Agent zero tutorial

# Table of contents

Section 1: Starting with Agent Zero

1. Downloading and setting up the eclipse plug-in.
2. Creating your first AgentZero project.
3. Writing your first algorithm.
4. Running and debugging the algorithm.

Section 2: working with synchronized search

1. The synchronized execution mode
2. Implementation to look at

Section 1: Starting with Agent Zero

Everyone have to start somewhere,

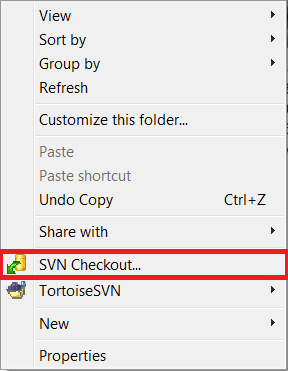
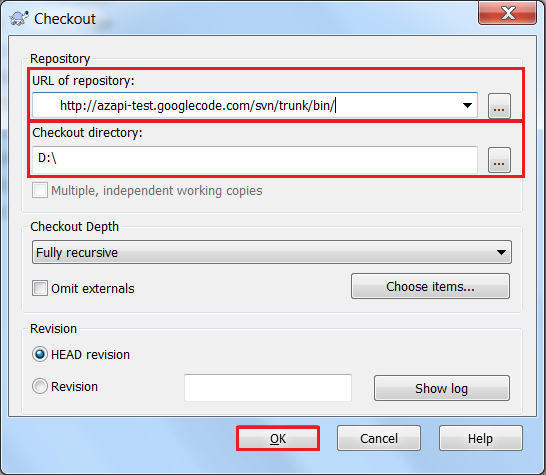
In the following section we will cover the basics of installing agent zero and running your algorithms

If you already familiar with agent zero skip this section – you may want to start at Secetion 2 for more advance topics.

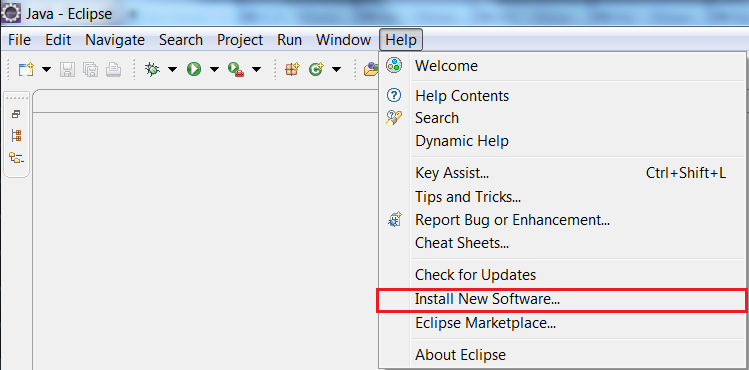
Chapter 1: Downloading and setting up the eclipse plug-in

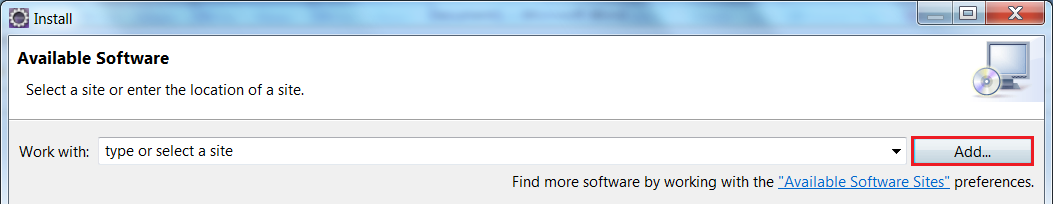
For the download you need to use SVN client. This demonstration is using Tortoise SVN, but you can use any client you know.

