# Lecture 9: Practical BDI Programming

## Autonomous Agents and Multiagent Systems
## DIS, La Sapienza - PhD Course

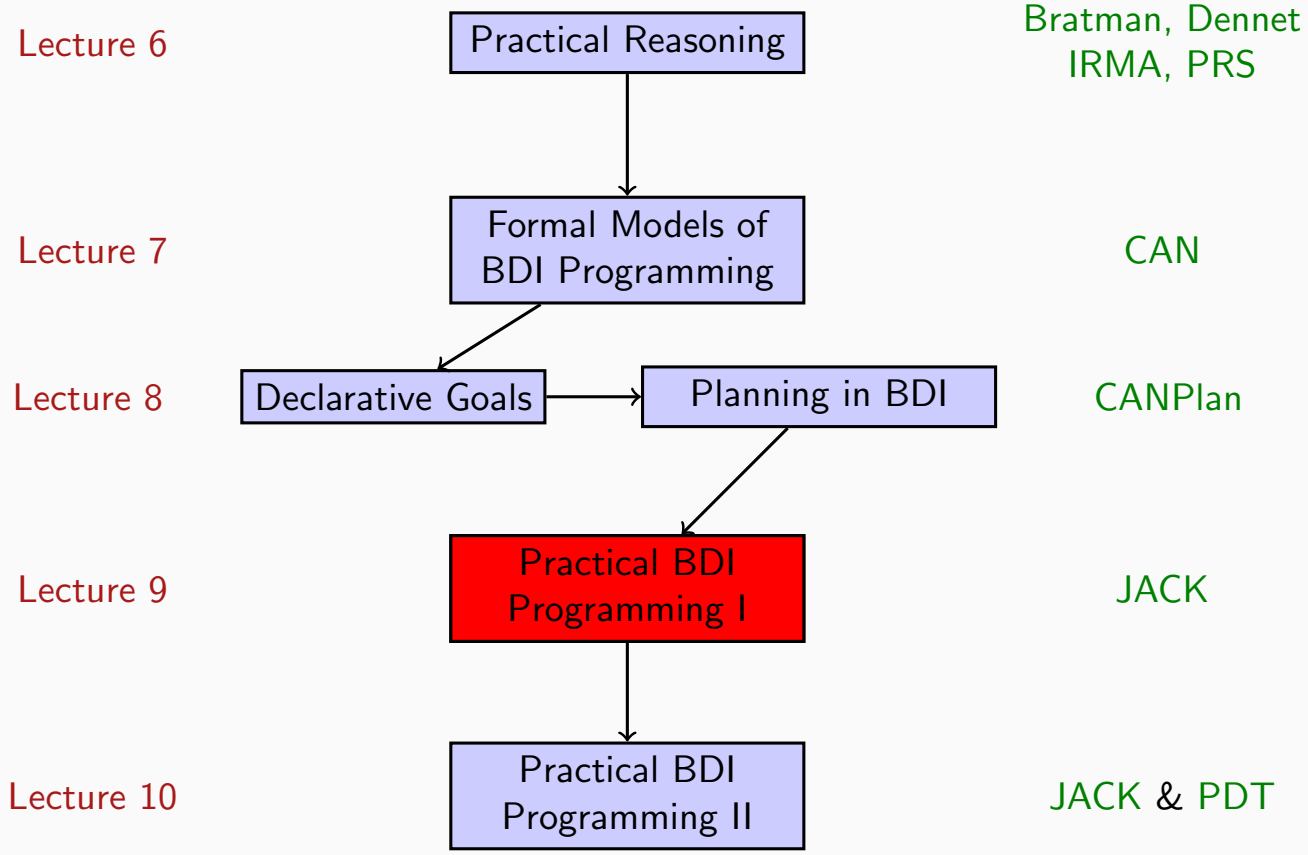
Sebastian Sardina[1]

[1]Department of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA


December 5, 2007



# Roadmap for Next Lectures



## Last Lecture Review

1. Described declarative goals in CAN:
   - motivated from agent-theory;
   - hybrid account of goals: declarative & procedural aspects;
   - incorporated new construct Goal($\phi_s, P, \phi_f$) to the language.
2. Showed how accommodate on-demand planning in CAN:
   - motivated the need for planning in BDI systems;
   - reviewed HTN-planning: off-line decomposition of tasks;
   - incorporated new construct Plan($P$) to the language;



## This Lecture

# JACK: A JAVA-based BDI System

1. Captures the basic notions of BDI programming.
2. Is just an extension of Java.
3. Main components: agents, beliefsets, events, plans, capabilities, etc.
4. Special syntax style.

# Outline







# Some BDI Agent-oriented Programming Languages

Some formal BDI programming languages:

1. AgentSpeak: first formal BDI language.
2. 3APL: language with different kind of plan rules.
3. GOAL: based on declarative goals only.
4. CAN(Plan): failure-handling, declarative goals, and planning. $\surd$

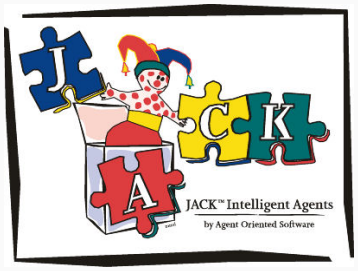
Some BDI programming language systems/platforms/architectures:

1. PRS and dMars: first BDI-based systems; C++ based.
2. JAM: a hybrid extension of PRS.
3. JASON: JAVA-based implementation of AgentSpeak.
4. JADEX: based on JADE communication platform.
5. SPARK: SRI's BDI system.
6. JACK: powerful commercial JAVA-based BDI-system. $\Longleftarrow$ NEXT!

# What is JACK?

From Agent Oriented Software (AOS) web's page:

*JACK is an environment for building, running and integrating commercial-grade multi-agent systems using a component-based approach. JACK is based upon the company's Research and Development work on software agent technologies.*



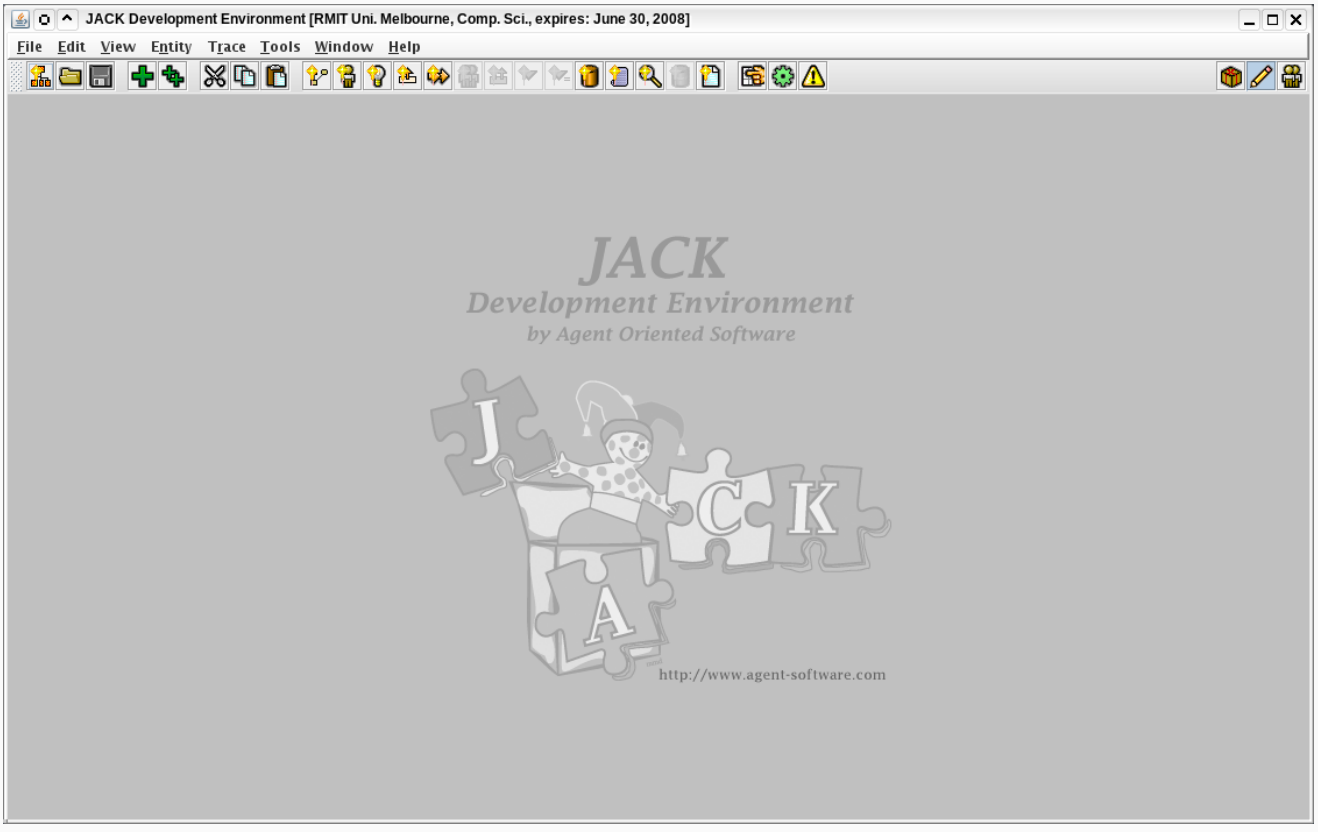


`http://www.agent-software.com`





# JACK Development Environment (JDE)



# But, what is JACK?

From JACK's manual:

> *JACK Intelligent Agents (JACK) is an Agent Oriented development environment built on top of and integrated with the Java programming language. It includes all components of the Java development environment as well as offering specific extensions to implement agent behavior.*





# JACK and JAVA

> *JACK's relationship to Java is analogous to the relationship between the C++ and C languages. C was developed as a procedural language and subsequently C++ was developed to provide programmers with object-oriented extensions to the existing language. Similarly, JACK has been developed to provide agent-oriented extensions to the Java programming language. JACK source code is first compiled into regular Java code before being executed.*
>
> *In the same way that object-oriented programming introduces a number of key concepts that influence the entire logical and physical structure of the resulting software system, so too does agent-oriented programming. In agent-oriented programming, a system is modelled in terms of agents. These agents are autonomous reasoning entities capable of making pro-active decisions while reacting to events in their environment.*

# The JACK BDI Programming Language

## 1 JACK Agent Language

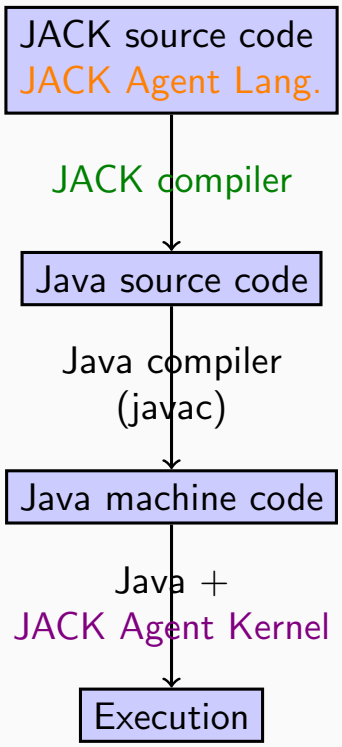
- Used to describe an agent-oriented software system.
- Super-set of Java (agent-oriented features extensions).

