SISMA 2008/2009 - Seminar

simpA

An Agent-Oriented Approach for Prototyping Concurrent Applications on Top of Java

Alessandro Ricci

aliCE group at DEIS, Università di Bologna, Cesena

a.ricci@unibo.it

joint work with:
- Mirko Viroli
- Giulio Piancastelli, PhD student at DEIS

MOTIVATIONS

- Looking for new abstraction layers for programming and engineering complex software systems
  - concurrent, distributed
- Concurrency in particular
  - “Software Concurrency Revolution” [Sutter,Larus (Microsoft) - ACM QUEUE 3(7) 2005]
    - Concurrency as important aspect in mainstream programming and software engineering
    - Pushing technologies
      - Multi-core architectures, Internet, ..., etc

> Beyond fine-grained OS-based mechanisms

- beyond processes, threads, synchronized blocks, semaphores, futures, call-backs, ...
- [Sutter, Larus]:*“...What we need is OO for concurrency - higher-level abstractions that help build concurrent programs, just as object-oriented abstractions help build large componentized programs..."*

SOME RELATED

- OOCP research (80s / 90s in particular)
  - actors and actor-like approaches
  - active objects
  - ...
- State of the art
  - Polyphonic C#, JR, JAC, ...
  - Scala (+ actors)
  - Erlang (-> process & msg passing)
  - ...
  - > most of them basically extends the basic OO model
- java.util.concurrent library (JDK 5.0)
  - very efficient and flexible low-level mechanisms
  - patterns

OUR CONTRIBUTION

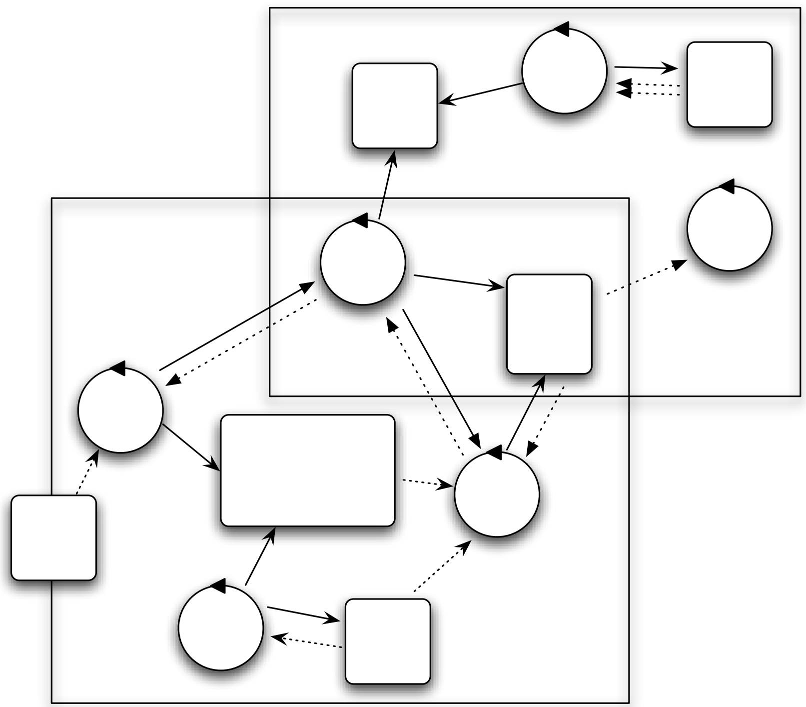
- A&A (Agents and Artifacts)
  - novel conceptual / programming model
  - introducing a new abstraction layer based on *agent-oriented* abstractions
- simpA
  - Java extension supporting A&A
- simpAL (ongoing work)
  - full-fledged language and VM implementing A&A

AGENDA

- Motivations and Background
- A&A programming model
- simpA framework

A&A BASIC ABSTRACTIONS

- Inspiration from **Activity Theory** and human working environments
  - human actors doing activities in shared context, cooperating by msg passing and sharing and using artifacts (resources, tools,...)
- Applications as **workspaces** composed by **agents** and **artifacts**
  - agents ~ human actors
  - artifacts ~ artifacts used by humans
  - workspaces ~ shared environments

