Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski
Lecture 5: Towards objects
Outline

- Refresher
- Conditional Critical Regions (CCRs)
- Rendezvous
- Actors and Active Objects
- Inheritance Anomaly
Refresher: mutex, semaphore

- **Mutex**: provides mutual exclusion
  
  ```
  m.lock
  -- critical section
  m.unlock
  ```

- **Semaphore**: generalization of mutex
  
  ```
  P(s)
  -- critical section
  V(s)
  ```
Monitors

- Mutual exclusion (enter ... exit)
- Condition synchronization (wait, signal)
  - Multiple condition variables
  - One wait queue per condition variable
  - FIFO servicing
  - Waiting tasks have priority over new ones
  - Signal-and-exit or signal-and-continue

```cpp
m.enter
if condition then cond_var.wait
cond_var.signal
m.exit
```
Refresher: properties of systems

- **Safety**
  - "Nothing bad ever happens"
  - e.g. mutual exclusion

- **Liveness**
  - "Eventually something good happens"
  - e.g. if a task is waiting to enter a monitor, it will eventually succeed
Refresher: scenarios

- Producers-consumer
- Readers-writers
- Dining philosophers
- Barrier synchronization
Conditional Critical Regions (CCRs)

- Provide **mutual exclusion** and **condition synchronization** at the same time

- Mutual exclusion w.r.t. a **named resource**
  
  ```
  with r do
      ...
  end
  ```

- Condition synchronization based on a **guard**
  
  ```
  with r when b do
      ...
  end
  ```
Pros and cons of CCRs

- Scoped mutual exclusion +
- Atomic reservation of several resources +
- Elegant and compact notation +

- Synchronization code present in routine bodies -
- Aliasing of resources not supported -
  
  ```
  with r do
  with s do
    ...
  end
  end
  ```

- Guard re-evaluation difficult to optimize -
Monitors (cf. Java) get closer to natural interaction between objects
  - Based on calls to custom procedures
    - With specific synchronization semantics
  - Yet, signals/conditions within procedures
    - Procedures as logical units of computation are not indivisible

rendezvous: concurrent tasks “meet”
  - By performing (completing) entries
    - Accepted at particular state of the callee
    - Upon prior fulfillment of conditions (guards)
What is a rendezvous?

a) real life (symmetric)

b) Ada (asymmetric)

requesting task

accepting task

Entry 1

Entry 2

Entry 3
task Int_Buffer is
  entry Put(I: in Integer);
  entry Get(I: out Integer);
end;

task body Int_Buffer is
  Val: Integer;
begin
  loop
    accept Put(I: in Integer) do
      Val := I;
      end Put;
    accept Get(I: out Integer) do
      I := Val;
      end Get;
  end loop;
end Int_Buffer;
Syntax

- **Header**
  - Lists externally callable entries (“procedures”)
    - Atomic units synchronizing caller and callee

- **Body**
  - Describes entries
  - Synchronization of calls to these, e.g.,
    - **Order** (accept ... end; ... accept ... end;)
    - **Alternatives** (select ... or ... ... end;)
    - **Guards** (when ... => accept ... end;)
    - Usually **loops are used**
Synchronization

- Synchronization of caller and callee on entry only
  - i.e. between accept and end
  - After end, caller is released

- Pattern
  - Release caller asap
  - Use entries mainly for synchronization and passing arguments
  - Instructions following accept are performed in mutual exclusion
Precise semantics

When calling task and accepting task agree on a rendezvous:

- **Calling task** passes *in* parameters and it blocks.

- **Accepting task** executes statements in *accept* body.

- **Out** parameters are passed back to calling task.

- rendezvous is complete and calling task is unblocked.
task Bounded_Int_Buffer is
  entry Put(I: in Integer);
  entry Get(I: out Integer);
end;

task body Bounded_Int_Buffer is
  A: Int_Array; X, Y: Index := 0;
  Count: Integer range 0..N := 0;
begin
  loop
    select
      when Count < N =>
        accept Put(I: in Integer) do
          A(X) := I;
          end Put;
          X := X + 1;
          Count := Count + 1;
      or
        when Count > 0 =>
          accept Get(I: out Integer) do
            I := A(Y);
            end Get;
            Y := Y + 1;
            Count := Count – 1;
          end select;
    end loop;
end Bounded_Int_Buffer;
Select statement

- Arbitrary number of guarded accept statements
  
  select
  
  when a => accept ... 
  or
  
  when b => accept ... 
  or
  
  ...  
  end

- Last alternative possibly
  
  - else
  
  - delay t
  
  - terminate
Semantics of `select`

1. Guards are evaluated, yielding set of open alternatives.

2. If there are calling tasks waiting on entry queues for open alternatives, start a rendezvous with one of them.

3. If all queues for open alternatives are empty, accepting task is suspended. As soon as some calling task requests an open alternative, a rendezvous with that task is started.