## 2 The JACK Agent Compiler

- Converts JACK Agent Language into pure Java.
- This Java source code can then be compiled into Java virtual machine code.

## 3 The JACK Agent Kernel

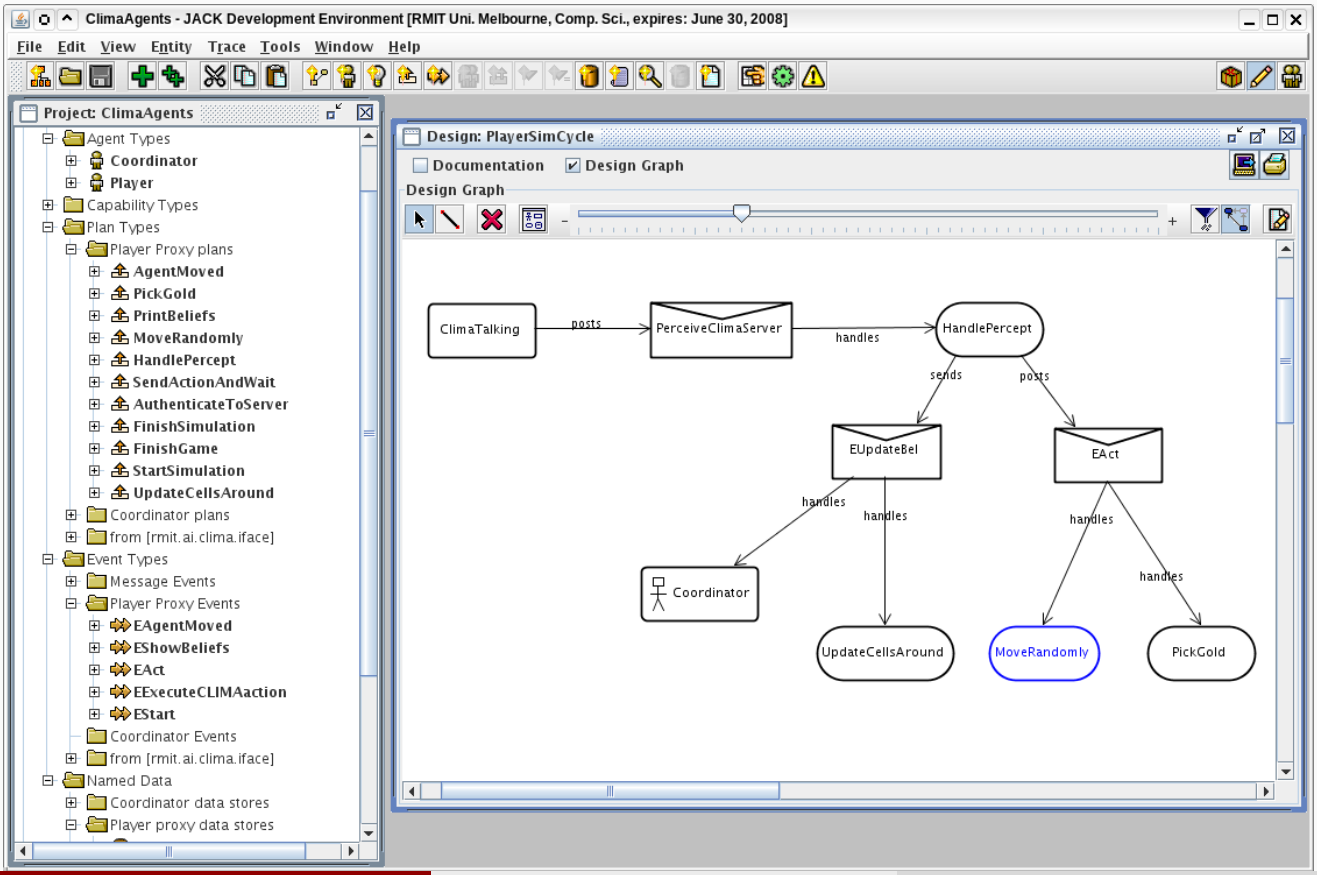
- Runtime engine for programs written in the JACK Agent Language.
- Set of classes that give JACK Agent Language programs their agent-oriented functionality.
- Run behind the scenes.
- Implement the underlying infrastructure and functionality for agents.








# JACK Development Environment (JDE)



# Applications

1. Autonomous UAVs (Unmanned Aerial Vehicles).
2. Information management.
   - Oil Trading and Operations
   - Reactive ISR Information Broker (ISR Broker) for the onboard interfacing to aircraft mission systems.
3. Military simulation systems (Rules of Engagement Evaluation Environment).
4. Human behavior representation.
5. Decision support in critical operational situations.
6. Smart user interface management.
7. RoboCup (robotic soccer).
8. CLIMA Multi-agent competition.





# Low Resource Requirements and Scalability

- Hundreds of agents running on low-end hardware.
- Agent creation $> 1,000$ per second.
- Agent destruction $> 100,000$ per second.
- Message to another agent (in same process) $100,000$ per second
- JACK agents can be:
  1. all concentrated in a single operating system process;
  2. scattered over a multi-platform network;
  3. any combination of the previous.

High performance component of larger system
... in keeping with design philosophy

# The Core Concepts of JACK

JACK extends Java with agent-oriented concepts, such as:

- Agents.
- Capabilities.
- Events.
- Messages.
- Plans.
- Agent Knowledge Bases (Databases).
- Resource and Concurrency Management.

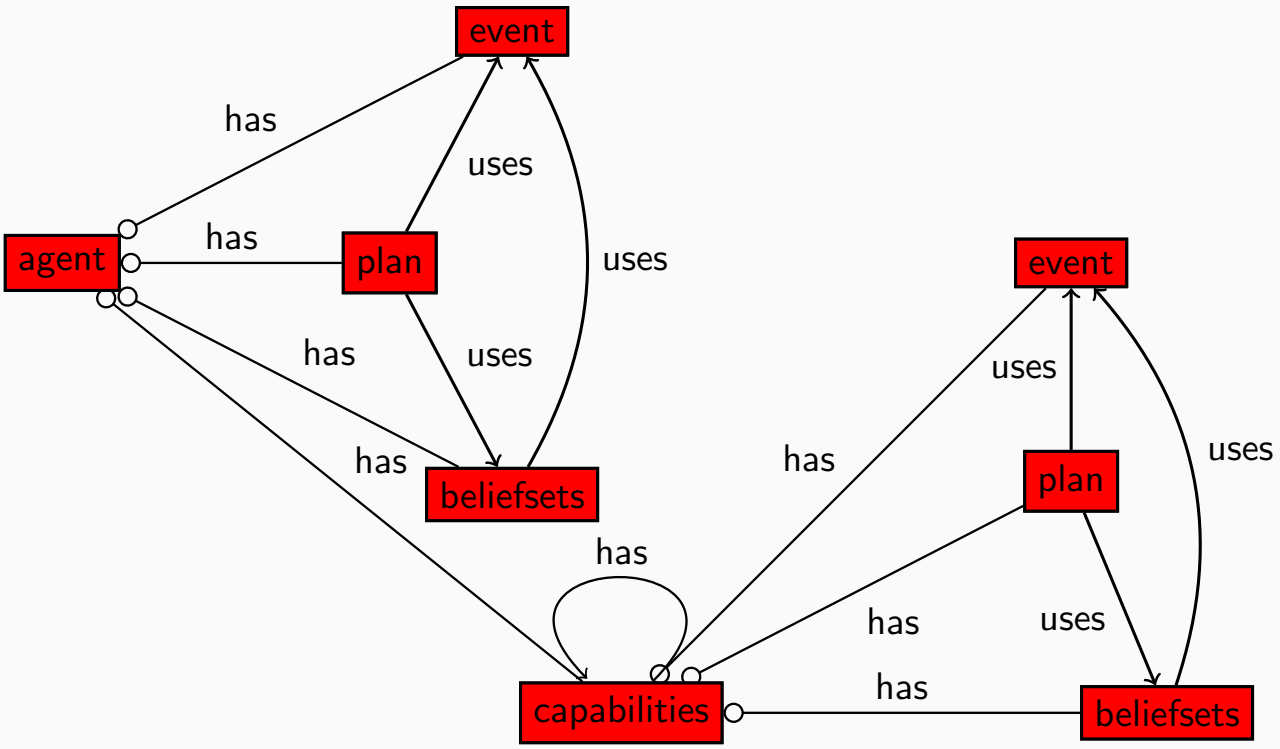




# JACK Programming Style

```
plan ExamplePlan extends Plan {
    ...
    #sends event MogulEvent msg;
    ...
    body()
    {
        @send("Mogul",msg.hello(3,true));
        if (@subtask(...))
            @send(...);
        else
            @post(....);
    }
```