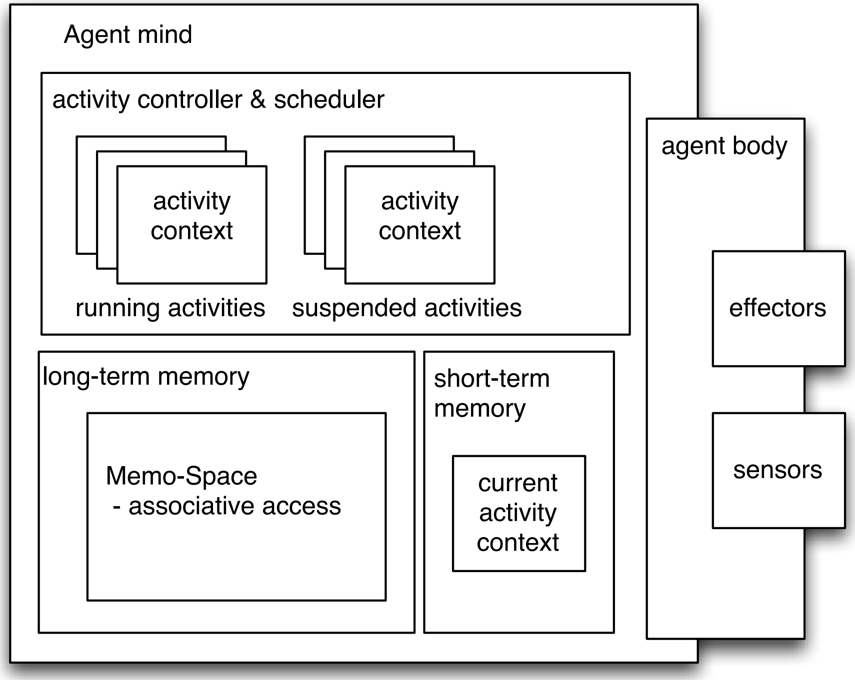


THE “AGENT” ABSTRACTION

- Pro-active entities in the workspace
  - designed to encapsulate the logic and control of activities
    - *action* as basic computational step
    - *activities* as composition of actions
  - Strong encapsulation
    - state + (active) behaviour + control of the behaviour
    - agents have no interfaces (!)
- Interacting with artifacts
  - observation and use
- Interacting with other agents
  - exchanging messages

AN AGENT ABSTRACT MODEL

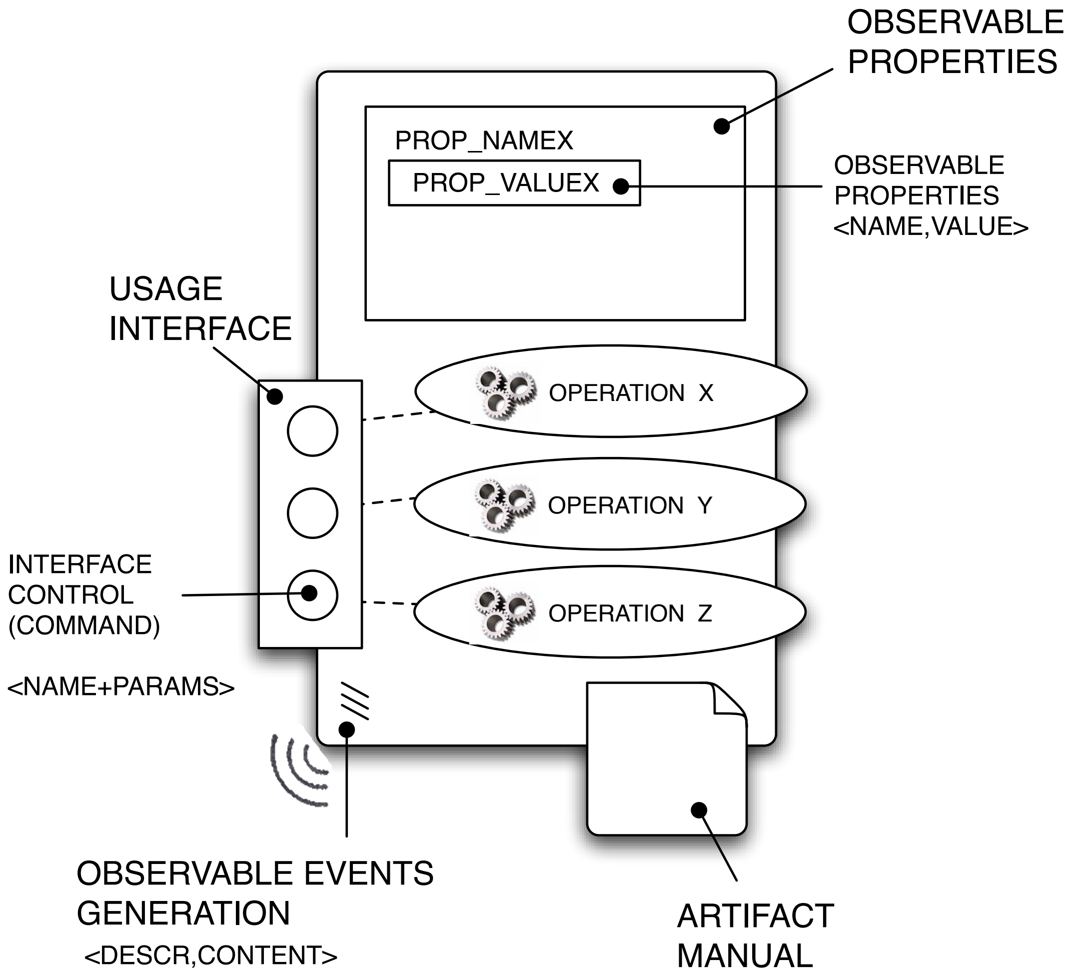
- Hierarchical model of activities
  - agents as scheduler, executors, controllers of activities
  - activity agenda specified by programmers
    - interpreted and executed by agents
- Long-term memory for doing activity
  - associative access
  - + short term memory contextualised to individual actitivities (activity context)
- Sensor space
  - sensors where to collect stimuli from the environment