**Step 1: downloading the plug-in:**

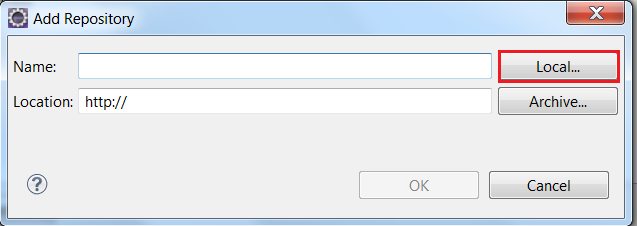
1. Download and install Tortoise SVN from [here](http://tortoisesvn.net/downloads.html).
2. Create a new folder named agentzero.
3. Left click on the agentzero folder.
4. In the menu that opens choose SVN checkout.
5. In the checkout screen enter on the “URL of repository”  
   field the following address:  
   http://azapi-test.googlecode.com/svn/trunk/bin/  
   and press OK.  
   
6. Wait until the download is complete.

**Step 2: installing the plug-in:**

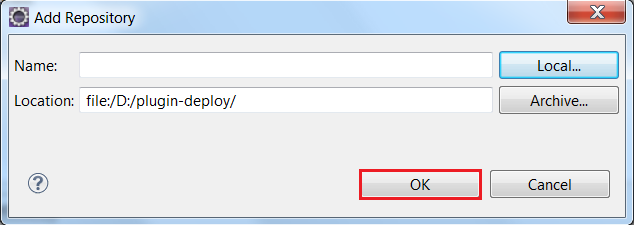
1. Open eclipse and go to help=> install new software.
2. On the “Install” screen press “Add”.



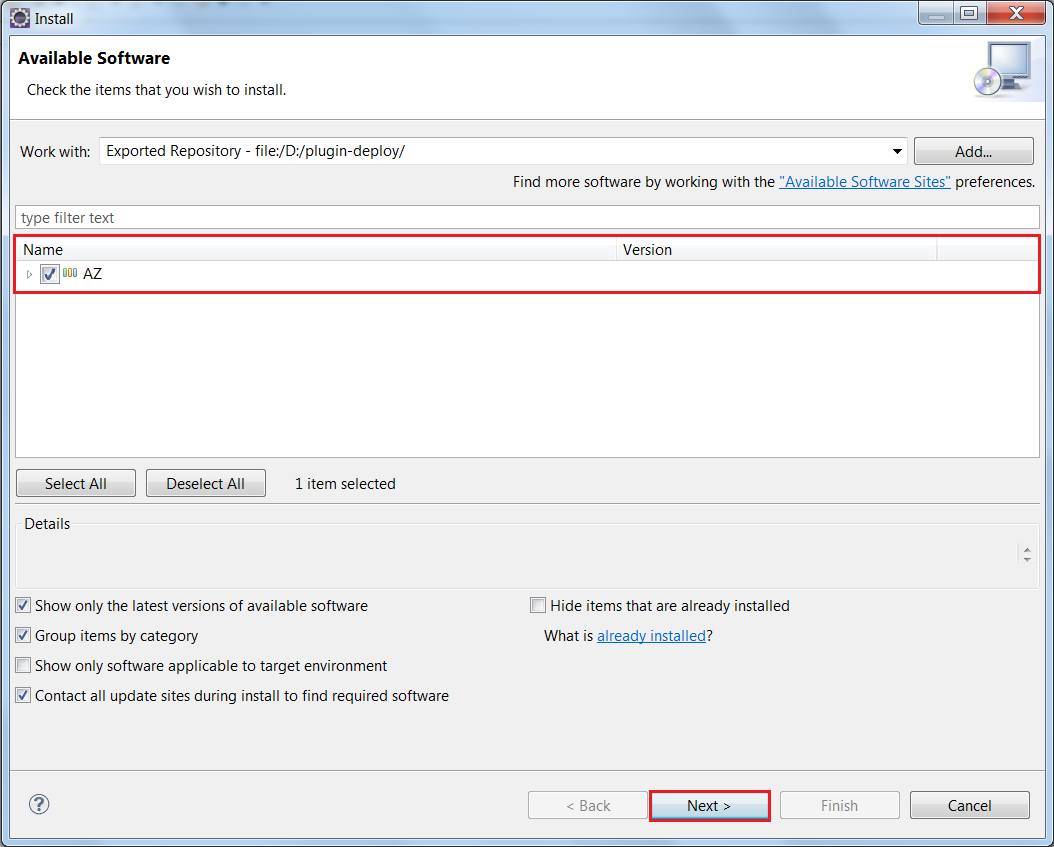
1. On the “Add Repository” screen press “local”, choose the agentzero folder and press “OK”.