# JACK Components








# Source Code

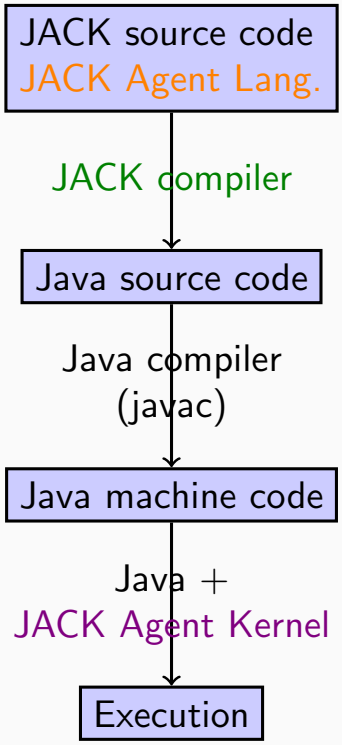
Source code will be created for some or all of the following entities:

| Extension | Usage |
|---|---|
| `.agent` | JACK agent definition. |
| `.cap` | JACK capability definition. |
| `.plan` | JACK plan definition. |
| `.event` | JACK event definition. |
| `.bel` | JACK beliefset definition |
| `.java` | Java class or interface definition |
| ..... | .... |

... plus a Java class that contains the application main() function that is the entry point for the Java virtual machine and any other Java file required by this application.

# The JACK BDI Programming Language

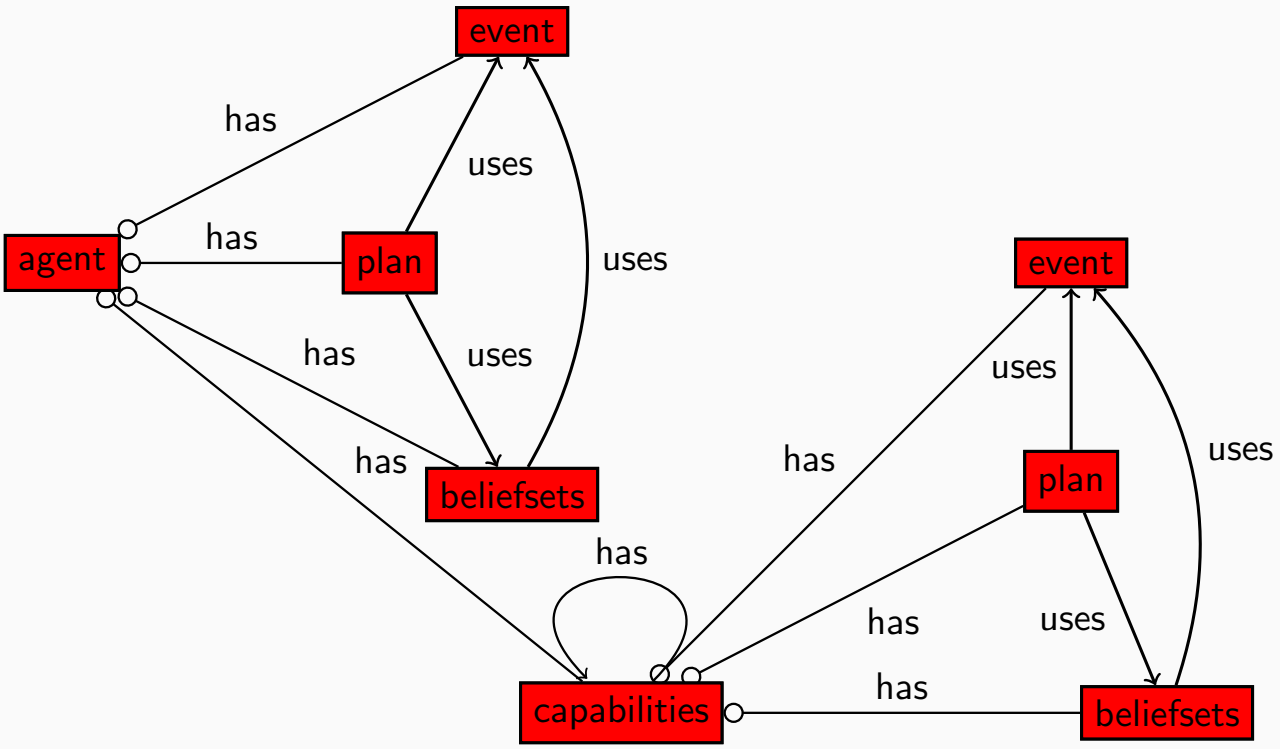
1. JACK Agent Language
   - Used to describe an agent-oriented software system.
   - Super-set of Java (agent-oriented features extensions).
2. The JACK Agent Compiler
   - Converts JACK Agent Language into pure Java.
   - This Java source code can then be compiled into Java virtual machine code.
3. The JACK Agent Kernel
   - Runtime engine for programs written in the JACK Agent Language.
   - Set of classes that give JACK Agent Language programs their agent-oriented functionality.
   - Run behind the scenes.
   - Implement the underlying infrastructure and functionality for agents.