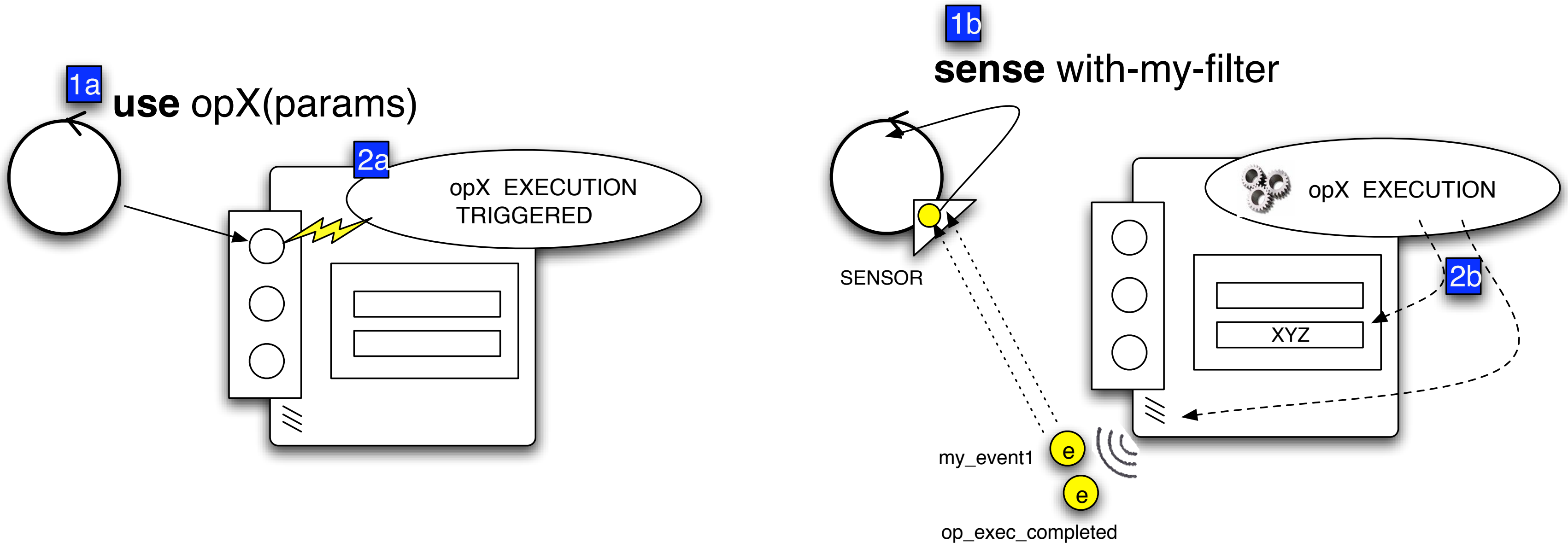
THE "ARTIFACT" ABSTRACTION

- Passive, *function*-oriented abstraction
  - designed to encapsulate some kind of *function*
    - the intended purpose of the artifact
  - functionality structured in terms of *operations*
  - instantiated, shared and used by agents to support their activities
- Basic kinds
  - resources
    - a dbase, a counter, a GUI interface, a printer,...
  - tools
    - a blackboard, a map, a channel, a synchronizer, ...

AN ARTIFACT ABSTRACT MODEL



AGENT-ARTIFACT INTERACTION: USE & OBSERVATION



- No control coupling
  - ...operations are not methods...

A&A FOR DESIGNING CONCURRENT SYSTEMS

- Decomposing a system in terms of workspaces with agents and artifacts as basic building blocks
  - static & dynamic decomposition
- Agents execute their activities concurrently
  - hierarchical activity model to structure complex activities
- Agents interact and coordinate by means of (1) using shared artifacts (2) directly communicating

simpA

- A Java-based framework to develop programs based on A&A abstraction layer
  - realised as a library
    - compiled and executed on top of a standard Java platform
  - exploiting Java 5 annotation
- Simplicity and minimality
  - minimizing the number of classes needed to define agents and artifacts
- Open-source project
  - http://www.alice.unibo.it/simpa

DEFINING AGENTS

- Single class extending `alice.simpa.Agent`
- Specifying activities
  - atomic: **`@ACTIVITY`** methods