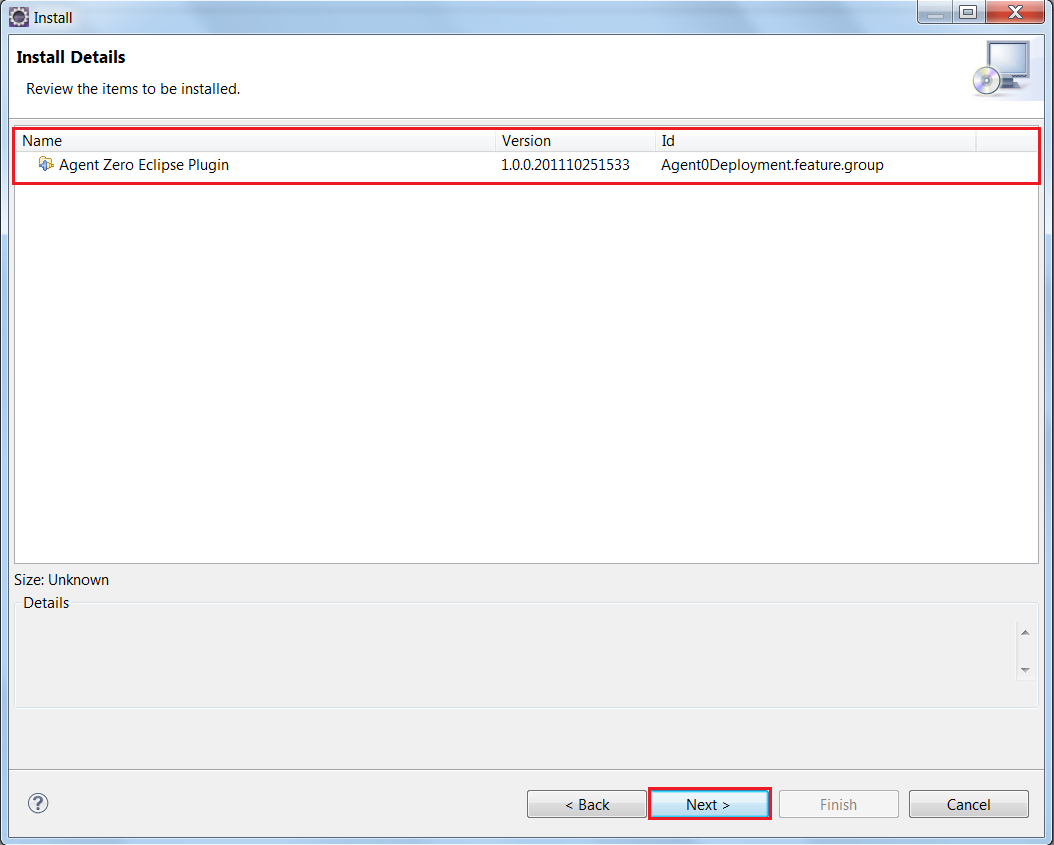
1. Enter the name “AgentZero” in the name field and press “OK”.