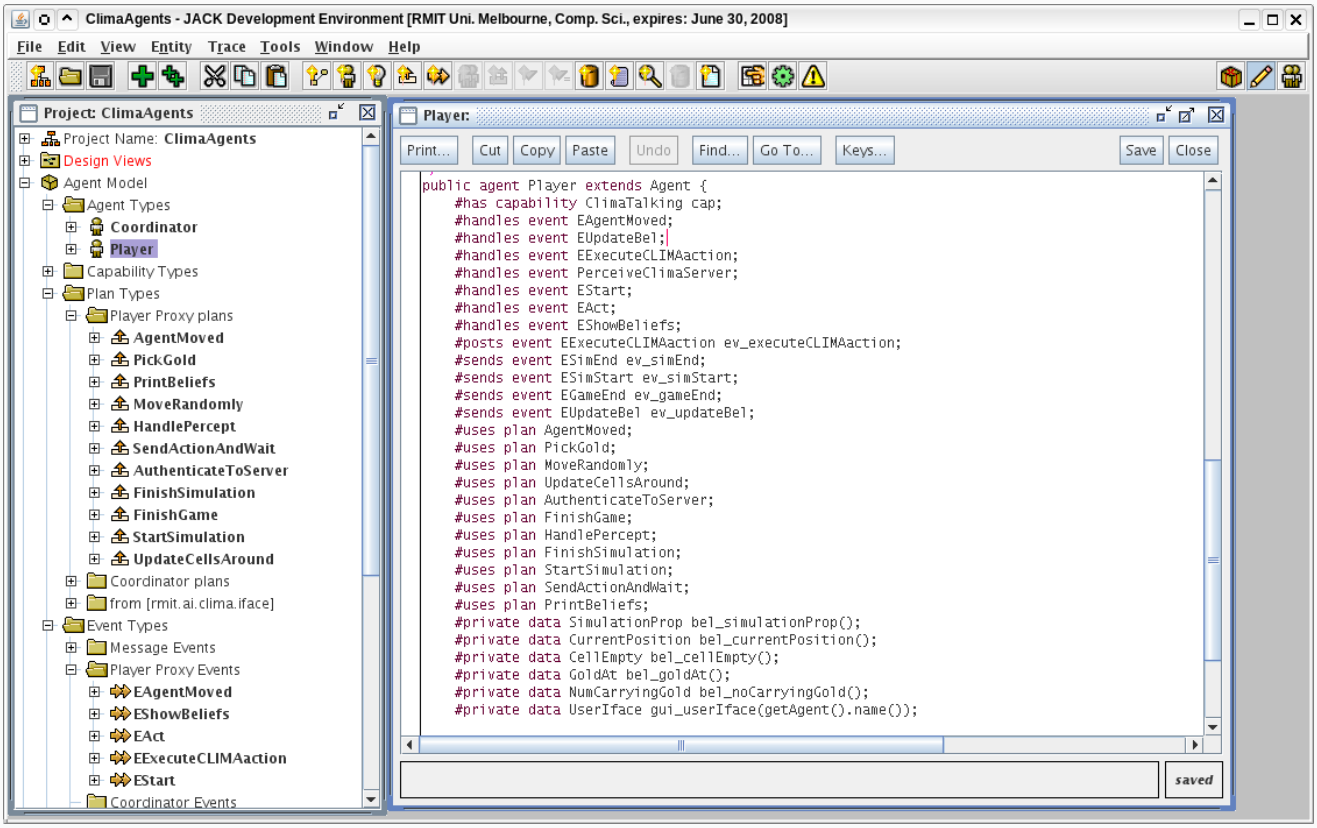





# JACK Components



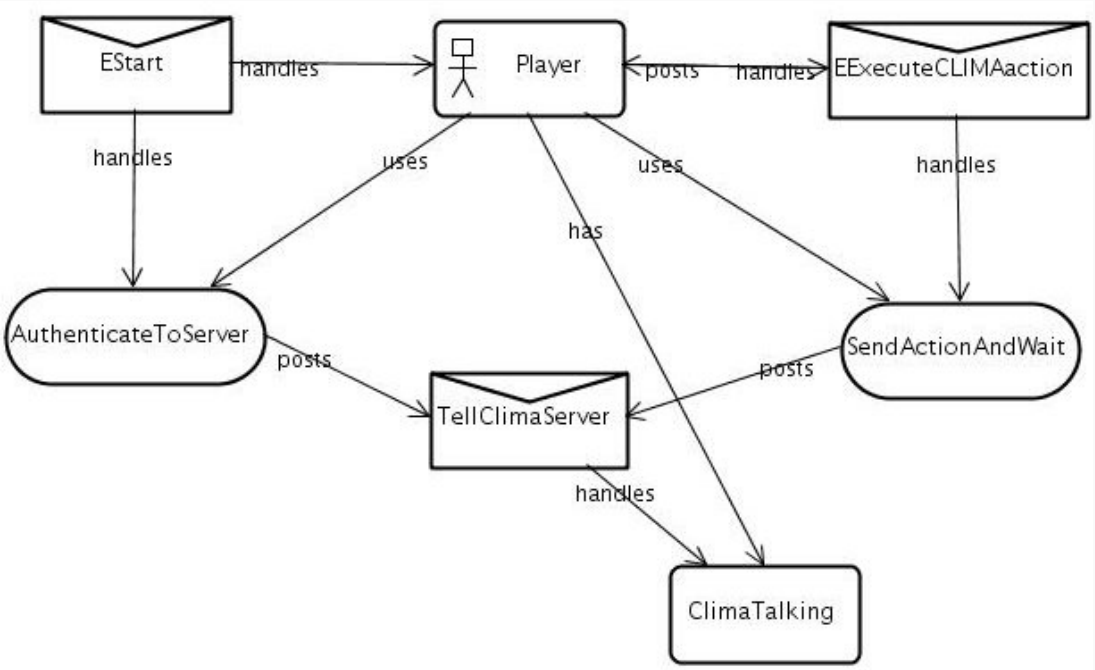
# Agents in JDE



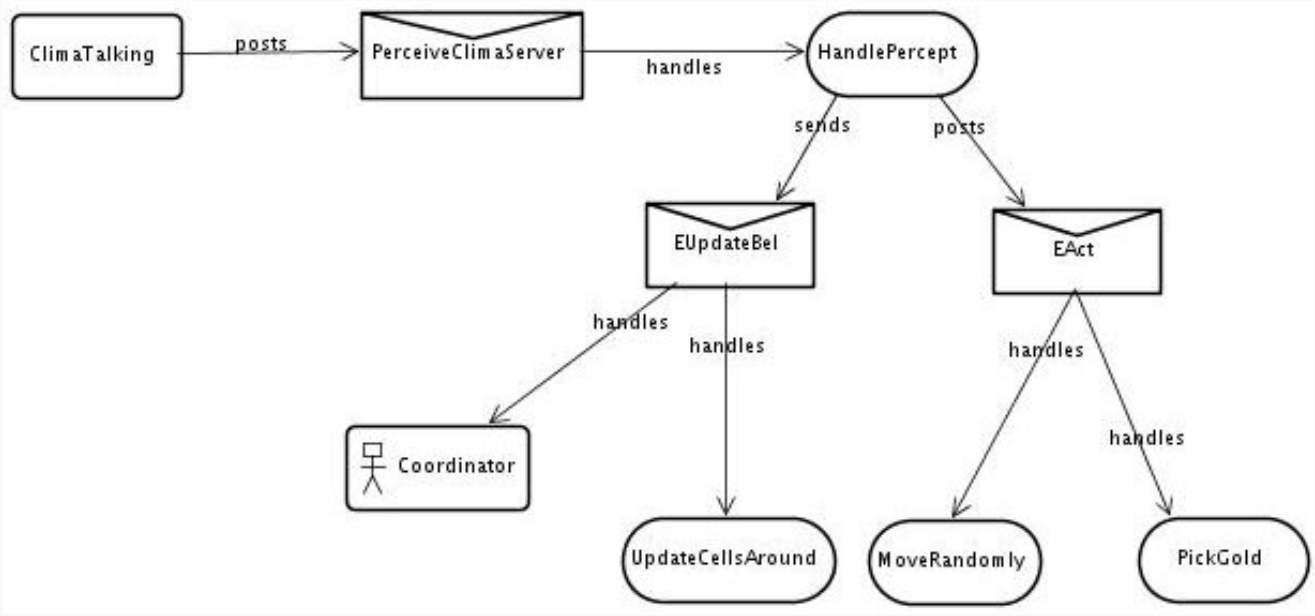





# Defining Agents in JDE: The Server Interaction View



# Defining Agents in JDE: The Player Cycle








# Defining Agents in JACK: `Player.agent`

Base class: `aos.jack.jak.agent.Agent`

```
public agent Player extends Agent {
    #has capability ClimaTalking cap;
    #handles event PerceiveClimaServer;
    #handles event EExecuteCLIMAaction;
    #handles event EAct;
    #posts event EExecuteCLIMAaction ev_executeAction;
    #sends event EInformLoc ev_informLoc;
      ...
    #uses plan MoveRandomly;
    #uses plan PickGold;
    #uses plan HandlePercept;
      ...
    #private data GoldAt bel_goldAt();
    #private data CurrentPosition bel_currPosition();
    #private data NumCarryingGold bel_noCarrGold();
     ....
}
```

# Belief Structure Declarations

Declares JACK databases and other data objects that are to be used in plans:

- `private` = each agent instance operates its own data;
- `agent` = shared among all agents of this type in the same process;
- `global` = shared among all agents in the process

```
#private data          <Type> <data_name> ( <args> ) ;
#agent data            <Type> <data_name> ( <args> ) ;
#global data           <Type> <data_name> ( <args> ) ;
```

```
#private data CurrentPosition bel_currPosition();
#agent data GoldAt bel_goldAt();
#private data NumCarryingGold bel_noCarrGold();
```





# Behavior Declarations

- Declares which events the agent handles, sends and posts:

```
#handles event <Type> ;
#sends event <Type> { <ref> } ;
#posts event <Type> { <ref> } ;
#has capability <Type> <ref> ;
```

- Declares which plans the agent uses:

```
#uses plan <Type> ;
```

```
#handles event PerceiveClimaServer;
#handles event EAct;
#posts event EExecuteCLIMAaction ev_executeAction;
#sends event EInformLoc ev_informLoc;
#uses plan MoveRandomly;
#uses plan PickGold;
#has capability ClimaTalking cap;
```

# Running the JACK Application: The `Main.java` File

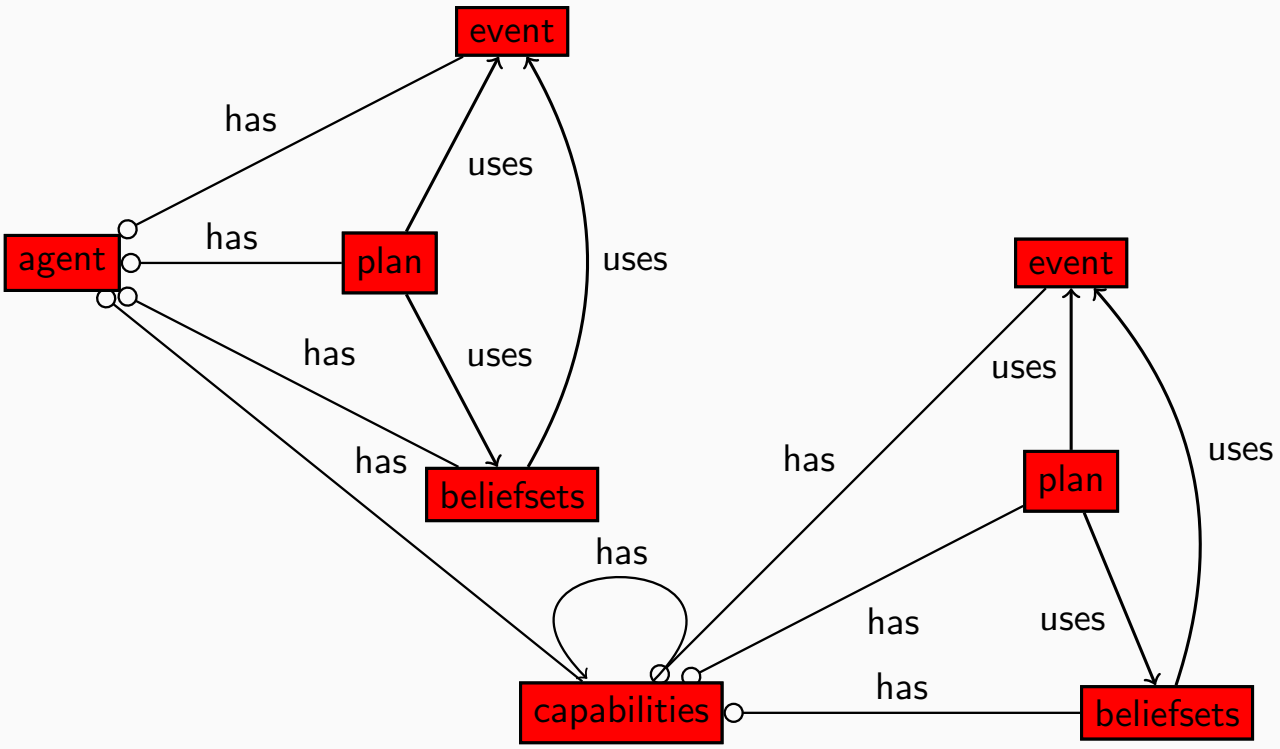
```
import agents.*;

public class Main {

    public static void main (String args[]) {
        aos.jack.Kernel.init( args );
        Player players[] = new Player[args.length];

        for(int i=1; i <= args.length; i++) {
            players[i-1] = new Player( args[i-1]);
        }
        new Coordinator("boss");
    }
}
```

```
$ java -cp jack.jar aos.main.JackBuild -r
$ java -cp jack.jar Main player1 player2 player3
```





# JACK Components



# Declarations for Events

Events are the origin of all activity within an agent-oriented system...

- Data members: information carried on in the event.
- `#posted as <name> ( <pars> ) <body>;`
  - defines how the event is posted.
- Other methods.
- Base class: `aos.jack.jak.event.*`:
  1. `Event`
  2. `MessageEvent`
  3. `BDIGoalEvent`, `BDIFactEvent`
  4. `BDIMessageEvent`
  5. `InferenceGoalEvent`
  6. `PlanChoice`
- `#set behavior <Attr> <value> ;`
  - plan reconsideration, plan set recalculation and meta-level reasoning.