    - sequence of statements and *actions*
      - internal actions
      - external actions
  - structured: **`@ACTIVITY_WITH_AGENDA`** methods
    - hierarchically composed by sub-activities described in activity *agenda*
- Agent behaviour
  - activity execution, following the agenda
  - `main` as default starting activity

NAIVE EXAMPLE

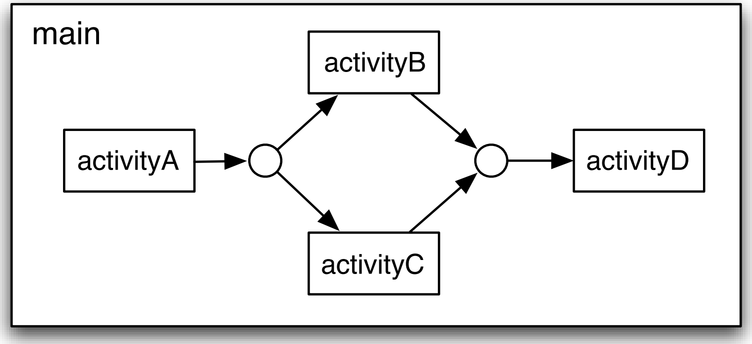
```
public class HelloAgent extends Agent {

  @ACTIVITY void main(){
    ArtifactId id = lookupArtifact("console");
    use(id,new Op("print","Hello, world!"));
  }

}
```

SPECIFYING STRUCTURED ACTIVITIES

- Activity agenda description
  - declaration of sub-activities to-do
- TODO description: **`@TODO`** annotation
  - specifying activity name + pre-condition + attributes
    - as soon as the precondition holds, the activity is executed
    - > multiple activities can be executed in parallel
- Pre-conditions
  - boolean expressions over the agent state
  - events occurred, agent knowledge

EXAMPLE

```
public class MyAgent extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="activityA"),
    @TODO(activity="activityB", pre="completed(activityA)"),
    @TODO(activity="activityC", pre="completed(activityA)"),
    @TODO(activity="activityD",
          pre="completed(activityB),completed(activityC)")
  }) void main(){}

  @ACTIVITY void activityA(){...}
  @ACTIVITY void activityB(){...}
  @ACTIVITY void activityC(){...}
  @ACTIVITY void activityD(){...}
}
```