**IMPORTANT:** No open alternatives = catastrophe unless...
Semantics of **select**

4a. *(else)* If no open alternatives or no calling tasks for them, statements following *else* are executed.

4b. *(delay t)* Like three but accepting task is suspended for at most *t* seconds; if no rendezvous is possible after that, statements following *delay t* are executed.

4c. *(terminate)* If accepting task is suspended and all potential calling tasks have either completed execution or are also waiting on terminate, the entire set of tasks is terminated.
Tuning up

- Accepting side
  - Sequencing
  - Choice
  - Sleeping, timeouts
  - Termination

- Caller side can express alternatives
  - select
  - else
  - delay
Examples: accepting side

a) Timeout

```
task body T is
begin
  loop
    select
      accept Sensor_1 do ...;
    or
      accept Sensor_2 do ...;
    or
      ...
    or
      delay 0.1;
      Raise_Alarm;
    end select;
  end loop;
end T;
```

b) Polling

```
task body T is
begin
  loop
    select
      accept Sensor_1 do ...;
    or
      accept Sensor_2 do ...;
    or
      ...
    else
      -- usual business here
      end select;
  end loop;
end T;
```
c) Three-way rendezvous

```plaintext
task body T is
begin
  ...
  accept Synch_2 do -- caller: T2
    accept Synch_3; -- caller: T3
    end Synch_2;
  ...
end T;
```

d) Mutex

```plaintext
task body Mutex is
begin
  loop
    accept Wait;
    accept Signal;
    end loop;
end Mutex;
```
Examples: calling side

e) Polling at most one server

```plaintext
task body T is
begin
  loop
    select
      Server_1.E(...);
    else
      select
        Server_2.E(...);
    end select;
  end loop;
end T;
```

f) Polling multiple servers

```plaintext
task body T is
begin
  loop
    select
      Server_1.E(...);
    else
      null;
    end select;
    select
      Server_2.E(...);
    else
      null;
    end select;
    ... 
  end loop;
end T;
```
g) Timeout in calling task

```plaintext
task body T is
begin
  loop
    select
      Sensor.Sample (...);
    or
      delay 0.1;
      Notify_Operator;
    end select;
  end loop;
end T;
```
Evaluation

- Very flexible

- Coming close to an object/procedure-based programming style
  - Guards define what entries are eligible for execution at what point
  - Defined outside entries
    - State machine-like approach
However

- Very explicit notion and handling of tasks
  - Tasks call entries on each other
    - Entries are “limited” procedures
  - Objects vs tasks

- Support for rendezvous in Java
  - Threads are (active) objects: +
    - With behavior (run() / start())
  - Procedure bodies contain synchronization code: −
    - while(...) wait() pattern
The Ada’95 way

- Ada’95 has module-based type system, no classes as in Java

- **Specific protected types and objects**
  - Combination of monitors and rendezvous
  - Functions can only read data; no mutual exclusion
  - Procedures are mutually exclusive

- Inherent implementation of readers/writers
Entries and Guards

- An entry encompasses guard
  - cf. wait-conditions in SCOOP
  - Replace explicit signals/conditions within procedures

- Guards are reevaluated
  - Every time a procedure or entry terminates
  - Not after functions
  - Re-evaluation of guards takes precedence over “new” calls
    - Scheduling can be redefined
Example: Semaphore

protected type Semaphore
   (Start: Integer:= 1) is
      entry Secure;
      procedure Release;
   private
      Count: Integer:= Start;
   end Semaphore;

protected body Semaphore is
   entry Secure
      when Count > 0 is
         begin
            Count:= Count - 1;
         end Secure;

   procedure Release is
      begin
         Count:= Count + 1;
      end Release;
   end Semaphore;
Differences with (Basic) Java

- Java uses condition variables
  - Ada uses conditional wait (no `notify()`!)
- Java allows non-synchronized methods
  - Ada enforces synchronization among all entries
- Java has one waiting queue per object
  - Ada has one waiting queue per entry
- Java’s queues are unordered
  - Ada queues are FIFO
- In Java, which thread is `notify()`’ed is unknown
  - In Ada, it is the head of the queue
1. **Mutex**
   Simple abstraction for mutual exclusion