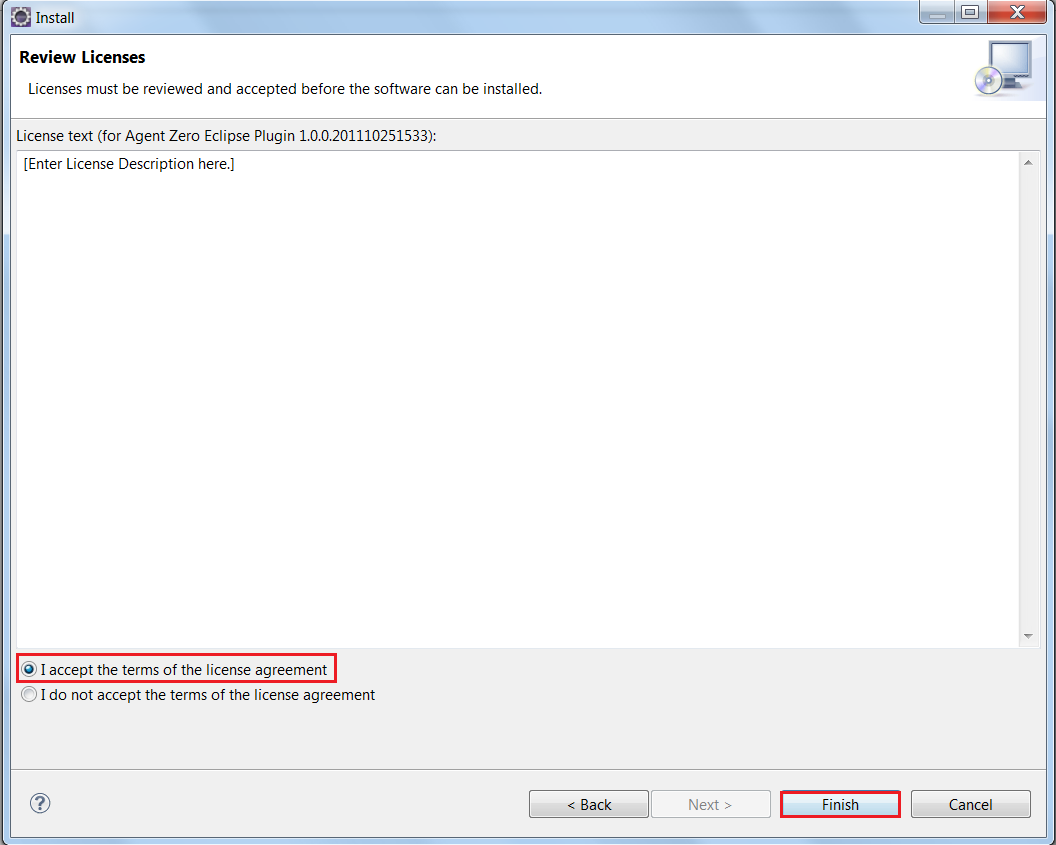
1. Check the category AZ and press “Next”.



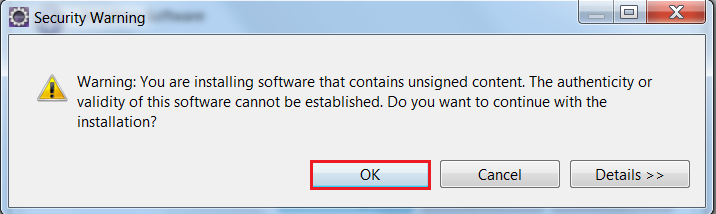
1. Press “Next” again to begin the installation.



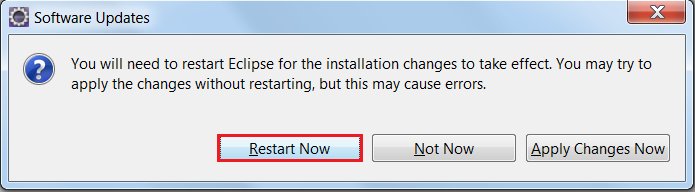
1. Check “I accept the terms of the license agreement” and press “Finish”.



