - The behavior of instances of B complies with that expected by clients of A
    - B “adds” behavior, this is transparent to clients of A

- Generally, in a sequential world
  - B inherits from A  $\Rightarrow$  B subtype of A
  - B subtype of A  $\Rightarrow$  B inherits from A
Concurrent objects

- Classes are still implementation patterns

- Types still describe protocols for using objects

- Additional facet of behavior: *interaction*
  - The sequence of requests sent to an object
  - The sequence of requests sent by an object
  - “Protocol in time”
But

- Inheritance/subtyping usually includes addition and redefinition (overriding) of features
  - Possibly with reuse of (portions of) original features ("super-calls")

- What happens with "protocol in time"?
  - Functional and interactive behavior interfere
    - Preemption in middle of procedures
    - Guards for procedures

- Inheritance anomaly [Matsuoka&Yonezawa'93]
  - Supposedly "inherent" conflict between inheritance and synchronization
Inheritance anomaly

Loss of benefits of inheritance:

- Definition of subclass \( C' \) of \( C \) requires redefinitions of features in \( C \) and parents

- Modification of feature \( m \) in \( C \) requires modifications of seemingly unrelated features in parents (and descendants) of \( C \)

Furthermore (mainly for mixin inheritance):

- Definition of feature \( m \) forces other features to follow specific protocol (also in future subclasses)
Main variants

1. Partitioning of acceptable states
   - State refinement, e.g. with additional queries
   - e.g. get_2

2. History-only sensitiveness of acceptable states
   - State transitions change
   - e.g. get_after_put

3. Modification of acceptable states
   - Similar to above, states themselves/conditions change
   - e.g. locker
Partitioning of acceptable states

- Body as in POOL, Synchronous Java, ...
- Buffer_2 where get_2() returns 2 elements

```java
class Buffer {
    ...  
    public void run {
        for(;;)
            select {
                case  
                    when (empty)  
                        accept put;
                    case  
                        when (full)  
                            accept get;
                else  
                    accept _any_;  
            }
        }
    }
}

class Buffer_2 extends Buffer {
    ...  
    public void run {
        for(;;)
            select {
                case  
                    when (empty)  
                        accept put;
                    case  
                        when (full)  
                            accept get, get_2;
                else  
                    accept _any_;  
            }
        }
    }
```
class BUFFER is
public interface: ... // put and get
behavior:
    empty       = {put};
    partial     = {put, get};
    full        = {get};
implementation:
    Boolean isFull, isEmpty;
    put (t: OBJECT) is ... 
        if (isFull) then become full; else become partial;
    end;
    ... 
    OBJECT: get () is ... 
        if (isEmpty) then become empty;
        else become partial;
    end;
end BUFFER;
class BUFFER_LAST inherit BUFFER is
public interface: ... // added method last
behavior:
    //empty inherited from BUFFER
    partial = {put, get, last} redefines partial;
    full = {get, last} redefines full;
implementation:
    Boolean isFull, isEmpty;
    put (t: OBJECT) is ... // INHERITED, NOT MODIFIED
        if (isFull) then become full; else become partial;
    end;
    ... // get similarly
    OBJECT: last () is ... // returns the bottom of the stack
        if (isEmpty) then become empty;
        else become partial;
    end;
end BUFFER_LAST;
class BUFFER_2 inherit BUFFER is
public interface: ... // as before
behavior:
   empty = renames empty;
   one = {put, get};
   partial = {put, get, get2} redefines partial;
   full = {get, get2} redefines full;
implementation:
   Boolean isOne; // added to isEmpty, isFull
put (t: OBJECT) is ... 
   if (isFull) then become full;
   if (isOne) then become one; else become partial;
end;
... // similar redefinition is necessary for get()
Couple: get2 () is ... // returns two elements on top
   if (isEmpty) then become empty;
   if (isOne) then become one else become partial;
end;
end BUFFER2;
class BBUFFER is
public interface: ... // as before
guards:
    put: !isFull()
    get: !isEmpty()
implementation:
    int in, out, buf[size];
    Boolean isFull() is in = out + size end;
    Boolean isEmpty() is in = out end;
...
    put (t: OBJECT) is ... in := in + 1; end;
    OBJECT get is ... out := out + 1; end;
end BBUFFER;

class BBUFFER2 inherits BBUFFER is ...
guards: get2: moreThanOne()
implementation:
    Boolean moreThanOne() is in > out + 1; end;
    Pair get2() is ... in := in + 2; end;
end BBUFFER2;
Method \texttt{gget()} may execute only after method \texttt{put()}

- The guards are not re-defined but the bodies are

```plaintext
class GGET_BUFFER inherits BBUFFER is ...
guards:
gget: (afterPut = false and not isEmpty())
implementation:
  Boolean afterPut := false;
  Object gget() is ... out := out + 1; afterPut := false; end;

  // both put and get need redefinition!!!
  put(t: Object) is ... in := in + 1; afterPut := true; end;
  Object get() is ... in := in + 1; afterPut := false; end;
```

---

**History-only sensitiveness**

- Method \texttt{gget()} may execute only after method \texttt{put()}
- The guards are not re-defined but the bodies are

```plaintext
class GGET_BUFFER inherits BBUFFER is ...
guards:
gget: (afterPut = false and not isEmpty())
implementation:
  Boolean afterPut := false;
  Object gget() is ... out := out + 1; afterPut := false; end;

  // both put and get need redefinition!!!
  put(t: Object) is ... in := in + 1; afterPut := true; end;
  Object get() is ... in := in + 1; afterPut := false; end;
```
class LOCKER is ...  
guards:  
    lock: (not locked)  
    unlock: (locked)  
implementation:  
    Boolean locked := false;  
    lock() is locked = true; end;  
    unlock() is locked = false; end;  
end;

class LOCKED_BUF inherits BBUFFER, LOCKER is ...  
guards: // need to redefine all the guards from BBUFFER!!  
    put: (not locked and not isFull())  
    get: (not locked and not isEmpty())  
implementation:  
    ... // nothing changes...  
end;
Summary: inheritance anomaly

- Apparent conflict between synchronization and inheritance

- Depends on synchronization and inheritance mechanism considered
  - Guards, bodies, monitors, behaviors, ...
  - Single/multiple inheritance, mixin inheritance, traits, ...

- Certain mechanisms are more robust w.r.t. particular anomalies
  - e.g. guards and state refinement
public class IntBuffer {
    private int[] a;
    private int count = 0;
    private final int max;

    public IntBuffer(int max) {
        this.max = max; a = new int[max];
    }

    synchronized public put(int val) {
        while (!count < max)
            wait();
        a[count++] = val;
        notifyAll();
    }

    synchronized public int get() {
        while (!count > 0)
            wait();
        int val = a[count--];
        notifyAll();
        return val;
    }
}
Remember: “guards” are within methods

1. Partitioning of acceptable states
   - Sufficient to add `get2()`

2. History-only sensitiveness of acceptable states
   - Need to maintain a boolean, e.g., `lastOp`, for tracking last operation
   - Must accordingly update body in `put()`, `get()`

3. Modification of acceptable states
   - Suppose adding methods `lock()` and `unlock()`
   - Must update conditions in `put()`, `get()`
How about SCOOP?

Does it suffer from inheritance anomalies?