2. **Semaphore**
   More selective waiting, copies of critical resources

3. **CCR**
   Scoped mutual exclusion, condition synchronization

4. **Monitors**
   Custom procedures, inherent mutual exclusion
   Support for conditional waiting with signals

5. **rendezvous, protected objects**
   Procedures as indivisible code units
   Inherent support for conditional waiting
Conclusions

- Several “classic” problems, e.g.,
  - Producer/consumer
  - Readers/writers
- Several “classic” tools, e.g.,
  - Semaphores
  - Monitors
- May be more or less integrated with objects
  - Simple object model so far...
  - Threads/tasks are still handled explicitly
  - Distribution?
  - Reuse?
Merging Objects and Tasks

- Objects in sequential context
  - Answer to incoming calls, reactive behavior
  - Objects are passive
    - Caller tasks execute on called objects
  - Single task is implicitly created upon program start
    - Root creation procedure
    - main

- Active objects
  - Objects with 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

- An actor
  - 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
An 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

\[ x_n \]

Specify replacement

\[ x_{n+1} \]

Create actors

\[ y_1 \]

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

- Rather rudimentary mechanism
  - Explicit message passing not unified with object/methods model
  - No inheritance
- Asynchronous message passing
  - How to implement queries, i.e. invocations with replies? Especially with FIFO?
  - Unbounded queues?
- Sometimes one would like to reason about a larger code portion in sequential case
  - Requires mechanism on top if only asynchronous invocations
“Actors are the real thing of which object-oriented programming is the caricature. Actors are what Alan Kay had in mind but couldn't initially achieve when inventing the object-oriented paradigm. When people say that OOP is based on an 'intuitive' modelling of the world as objects in interaction, the intuition they invoke is the Actor model, but their actual OO 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. Retrofitting concurrency in an existing object-oriented system can help express the Actor paradigm, but it requires more than the superficial addition of threads to unleash the real thing.”
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 describes state machine of object
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)
public class BoundedIntBuffer {
    private int[] a;
    private int 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 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
  ...

Chair of Software Engineering
Concurrent Object-Oriented Programming
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/a class
  - Protocol for interacting with instances
  - Describes available features
Subtyping and Inheritance

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

- (Static) typing
  - Can be used to ensure data type safety at compilation
  - In sequential world can be sufficient to guarantee safety
Suppose class \( B \) inherits from class \( A \)

**Subtype substitution**

- \( B \) subtype of \( A \) means
  - Wherever instance of \( A \) is suspected, an instance of \( B \) can be provided
  - The behavior of instances of \( B \) complies to that expected by clients of \( A \)
  - \( B \) “adds” behavior, 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
- One is however also interested in *interactive* behavior
  - 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]
Inheritance Anomaly

Loss of benefits of inheritance:

- Definition of subclass $C'$ of $C$ requires redefinitions of methods in $C$ and parents

- Modification of a method $m$ in $C$ requires modifications to seemingly unrelated methods in parents and descendants of $C$

And furthermore (mainly for mixin inheritance):

- Definition of a method $m$ forces other methods 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, but states themselves/conditions change
   - e.g. locker
Illustration of 1

- Body as in POOL, Synchronous Java, ...
- Queue2 where \texttt{deq2()} method returns 2 elements
- Queue2 \textit{is a subtype of} Queue1
- Queue2 \textit{has to redefine body}

```
CLASS Queue1...
BODY DO
    IF empty THEN ANSWER(enq)
    ELSIF full THEN ANSWER(deq)
    ELSE ANSWER ANY FI OD YDOB

CLASS Queue2...
BODY DO
    IF empty THEN ANSWER (enq)
    IF one THEN ANSWER (deq)
    ELSIF full THEN ANSWER (deq, deq2)
    ELSE ANSWER ANY FI OD YDOB
```
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 inherits BUFFER is
public interface: ... // added method last behavior:
    empty_     = renames empty;
    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 BUFFER2 inherits 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 the 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;
  BBUFFER (s: int) is size = s 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: plusOne()
implementation:
  Boolean plusOne() is in >= out + 2; end;
  Couple get2() is ... in := in + 2; end;
end BBUFFER2;
- Method `gget()` may execute only after method `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 re-definition!!
  put(t: Object) is ...
    in := in + 1; afterPut := true; end;
  Object get() is ...
    in := in + 1; afterPut := false; end;
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 isEmpty())
   get: (not locked and not isEmpty())
implementation:
   ... // nothing changes...
end;
Summary

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

- Certain mechanisms are more indulgent towards certain 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;
    }
}

Evaluation

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 wait conditions in `put()`, `get()`