# Two Event Definitions

```
// Prompts the player to move to the cell specified
public event MoveToCellEvent extends BDIGoalEvent {
    public int xCoord, yCoord;

    #posted as   post(int x, int y) {
        xCoord = x; yCoord = y;
    }
}
```

```
// Contains a role for the player proxy to perform
public event NewRole extends MessageEvent {
    public String role;

    #posted as send(String rolearg) {
        role = rolearg;
    }
}
```

# Types of Events

Normal Events: they represent transitory occurrences that initiate a single, immediate response from an agent (analogous to events in conventional event-driven programming).

- Event
- MessageEvent

BDI Events: goal-directed behavior in agents (beyond reactive behavior).

- BDIFactEvent
- BDIGoalEvent
- BDIMessageEvent
- InferenceGoalEvent
- PlanChoice

The key difference between normal events and BDI events is how an agent selects plans for execution.





# Normal Events: Reactive Behavior

- Represents an occurrence that must be acted upon immediately or not at all.
- Only interesting at the moment it is received.
- May initiate a single, immediate response from an agent.
- If the agent decides to do something (execute a plan instance) and that plan instance fails, it would not make sense to reconsider an alternative because the information that the agent is acting on is now out of date.
- Agent selects the first applicable plan instance for a given event and executes that plan instance only.

Two kinds:

1. `Event`: originated within the agent itself.
2. `MessageEvent`: originated outside the agent.
   - simple agent communication (requests, informs, etc.)
   - perceive the environment (e.g., sensors).

# Normal Events: Reactive Behavior (cont.)

Example: an RoboCup agent is programmed to play soccer:

- Environment may send the agent `MessageEvents` periodically to tell it where the ball is.
- This is a message that is *relevant* and *applicable* only at the moment that it arrives.
- If the agent is a mid-fielder and it is sufficiently close, it may invoke a plan to attempt to intercept the ball. If it is not close enough it may try to cover a designated opponent.
- If the agent is a goal-keeper, it may attempt to move itself between the ball and the goal.
- In each case, the agent is taking action on information that is applicable only at the moment at which it arrives.





# BDI Events: Proactive Behavior

- Used to commit to the desired outcome, not the method chosen to achieve it.
- Allow an agent to pursue long term goals.
- Agent does not simply react to incoming information, but sets itself a goal which it then tries to achieve.

How is this achieved?

1. Meta-level reasoning – writing plans about how to select plans.
2. Reconsidering alternative plans on plan failure – upon plan failure, consider other courses of action to achieve the goal that has been set.
3. Recalculating the applicable plan set – upon plan failure, assemble a new plan set which excludes any failed plans.

# Types of BDI Events

1. `BDIFactEvent`: Base class for all BDI Events in JACK.
   - Posted internally by agent itself; allows meta-level reasoning.
2. `BDIMessageEvent`: Received by agents from external sources.
   - Allows meta-level reasoning.
3. `BDIGoalEvent`: An objective that an agent wishes to achieve.
   - Meta-level reasoning.
   - Alternative plan selection.
   - Plan set recalculation are all available for this event type.
4. `InferenceGoalEvent`: Processes all applicable plans (rather than only one of them).
5. `PlanChoice`: Posted when there is more than one plan to choose from and optionally handled by the agent to implement meta-level reasoning.





# Meta-Level Reasoning: An Example

Imagine an agent programmed to play as a soccer goal keeper:

1. Three plans available for defending a penalty – jump to the left, jump to the right and stay in the center.
2. When a penalty is called (arriving as a message event) all three of these plans would be applicable.
3. If the message event is of class `MessageEvent`, the agent can only select the first plan declared in the agent.
4. If the message event is a `BDIMessageEvent`, the agent has the option of performing meta-level reasoning to determine which plan is best.
5. Meta-level reasoning may take into account various statistics from the game and past successes of each approach, then make the plan selection.

Meta-reasoning by handling special BDI event of type `PlanChoice`.

# BDIGoalEvent: Modeling Agent's Goals

Represents a goal or objective that an agent wishes to achieve:

- Arise within an agent as a result of "reasoning":
  - Example: a pilot agent adopting a goal to land an airplane.
- Agent commits to achieving the goal: agent tries hard!
- Agent will use all the reasoning powers at its disposal:
  - √ ability to choose between the plans that are applicable (meta-reasoning).
  - √ ability to re-consideration of alternative plans (failure handling);
  - √ the ability to reassess which plans are applicable (failure handling).
- Agent may adopt a goal with the following objectives in mind:
  1. achieve the goal (`@achieve`);
  2. insist that the goal is achieved (`@insist`);
  3. test whether the goal can be achieved (`@test`);
  4. determine a situation in which the goal can be achieved (`@determine`).





# InferenceGoalEvent and PlanChoice BDI Events

`PlanChoice` used to modularly program meta-level reasoning:

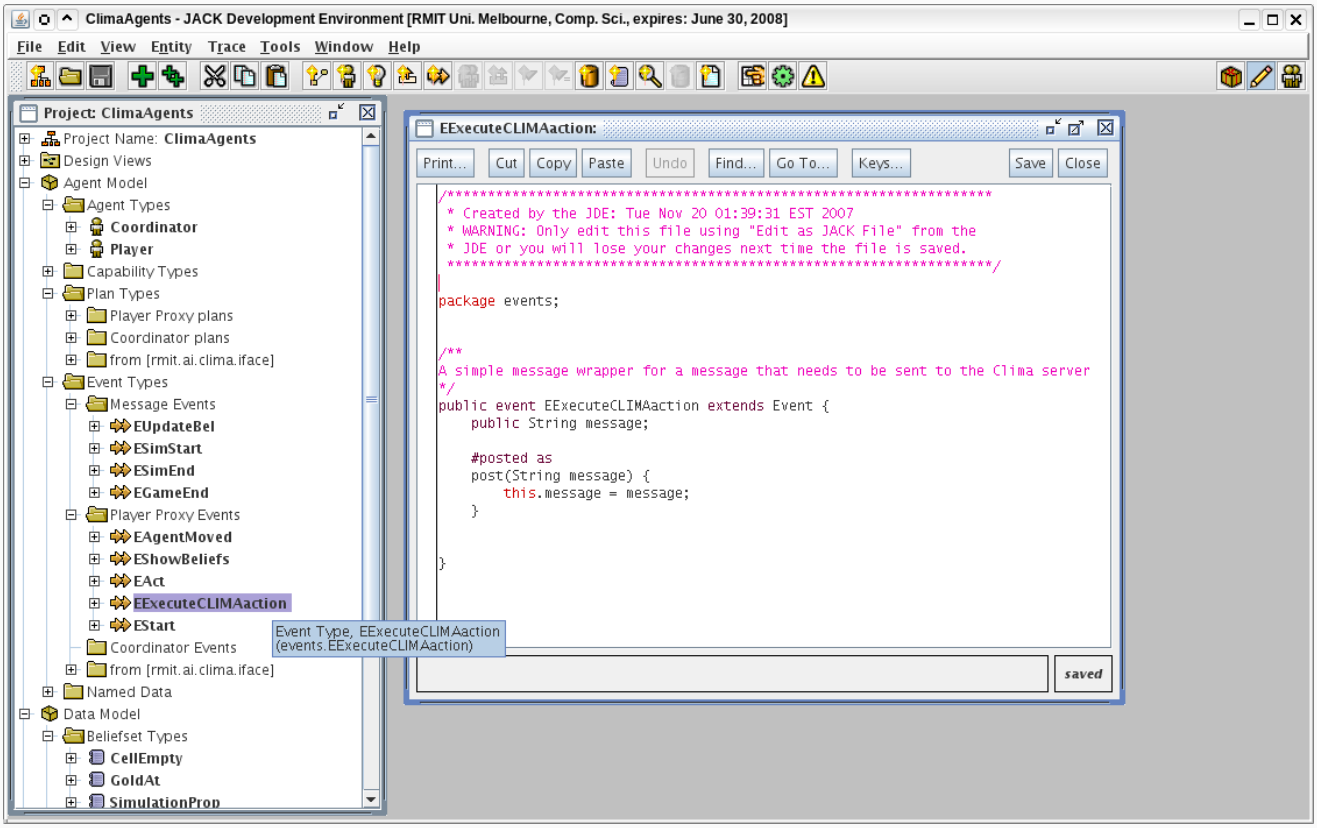
- posted automatically when there is more than one applicable plan;
- programmer may (or may not) want to handle such special event;
- corresponding plans reason about what the best plan to use is.

`InferenceGoalEvent` processing all applicable plans.