1. If a “Security Warning” screen will appear, just press “OK”.

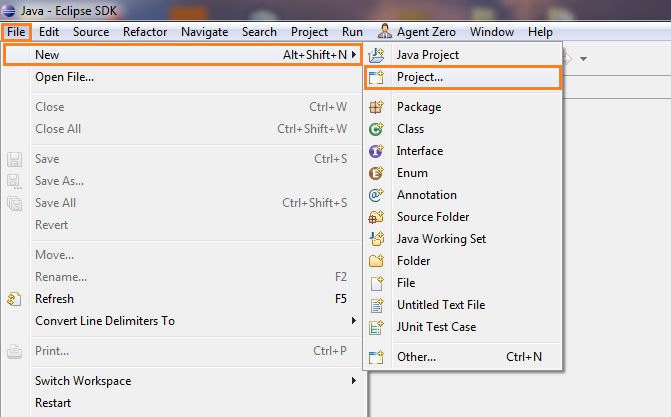


1. In the “Software Updates” screen press “Restart Now”.



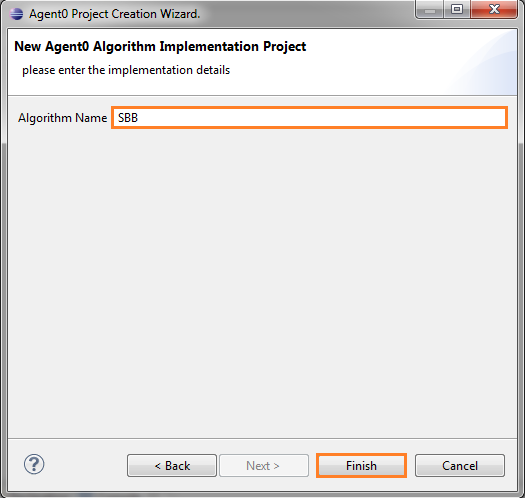
Chapter 2: Creating your first AgentZero project

1. Open Eclipse and press File=> New=> Project.



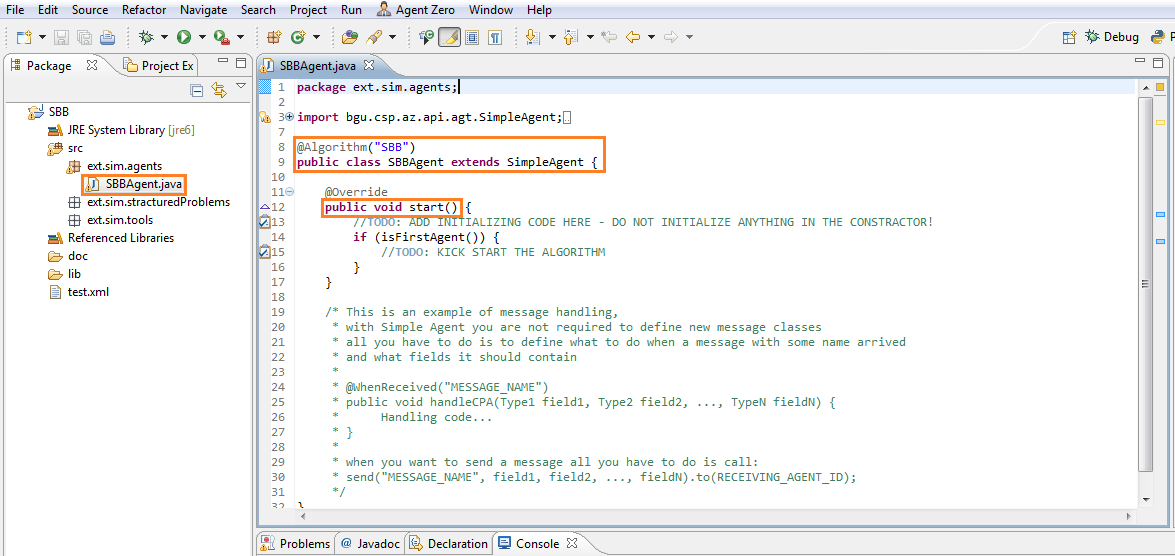
1. In the “Agent0 Project Creation wizard” fill the ‘Algorithm Name” field as you wish and press “Finish”.

For the purpose of this tutorial, it will be called SBB.



1. Navigate to src=> ext.sim.agents=> SBBAgent in the package explorer.

You will see that a minimal template has been created for you.



Chapter 3: writing your first algorithm

For starters, we encourage you to read the JavaDoc and especially the Agent class. At this writing, the JavaDoc can be downloaded via SVN in the following link: <http://azapi-test.googlecode.com/svn/trunk/doc/>.   
In this tutorial, we will cover the most common API parts, but you can check-out   
<http://azapi-test.googlecode.com/svn/trunk/alg/> to see the already implemented algorithms.

**Algorithm Entry Point:**

Simple agent is an abstract class which requires the implementation of the function start().

This is the entry point of the algorithm. The function start() will be called once for each agent.

**Defining Messages:**

The message handling methodology in SimpleAgent is that messages are actually remote procedure calls.

Defining a new message means defining a new method and annotating it with @WhenRecieved annotation

(there is a shortcut for this in the plugin : Alt+M). The message arguments are the parameters of the method.

