Concurrent Object-Oriented Programming
Lectures Overview

- Lecture 1: Introduction and Motivation
- Lecture 2: Classic Approaches to Concurrent Programming
- Lecture 3: Objects and Concurrency
- Lecture 4: From Concurrent to Distributed Programming
Lecture 3: Objects and Concurrency
Outline

- Actors and Active Objects
- Inheritance Anomaly
- Aspect-Oriented Programming
So Far

- Simple object model in examples
  - Roughly 2 levels: objects and (values of) primitive types

- Threads/tasks
  - Are still greatly independent from objects
  - Cf. tasks in Ada’95

- How about
  - Code reuse?
  - Distribution?
  - ...
procedure Buy_Meat is ...
end Buy_Meat;

procedure Shopping is
  task Get_Salad;

  task body Get_Salad is
    begin
      Buy_Salad;
    end Get_Salad;

  task Get_Wine;

  task body Get_Wine is
    begin
      Buy_Wine;
    end Get_Wine;

begin
  Buy_Meat;
end Shopping;

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;
Merging Objects and Tasks

- Objects in sequential context
  - Answer to incoming calls, reactive behavior
  - “Passive” objects
    - Caller tasks execute on called objects
  - Single task is implicitly created upon program start
    - Executes "main"
    - A.s.o.
- “Active” objects
  - Objects with autonomous behavior
  - Own associated task
    - Possibly synchronizes with tasks calling object
 Actors

- Well-described model
  - Underlying formalism, i.e., calculi
    - No shared data
  - Large influences

- Originated from AI
  - Applied to functional programming first
  - Later on object-oriented approaches
“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.”
Model Outline

- Independent entities
  - Executing concurrently
  - Autonomous behavior expressed through a “script”
- (Inter)acting asynchronously
  - Explicit asynchronous message passing
    - Communication channels FIFO and unlimited (Kahn networks, entities/nodes are tasks)
  - React to incoming messages
- Well-defined interfaces/types
  - Encapsulation
  - Every object is an actor, e.g., messages
Actor Model

1 | 2 | … (mail queue) … | n | n+1

create actors

\[ X_n \]

\[ X_{n+1} \]

\[ Y_1 \]

specify replacement
Actions

Aktor perform 3 types of actions:

1. An actor may send communications to specific actors it knows the mail address of
2. An actor may create new actors
3. An actor may specify a replacement which will accept the next communication (which is essentially another actor that takes the place of the actor that creates it, for the purpose of responding to certain communications)
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
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, cf. Synchronous Java, Jeeg
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 (single task runs)
- 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, i.e., application logic
  - 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
Substitution

- 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 $A$ $\implies$ $B$ subtype of $A$
  - $B$ subtype of $A$ $\implies$ $B$ 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
    - Cf. preemption in middle of procedures
    - E.g., 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

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

3. Modification of acceptable states
   - Similar to above, but states themselves/conditions change
Illustration of 1

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

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

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
END
```
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;
Method Guards

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;
Illustration of 2

- Method `gget()` may execute only after method `put()`.
- The guards are not re-defined but the bodies are.

```
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 isFull())
   get: (not locked and not isEmpty())
implementation:
   ... // nothing changes...
end; // nothing changes...
In 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
How about (standard) Java?

```java
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 $\text{get2()}$

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

3. Modification of acceptable states
   - Suppose adding methods $\text{lock()}$ and $\text{unlock()}$
   - Must update wait conditions, i.e. $\text{put()}, \text{get()}$
Aspect-Oriented Programming

- **Intuition**
  - Functional behavior, i.e., application logic, is main aspect of program
  - Interaction behavior is another one

- **Origins**
  - Reflective approach to concurrency
    - cf. [Briot et al.‘98]
  - With well-defined interfaces
    - More than only pre-/post-invocation actions
  - And an underlying methodology
By generalizing every program involves various aspects, e.g.
- Synchronization/concurrency
- Persistence
- Security
- Replication
- Logging
- Debugging
- ...

These “cut across” application: crosscutting
Crosscutting

- **Goal**
  - Aspects can be dealt with by programmer in isolation each
  - Can be “plugged” into application

- Aspects are rarely orthogonal
  - *Developer* must identify conflicts
    - **Crosscutting concerns**
    - “Aspects are well-modularized crosscutting concerns”
  - Aspects are *weaved*
Variations

- **Static** AOP
  - Similar to compile-time meta-object protocol
  - Aspects are defined like classes, types, etc.
  - Weaving occurs at compilation

- **Dynamic** AOP
  - New aspects can be added on the go
  - Weaving occurs at runtime
  - Challenge: consistency at transition
Freely available at [http://eclipse.org/aspectj]

Concepts
- **Pointcuts**
  - Describes *what triggers* an aspect
- **Advices**
  - Description of *what to perform* for pointcuts and *when*
- **Aspects**
  - Define *what* pointcuts and advices *apply to*
Syntax Overview

- **pointcut**
  - Refers to event, e.g., `set(<field>)`, `calls(<method>)`, `handler(<exception>)`, `target(<class>)`
  - Support for `*`, `||`, `&&`, `!`, `...`
  - Note: nested with `cflow(<pointcut>)`

- **Advice**
  - Refers to pointcut
  - E.g., `after(<args>)`, `before(<args>)`, `around(<args>)`

- **aspect**
  - Defines pointcuts and advices
  - Local variables, methods, `...`
aspect RegistryReaderWriterSynchronizing
  of pertarget(instanceof(readers()) || writers()) {

  // internal variables
  protected int activeReaders, activeWriters, waitingReaders, waitingWriters;

  // procedures
  protected synchronized void beforeRead() {
    ++waitingReaders;
    while (!(waitingWriters == 0 && activeWriters == 0)) {
      try { wait(); } catch (InterruptedException ex) {} 
    }
    --waitingReaders;
    ++activeReaders;
  }
  protected synchronized void afterRead() {...}
  protected synchronized void beforeWrite() {...}
  protected synchronized void afterWrite() {...}

  ...

...  

// pointcuts  
pointcut readers():  
calls(Vector Registry.elementsNear(int, int));
pointcut writers():  
calls(void Registry.add(FigureElement)) ||  
calls(void Registry.remove(FigureElement));

// advices  
before(): readers() { beforeRead(); }  
after(): readers() { afterRead(); }  
before(): writers() { beforeWrite(); }  
after(): writers() { afterWrite(); }
}
Evaluation

- Aspects and inheritance
  - Aspects can extend (inherit from) aspects
  - Thus possible to write subaspects for subclasses

- In practice?
  - Often more complex scenarios than in toy examples
  - Inheritance issues to be dealt with in aspects rather than in main code
  - Distribution (failures) are hard to handle automatically [Kienzle&Guerraoui’02]
  - ...