- all applicable plans (and their instances) are executed;
- failure is ignored (i.e. the event always succeeds);
- Example: an event `!moveBlockTo(X,box)` and one plan-rule of the form:

$$movelBlock(X, L) : Block(X) \leftarrow pick(X); goTo(L); \ldots$$

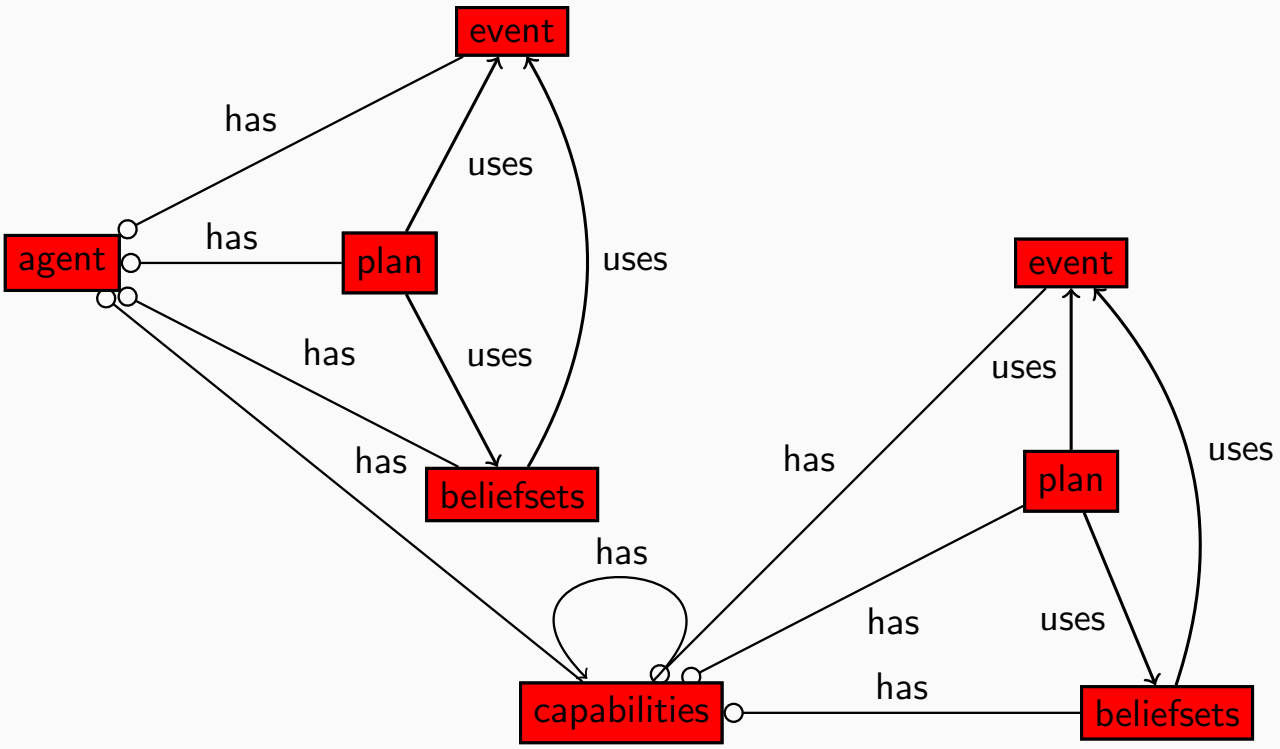
# Events in JDE








# JACK Components



# Plans

Plans can be thought of as pages from a procedures manual, or even as being like methods and functions from more conventional programming languages.

They describe exactly what an agent should do when a given event occurs.

Agent is equipped with a set of plans, describing the agent's set of skills.

When the event that a plan addresses occurs, the agent can execute this plan to handle it.

So, suppose the agent has the following BDI event:

```
// Prompts the player to act twoards the game server
public event EAct extends BDIGoalEvent {

    #posted as   post() { }
}
```





# Handling EAct BDI Event: Random Movement

```
public plan MoveRandomly extends Plan {
    final static String []
              actions = { "left", "right", "up", "down" };
    Random random = new Random();
    #handles event EAct ev_act;
    #posts event EExecuteAction ev_executeAction;

    static boolean relevant(EAct ev) {
        return true;
    }

    context()    {
        true;
    }

    #reasoning method body()    {
        ev_executeAction.post(actions[ random.nextInt(4) ]);
    }
}
```

# BDI Plan Selection

1. Identify the plans which handle the event type: `#handles event ..`
   - syntactic relevance.
2. Use the `relevant()` method to check additional information regarding the event.
   - inspect data carried on in the event.
3. Use the `context()` method to check information stored as part of the agent's beliefs.
   - defines the set of all applicable plans (types & instances).
4. All applicable plans are collected at this point.
5. If there are still multiple plans left in the applicable plan set, additional means are used to select one of them:
   - declaration order;
   - prominence w.r.t. plan ranks;
   - meta-level reasoning via `PlanChoice` handling.





# Pick Gold Pieces I

```
public plan PickGold extends Plan {

    #handles event EPickGold ev_pickGold;
    #posts event EExecuteAction ev_executeAction;

    static boolean relevant(EAct ev)    {
        true;
    }

    context()    {
        true;
    }

    #reasoning method body()    {
        ev_executeAction.post(actions[ "pick" ]);
    }
```

# Pick Gold Pieces II

```
public plan PickGold extends Plan {

    #handles event EPickGold ev_pickGold;
    #posts event EExecuteAction ev_executeAction;

    static boolean relevant(EAct ev)    {
        return (ev_pickGold.no < 10);
    }

    context()    {
       true;
    }

    #reasoning method body()    {
       ev_executeAction.post(actions[ "pick" ]);
    }
```





# Pick Gold Pieces III

```
public plan PickGold extends Plan {
    logical int $posX, $posY;

    #handles event EPickGold ev_pickGold;
    #uses data CurrentPosition bel_currentPosition;
    #posts event EExecuteAction ev_executeAction;
    #uses data GoldAt bel_goldAt;

    static boolean relevant(EAct ev)    {
        return (ev_pickGold.no < 10);
    }
    context()    {
     (bel_currentPosition.get($posX,$posY) &&
      bel_goldAt.check($posX,$posY));
    }

    #reasoning method body()    {
       ev_executeAction.post(actions[ "pick" ]);
    }
```

# The Reasoning Method body()

Special kind of method in the JACK Language called a reasoning method.

Reasoning methods are quite different from ordinary methods in Java.

Each statement in a reasoning method is treated as a logical statement.

- Failure of a plan statement will cause the body() method to fail.
- If execution proceeds to the end, the body() method succeeds.

```
...

#reasoning method body()    {
    ev_executeAction.post(actions[ "pick" ]);
}
```

The body() method can call *other* reasoning methods as it executes.
∴ Describe logical behavior that the reasoning method should adhere to.

Reasoning methods execute as Finite State Machines (FSMs).





# Some Reasoning Methods

- `@subtask(event);`
- `@post(event);`
- `@send(agent_name, message_event);`
- `@wait_for(wait_condition);`
- `@action(parameters) <body>;`
- `@maintain(logical_condition, event);`
- `@reply(original_event, reply_event);`
- `@sleep(timeout);`
- `@achieve(condition, goal_event);`
- `@insist(condition, goal_event);`
- `@test(test_condition, goal_event);`
- `@determine(binding_condition, goal_event);`
- `@parallel(parameters) <body>;`

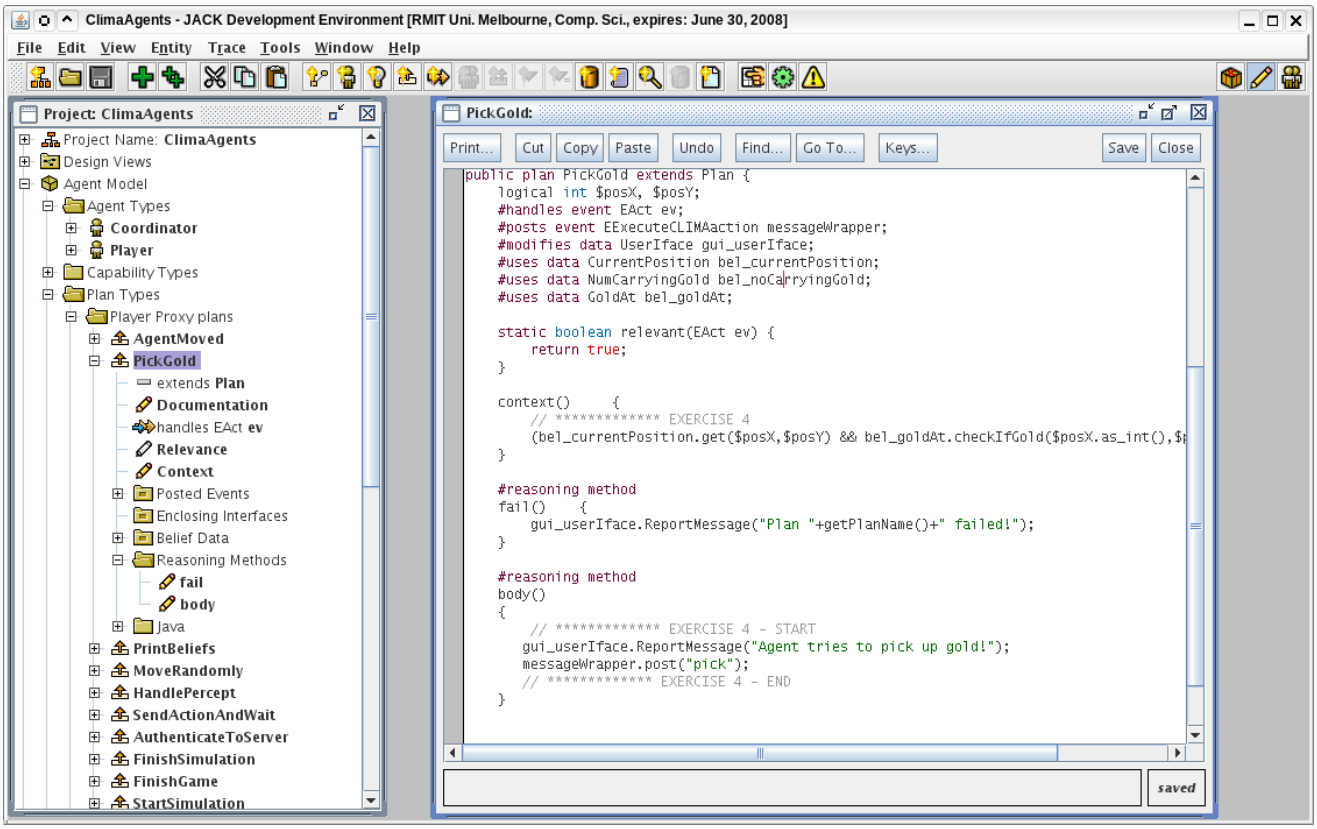
# Gold Deposition Plan

```
public plan DepositGold extends Plan {

  #reasoning method  body() {
    // wait until we believe this
    @wait_for(readyToGo.query(true));
    ...
    // find out where we believe the depot is
    logical int $depX,$depY;
    depotPosition.get($depotX,$depotY);

    state.add("toDepot");
    // first, go to depot position
    @maintain(state.get("toDepot"), goTo.post($depX,$depY));
    // then, drop all gold
    @subtask(drop.post());

    // finally, inform coordinator that we are free again
    @send(coordinator, myState.post("free"))
}
```