The message also contains Metadata for passing data about the message (timestamp, sender, etc.).

The actual message object can be retrieved by calling currentMessage() for cases in which you need access to the message.

**Sending Messages:**

Message can be sent via one of the two methods:  
\* broadcast(MessageName, Args…) - will broadcast the message to all agent Except the sending agent.  
\* send(MessageName, Args…).to\*(…) – will send the message to the agents corresponding to the type of   
 “to”.

The sent messages will be posted via the mail service, to the relevant agent message queue and when the time is right, the remote procedure will get called.

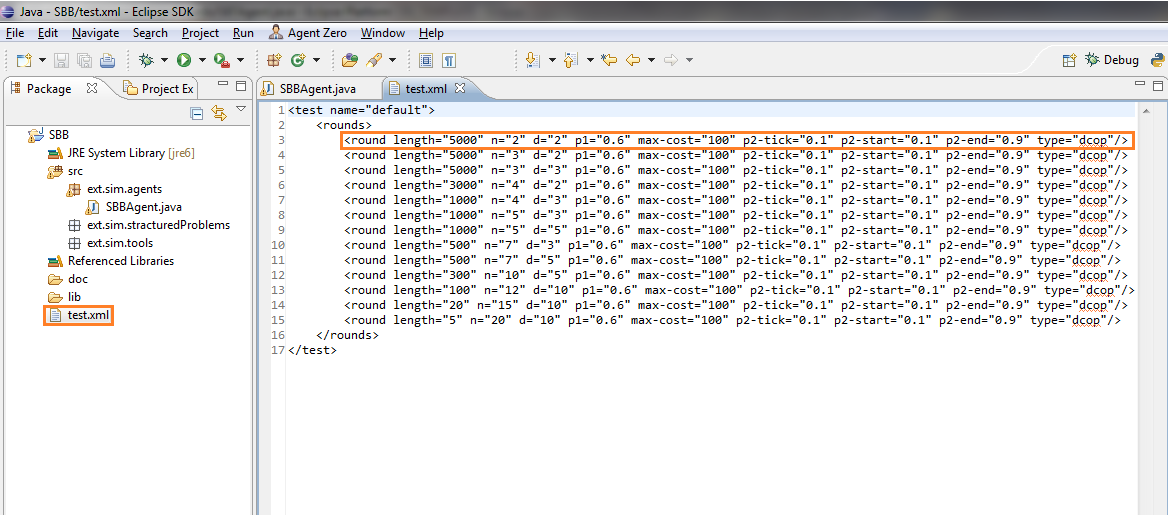
**Hooks:**

A hook is a method that is used to alter the behavior of the agent’s basic functionality.  
At this writing, there are two hooks available:  
\* beforeMessageSending(Message) - override this function in case you want to make some action every   
 time before sending a message. This is a great place to write logs, attach timestamps to the message   
 metadata etc.   
\* beforeMessageProcessing(Message) - override this method to perform preprocessing before messages  
 arrive to their remote procedures. This is the place to change the message or even return completely other   
 one. In the case of returning null the message is rejected and dumped.

Those are the very basic knowledge that is needed for implementing an algorithm. You should read the JavaDoc for more details.