AGENT MEMORY: MEMOs

- Long-term memory organized as a *memo-space*
  - associative store ~ blackboard with *memos*
  - internal actions to create, associatively access, read memos
- Memo data structure
  - flat labelled tuples of data-objects and values
    - can be partially specified (-> with variables)
- Memo usage
  - storing information useful or result of agent work
  - coordinating activities
    - `memo` predicate in TODO precondition

MEMO EXAMPLE

```
public class MyAgent extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="activityA"),
    @TODO(activity="activityB", pre="completed(activityA)")
    @TODO(activity="activityC", pre="completed(activityA)"),
    @TODO(activity="activityD",
          pre="completed(activityB),completed(activityC)")
  }) void main(){}

  @ACTIVITY void activityA(){
    memo("x",1);    // attach a new memo x(1)
  }

  @ACTIVITY void activityB(){
    int v = getMemo("x").intValue(0); // read 0-th memo argument
    memo("y", v+1, null);  // attach a new memo y(2,_)
  }

  @ACTIVITY void activityC(){
    memo("z", getMemo("x").intValue(0)*5);
  }

  @ACTIVITY void activityD(){
    int z = getMemo("z").intValue(0);
    int w = z*y0.intValue();
    log("the result is: "+w);
  }
}
```

MEMO EXAMPLE 2

```
public class MyAgent extends Agent {

 @ACTIVITY_WITH_AGENDA({
    @TODO(activity="activityA"),
    @TODO(activity="activityB", pre="memo(x(_))"),
    @TODO(activity="activityC", pre="memo(x(1))"),
    @TODO(activity="activityD", pre="memo(y(_,_)),memo(z(_))")
  }) void main(){}

  @ACTIVITY void activityA(){
    memo("x",1);    // attach a new memo x(1)
  }

  @ACTIVITY void activityB(){
    int v = getMemo("x").intValue(0); // read 0-th memo argument
    memo("y", v+1, null);  // attach a new memo y(2,_)
  }

  @ACTIVITY void activityC(){
    memo("z", getMemo("x").intValue(0)*5);
  }

  @ACTIVITY void activityD(){
    int z = getMemo("z").intValue(0);
    int w = z*y0.intValue();
    log("the result is: "+w);
  }
}
```

IMPLEMENTING CYCLIC ACTIVITIES

- Cyclic / non-terminating activities is quite common when programming agents

```
while (true){
  ...
}
```

...considered harmful

- in simpA: *persistent* todo
  - `todos` re-inserted in the agenda as soon as the activity has completed

EXAMPLE

```
public class MyAgent extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="preparing"),
    @TODO(activity="processing", persistent=true,
          pre="completed(preparing), memo(ntasks_done(X)),X<100")
  }) void main(){}

  @ACTIVITY void preparing(){...}

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="getTaskTodo"),
    @TODO(activity="doTask", pre="task_todo(_)")
  }) void processing(){}

  @ACTIVITY void getTaskTodo(){
   // <get a new task todo>
    memo("task_todo",taskInfo);
  }
  @ACTIVITY void doTask(){
    Memo m = delMemo("task_todo");
    // <do task>
  }
}
```

DEFINING ARTIFACTS

- Single class extending `alice.simpa.Artifact`
- Specifying the operations
  - atomic: **`@OPERATION`** methods
    - name+params -> usage interface control
    - no return value
  - structured
    - linear composition of atomic operation steps composed dynamically
  - `init` operation
    - automatically executed when the artifact is created
- Specifying artifact state
  - instance fields of the class

NAIVE EXAMPLE

```
public class Count extends Artifact {
  int count;

  @OPERATION void init(){
    count = 0;
  }

  @OPERATION void inc(){
     count++;
  }
}
```

ARTIFACT OBSERVABLE EVENTS

- Observable events
  - generated by `signal` primitive
  - represented as labelled tuples
    - event_name(Arg0,Arg1,...)