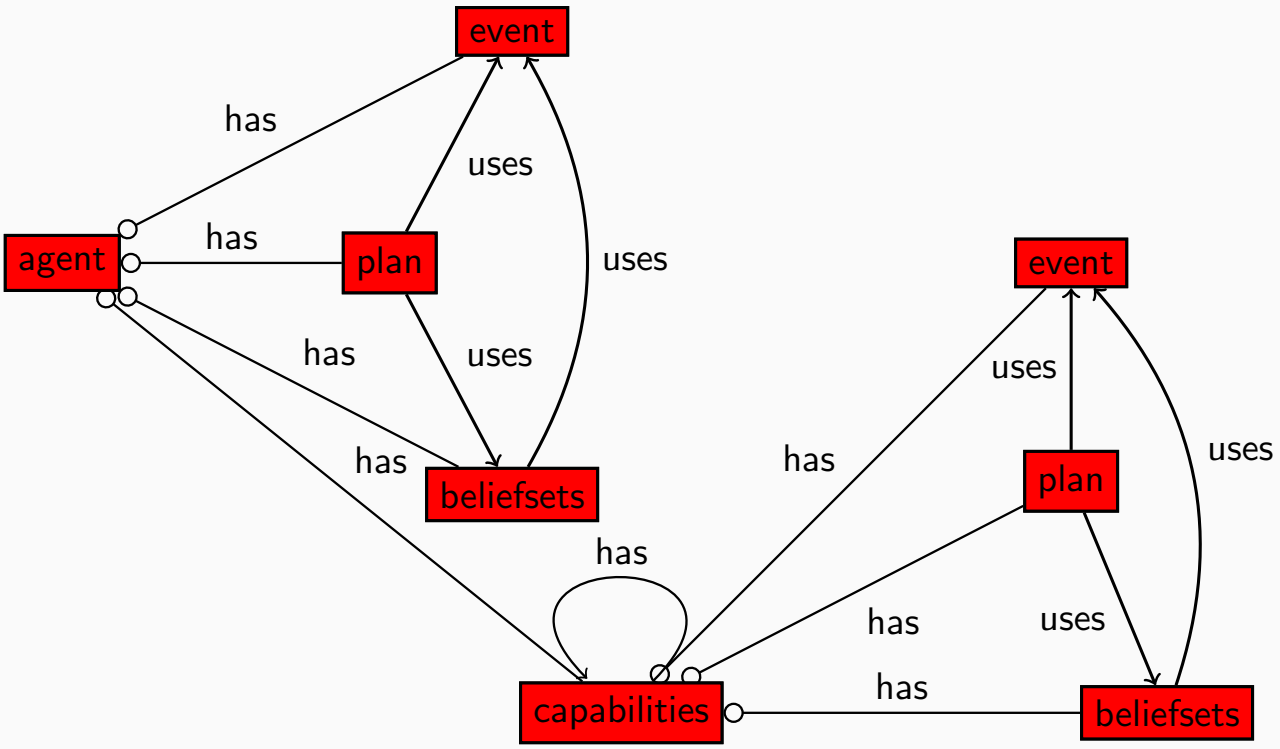




# Plans in JDE



# JACK Components








# Beliefsets

- Used to maintain an agent's beliefs about the world.
- Represents these beliefs in a first order, tuple-based relational model.
- Designed specifically to work within the agent-oriented programming paradigm & integrated with the other JACK Agent Language classes:
  1. Automatic maintenance of logical consistency and key constraints.
  2. Either `OpenWorld` or `ClosedWorld` logic semantics.
  3. Ability to post events automatically when a beliefset changes.
     - initiate action within the agent based on a change of beliefs.
  4. Ability to support beliefset cursor statements: multiple query bindings.

# Beliefset Definitions

```
beliefset RelationName extends <ClosedWorld | OpenWorld > {
        // Zero or more #key field declarations.
        // Zero or more #value field declarations
        // One or more queries
}
```

- #key field FieldType field_name;
- #value field FieldType field_name;

- #indexed query methodName(parameters)
- #linear query methodName(parameters)

- #complex query methodName(parameters) <statements>
- #function query return-type methodName(param)<statements>

- #posts event EventType handle





# Storing the Current Position

```
public beliefset CurrentPosition extends ClosedWorld {
   #value field int posX;
   #value field int posY;
   #indexed query check(int posX, int posY);
   #indexed query get(logical int posX, logical int posY);
```

```
public plan DepositGold extends Plan {
    #uses data CurrentPosition bel_currentPosition;
      ...
    #reasoning method body() {
      ...
      bel_currentPosition.add(10, 15);
      ...
      logical int $x,$y; // unbound variables here
      bel_currentPosition.get($x, $y);
      ...
      if (bel_currentPosition.check(8, 12)) { .... };
    }
}
```

# Storing the Current Position

```
public beliefset GoldAt extends OpenWorld {
  #key field int posX;
  #key field int posY;
  #indexed query check(int posX, int posY);
  #indexed query get(logical int posY, logical int posY);
  #function query public int getTotalGoldCount() {...}
```

```
public plan Collect extends Plan {
    #uses data GoldAt bel_goldAt;
      ...
    #reasoning method body() {
      ...
      bel_goldAt.add(x+1, y);
      bel_goldAt.remove(10,12);
      int totalNoOfGold = bel_goldAt.getTotalGold();
      bel_goldAt.check(10,12); // would (generally) fail!
      ...
    }
}
```



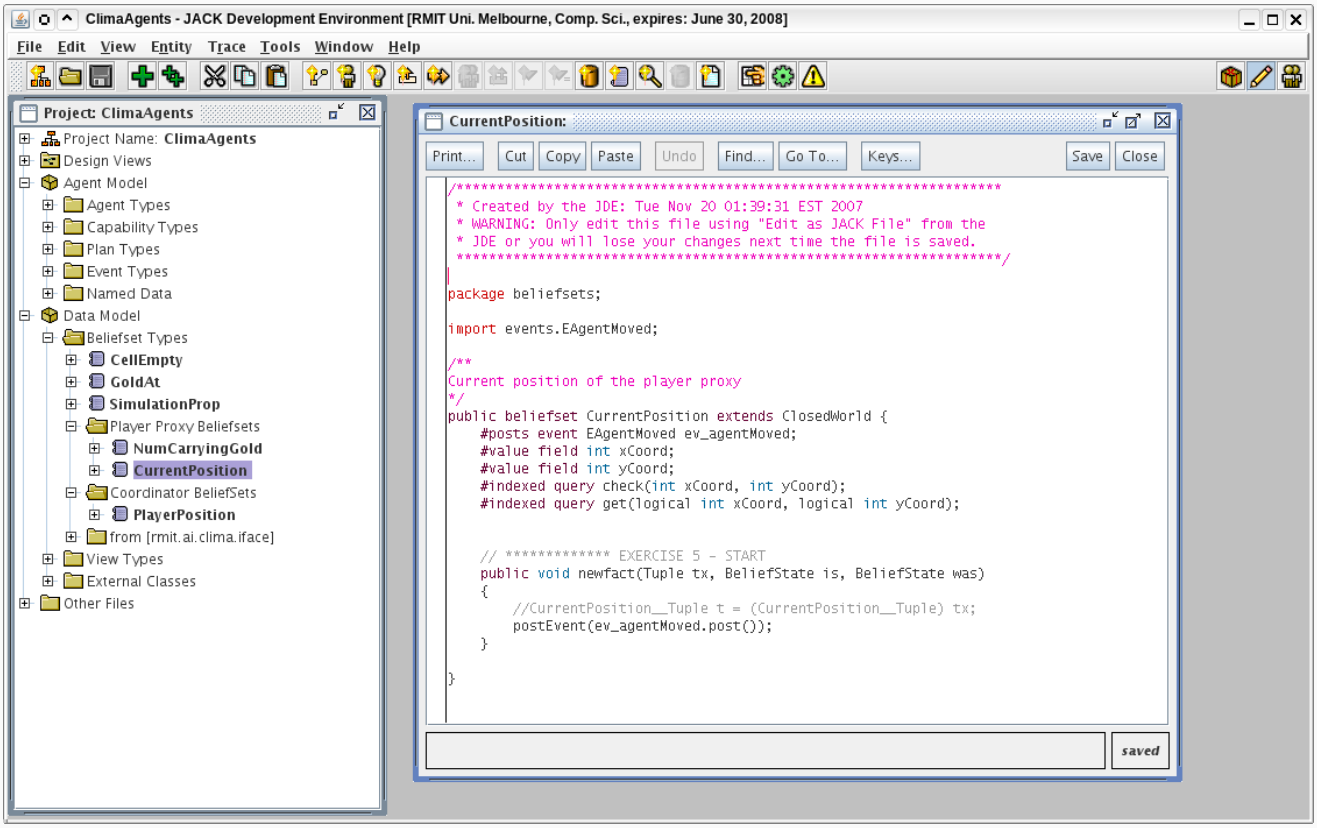


# Automatic Events: Pro-active Behavior

```
public beliefset CurrentPosition extends ClosedWorld {
   #value field int posX;
   #value field int posY;
   #indexed query check(int posX, int posY);
   #indexed query get(logical int posX, logical int posY);
  #posts event EAgentMoved ev_agentMoved;
```

```
public plan ReportMove extends Plan {
    #handles EAgentMoved ev_agentMoved;
      ...
    #reasoning method body() {
      ...
    }
}
```