Chapter 4: Running and debugging your algorithm

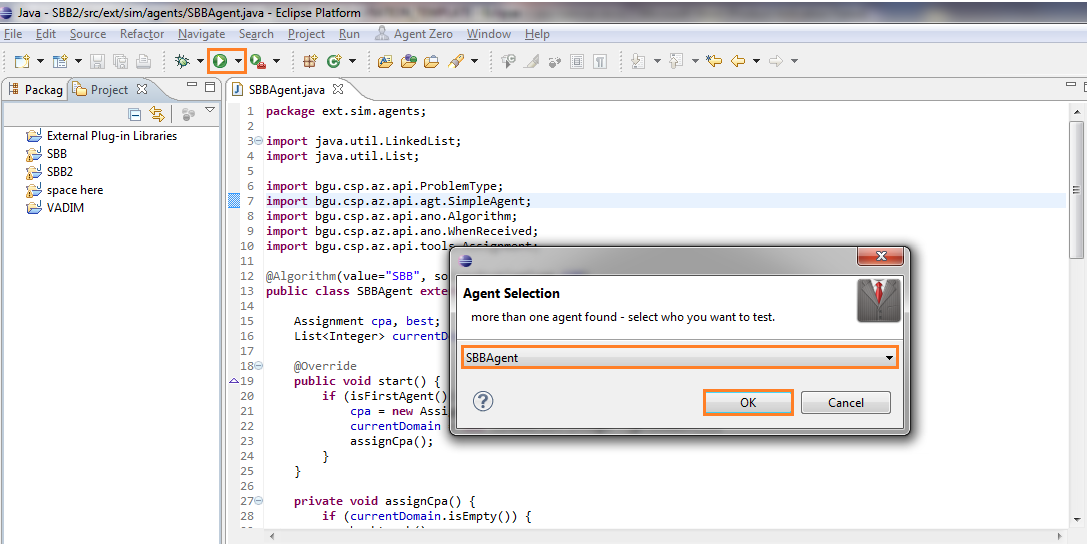
Running and debugging your algorithm in the Eclipse plug in is very simple.

Before running the algorithm, let’s take a look at the test.xml in the root of your project. At this writing, there is no suitable editor for editing the test.xml, but it is a simple xml and you should be able to edit it by hand.  
let’s look at this file structure:  


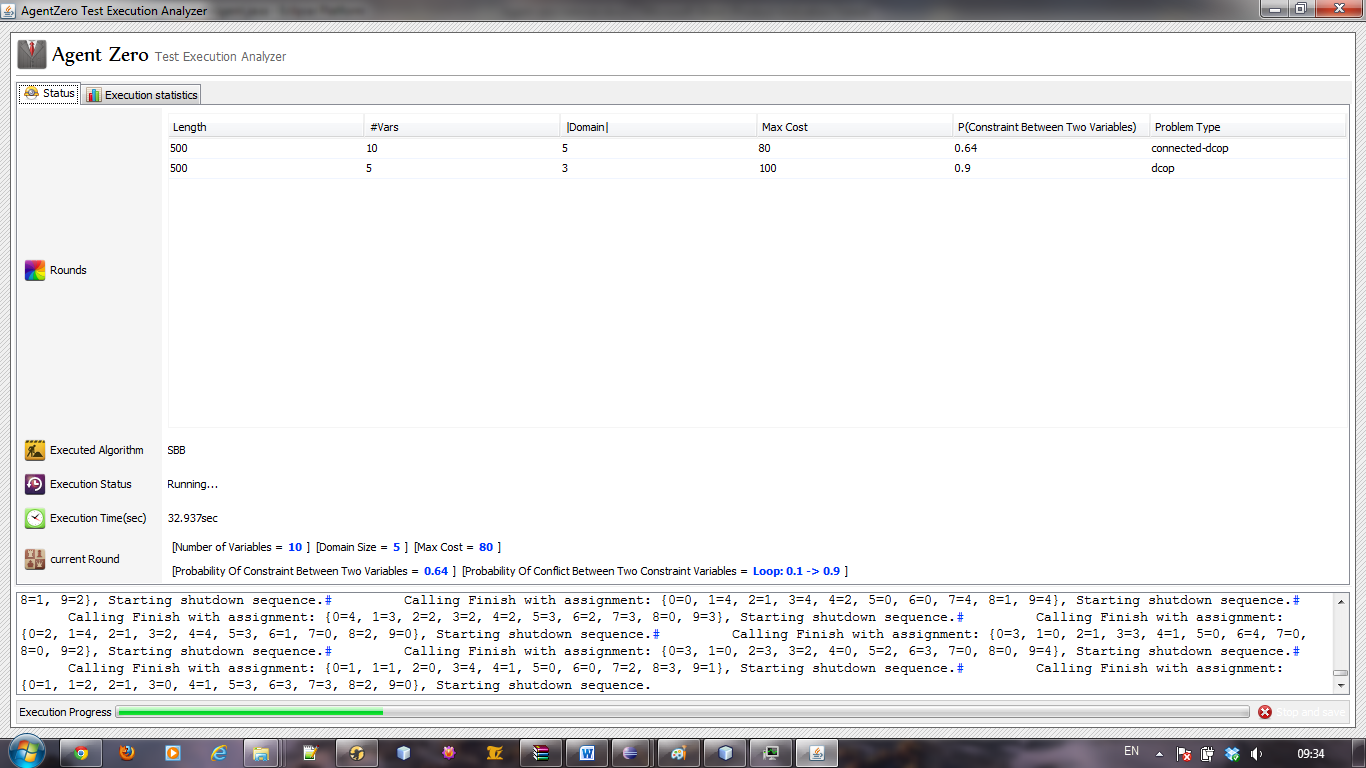
When done with test.xml – close the file and save it. You cannot execute  
while viewing the file.