- Automatically made observable to...
  - the agent who executed the operation
  - all the agents observing the artifact

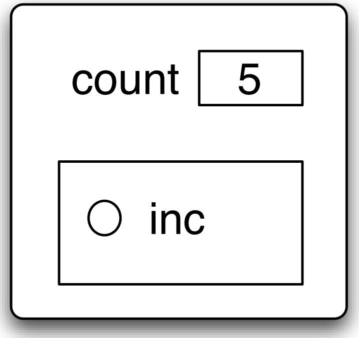
EXAMPLE

```
public class Count extends Artifact {
  int count;

  @OPERATION void init(){
    count = 0;
  }

  @OPERATION void inc(){
     count++;
     signal("new_count_value", count);
   }
}
```

ARTIFACT OBSERVABLE PROPERTIES

- Observable properties
  - declared by **defineObsProperty** primitive
    - characterized by a property name and a property value
  - internal primitives to read / update property value
    - **updateObsProperty**
    - **getObsProperty**
- Automatically made observable to all the agents observing the artifact

EXAMPLE



OBSERVABLE PROPERTIES:

count: int

USAGE INTERFACE:

inc: [op_exec_completed]

```
public class Count extends Artifact {

  @OPERATION void init(){
    defineObsProperty("count", 0);
  }

  @OPERATION void inc(){
     int count = getObsProperty("count");
     updateObsProperty("count", count + 1);
   }
}
```

MORE ON ARTIFACTS

- Structured operations
  - specifying operations composed by chains of atomic operation steps
  - to support the concurrent execution of multiple operations on the same artifact
    - by interleaving steps
- Linkability
  - dynamically composing / linking multiple artifacts together
- Artifact manual
  - document containing a formal description of artifact functionality and operating instructions
    - open systems
    - toward ‘intelligent’ use of artifacts

AGENT-ARTIFACT INTERACTION

- Basic actions available to agents for interacting with artifacts
  - use
    - to use an artifact through its usage interface, triggering the execution of operation

```
use(what:Artifact, op:Operation{,sid:SensorId}{,timeout:long}):OpId
```

  - sense
    - to retrieve events collected by sensors

```
sense(sid:SensorId{,filter:String}{,timeout:long}):Perception
```

  - focus
    - to start / stop a continuous observation of an artifact

```
focus(what:Artifact,sid:SensorId)
stopFocusing(what:Artifact)
```

ARTIFACT INSTANTIATION & LOOKUP

- using "factory" artifacts
  - providing functionalities to instantiate dynamically artifacts and agents
  - one for each workspace
  - agent auxiliary action: **`makeArtifact`**
    - encapsulating the access to factory artifacts
- using "registry" artifacts
  - providing functionalities to lookup dynamically artifacts and agents
  - one for each workspace
  - agent auxiliary action: **`lookupArtifact`**
    - encapsulating the access to registry artifacts

AN EXAMPLE

```
public class CountUser extends Agent {
  @ACTIVITY void main() {

    SensorId sid = linkDefaultSensor();
    ArtifactId countId = makeArtifact("myCount","Count");

    use(countId,new Op("inc"));

    use(countId,new Op("inc"),sid);

    try {
      Perception p = sense(sid,"new_count_value",1000);
      long value =  p.getContent(0).longValue;

      ArtifactId dbaseId = lookupArtifact("myArchive");
      focus(dbaseId,sid);
      use(dbaseId, new Op("write",new DBRecord(value));

    } catch (NoPerceptionException ex){
      log("No count_value perception from the count");
    }
  }
}
```

APPLICATION MODEL

- An application is defined by a workspace + one main (boot) agent
  - default artifacts
    - registry, factory, security-registry, etc.
- Application launcher
  - specifying the workspace name + boot agent

```
public class HelloWorld {
    public static void main(String[] args) throws Exception {
        SIMPALauncher.launchApplication("hello-world-app",
                              "basic.HelloAgent","Michelangelo”);
    }
}
```