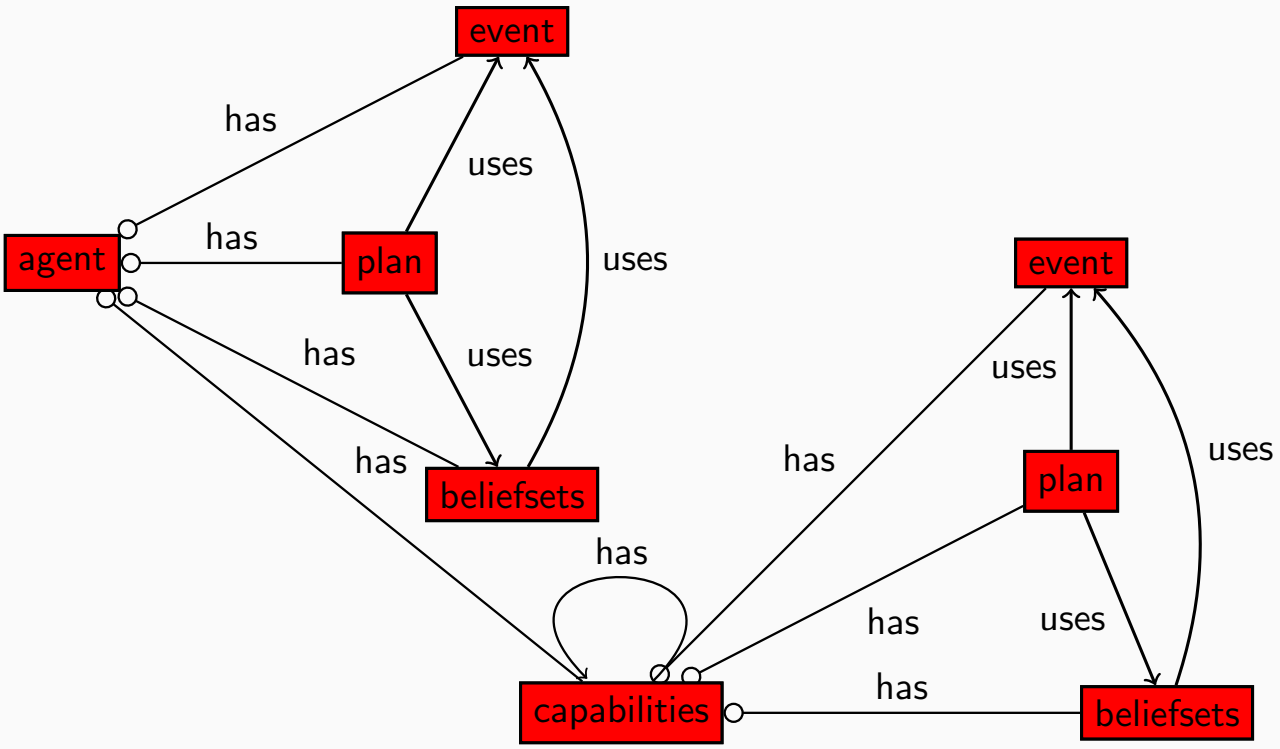
# Beliefsets in JDE








# JACK Components



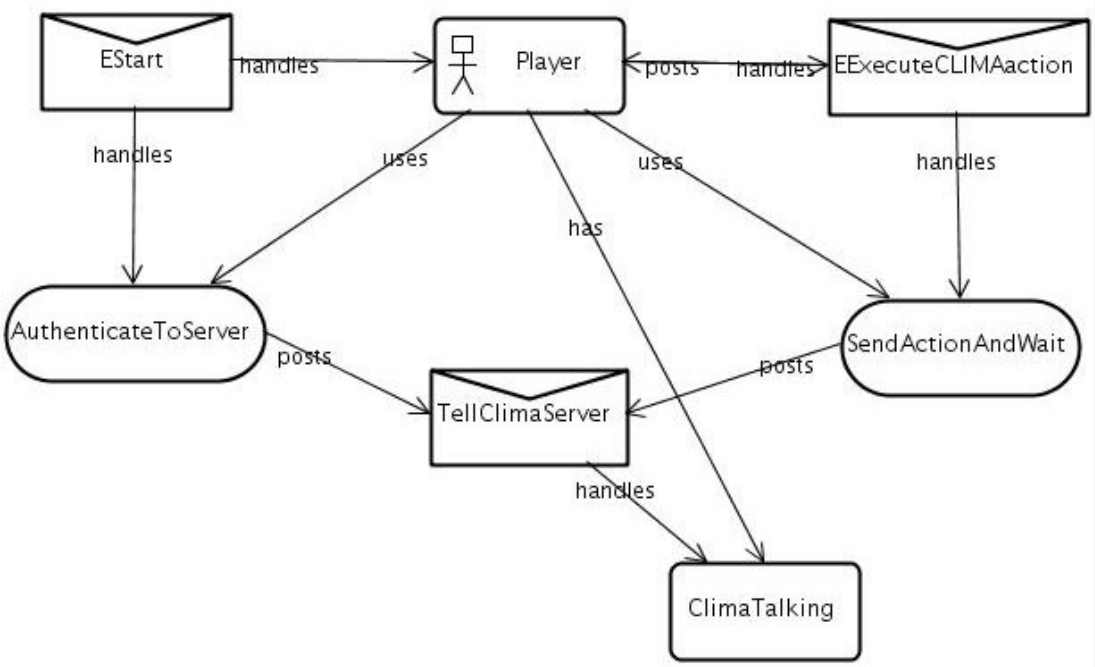
# Capabilities

- Means of structuring reasoning elements of agents into clusters that implement selected reasoning capabilities.
  - represent functional aspects of an agent that can be plugged in as required.
- Capabilities are built in a similar fashion to simple agents
- A capability can include:
  1. events;
  2. beliefsets,
  3. plans,
  4. Java code;
  5. other capabilities.
- Simplifies agent system design, allows code reuse, and encapsulation of agent functionality.

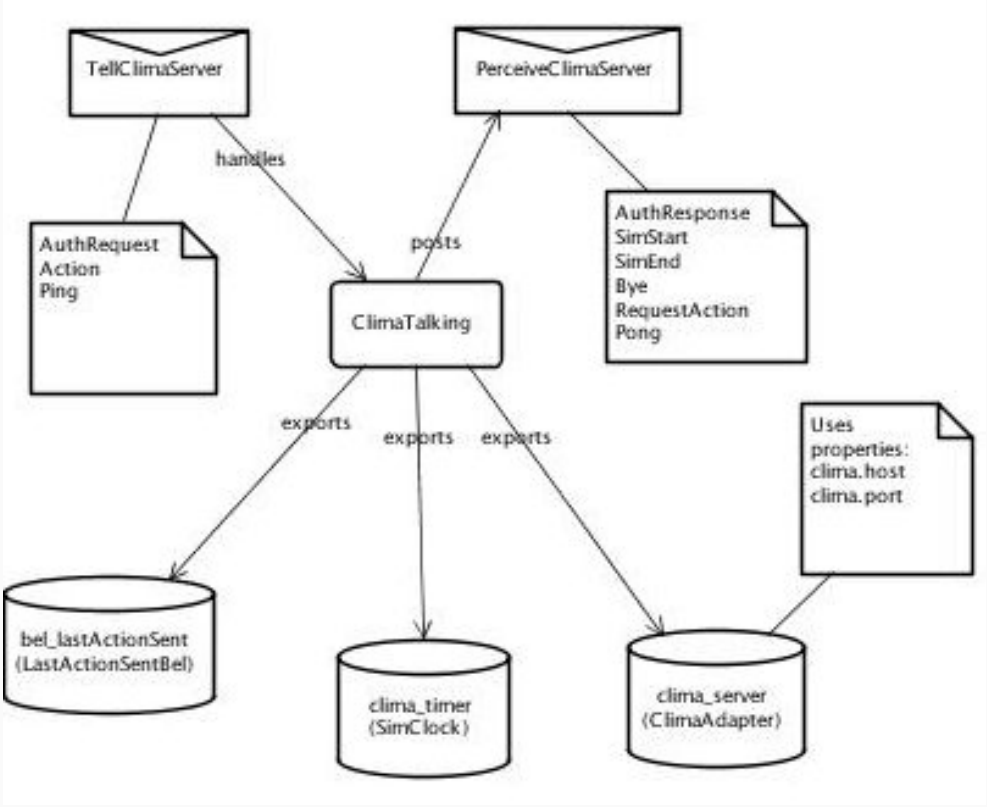




# Defining Agents in JDE: The Server Interaction View



# The ClimaTalking Capability








# Review

In this lecture we review BDI programming in JACK:

1. Review the general agent-approach by extending Java:
   - The JACK Agent Language.
   - The JACK compiler;
   - The JACK kernel.
2. Review the different structures of the agent language:
   - Agents;
   - Events;
   - Plans;
   - Beliefsets;
   - Capabilities.

## Next Lecture

# Agent-oriented Methodologies & PDT





## BDI Programming Systems



P. Busetta, Ralph Rönnquist, A. Hodgson, and A. Lucas.
JACK Intelligent Agents: Components for intelligent agents in Java.
AgentLink News Letter, Jan 1999.


M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge.
The dMARS architechure: A specification of the distributed multi-agent reasoning system.
*Autonomous Agents and Multi-Agent Systems*, 9(1–2):5–53, 2004.


R. H. Bordini and J. F. H ubner.
BDI agent programming in AgentSpeak using Jason.
In *Proceedings of CLIMA-06*, pages 143–164, 2006.


Alexander Pokahr, Lars Braubach, Winfried Lamersdorf
Jadex: Implementing a BDI-Infrastructure for JADE Agents.
In *Search of Innovation (Special Issue on JADE)*, 3(5):76–85, 2003.


David Morley, and Karen Myers
The SPARK Agent Framework.
In *Proc. of AAAMS-04*, 712–719, 2004.