ADVANCED ISSUES

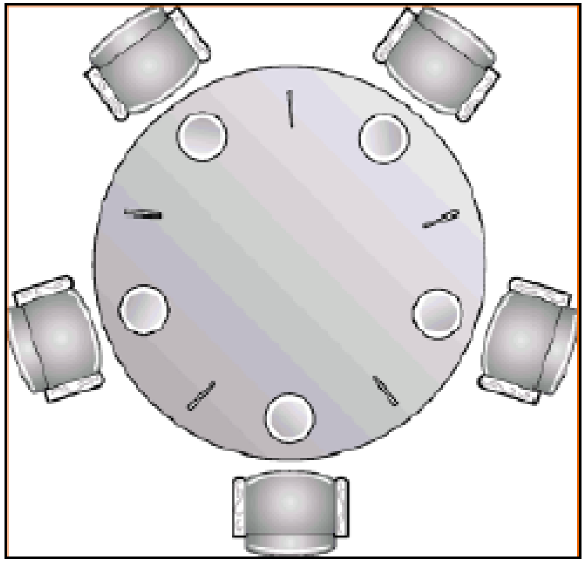
- Openness
  - agents can dynamically join and quit workspaces
  - RBAC model for ruling agent access & use of artifacts
    - security-registry artifact to keep track of roles and role policies
- Distribution
  - agents can join and work concurrently on multiple workspaces..
  - ..distributed over multiple simpA nodes

COMPLETE EXAMPLES

- Well-known examples in concurrent programming
  - Dining Philosophers
    - philosopher agents using a table as coordination artifact
  - Producers-Consumers
    - producers and consumers agents sharing and using a bounded buffer artifact
  - Readers-Writers
    - readers and writers agents sharing and using a dbase artifact providing locking functionalities
  - ...
- Implementation available in simpA distribution

“HELLO PHILOSOPHERS” EXAMPLE

- Dijkstra well-known problem about cooperative processes coordination
  - 5 philosophers thinking and eating rice at the same table, sharing 5 chopsticks
  - coordination to share chopsticks & avoid deadlock
  - kind of “hello world” for concurrent programming
- Rethinking the problem in simpA
  - restaurant as a workspace
  - philosophers + waiter as agents
  - a table as a coordination artifact



THE TABLE ARTIFACT

```
public class Table extends Artifact {

  boolean[] chops;

  @OPERATION void init(int nchops){
    chops = new boolean[nchops];
    for (int i = 0; i < chops.length; i++){
      chops[i]=true;
    }
  }

  @OPERATION(guard ="chopsAvailable") void getChops(int firstChop, int secondChop){
    chops[firstChop] = chops[secondChop] = false;
    signal("chops_acquired");
  }

  @GUARD boolean chopsAvailable(int firstChop,int secondChop){
    return chops[firstChop] && chops[secondChop];
  }

  @OPERATION void releaseChops(int firstChop, int secondChop){
    chops[firstChop] = chops[secondChop] = true;
  }
}
```

Usage interface:
- **getChops**
- **releaseChops**

Observable events generateds
- **chops_acquired**

PHILOSOPHER AGENT

```
public class Philosopher extends Agent {

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="init"),
    @TODO(activity="living", pre="completed(init),!memo(starved)", persistent=true),
  }) void  main(){}

  @ACTIVITY void init() {
     memo("hungry");
  }

  @ACTIVITY_WITH_AGENDA({
    @TODO(activity="eating", pre="memo(hungry)"),
    @TODO(activity="thinking", pre="completed(eating)"),
  }) void living(){}

  @ACTIVITY void eating(){
    ArtifactId tableId = lookupArtifact("table");
     SensorId sid = linkDefaultSensor();
     use(tableId, new Op("getChops", MYLEFTCHOP_ID, MYRIGHTCHOP_ID), sid);
     try {
      sense(sid,"chops_acquired",5000);
      // eat
      use(tableId, new Op("releaseChops", MYLEFTCHOP_ID, MYRIGHTCHOP_ID));
      removeMemo("hungry");
    } catch (NoPerceptionException ex){
      memo(“starved”);
    }
  }

  @ACTIVITY void thinking(){
    // think
    memo("hungry");
  }
}
```

CONCLUDING REMARKS

- First-class abstractions for active and passive entities
  - a solution to the active & passive object issue
  - strong encapsulation
- Bridging the gap between design & implementation
  - A&A as a simple and intuitive way to decompose a system
  - simpA as a first simple implementation framework
- Orthogonality with respect to OO
  - OO used for ADTs
  - using pure Java without concurrency mechanisms

AVAILABLE THESES

- Extending the basic simpA model
  - integrating AI techniques on top of activities and agenda
  - exploiting tuProlog
- Exploring new agent-oriented languages
  - integrating main strenghts of simpA & Jason
- Applications
  - applying simpA for SOA/WS, Autonomic Computing, Virtualization systems