We will focus on the round node which represents a simple loop of applying your algorithm on random problems of the same type and checking the results (versus trusted algorithm).  
The available attributes:   
**length** – how many problems from this type to run.  
**n** - the number of variables for each generated problem.  
**d** - the domain size for every variable in the generated problem.  
**p1** - probability of conflict between two variables in the generated problem.  
**max-cost** - every constraint cost in the generated problem will be between 0 and this value (for CSP max-cost   
 can be ignored or set to 1).  
**p2-tick, p2-start, p2-end** – p2 is the probability of conflicted values between two constrained variables.   
In a given round, p2 will go from p2-start to p2-end in jumps of p2-tick.  
**type** - the problem type will be significant in the next version.

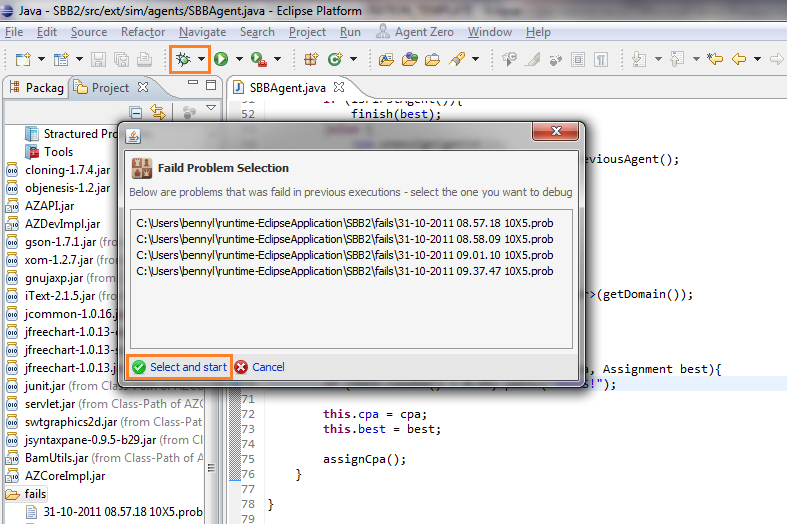
To execute your project press on the “Play” button. If you have several agents in your project, choose the one to run and press “OK”.



This screen summarizing your current execution will appear.



If the algorithm fails on a generated problem, it will be saved.  
When you want to debug a failed problem, press on the “Bug” icon, the saved problems will appear.  
Choose the one you want to debug and press “Select and start”.



Section 2: working with synchronized search

In this section we are going to learn how to write and execute synchronized search algorithms

First we will learn about the synchronized execution mode and the basics that the algorithm implementer needs to know in order to successfully write an algorithm that can be executed in this mode, then we will see how to start an agent in synchronized execution mode.

At the end of this section we will inspect a DSA implementation that is written in agent Zero.

Chapter 1: the synchronized execution mode

Agent zero allow an agent to be executed in synchronized execution mode,

At this mode the agents will work synchronically between “shared ticks”

Each “tick” represents a synchronization point – agents will always receive messages that sent to them during the previous tick, when agents send messages they will arrive to their destination in the next tick

A tick ends when the all the agents finish receiving all the messages that was sent to them during the previous tick (the first tick (=tick 0) is the initializing tick; at this tick the agents “start” function will get called).

The number of ticks that passed since the algorithm started called “system time”

Every time a tick ends the method Agent.onMailBoxEmpty will get called   
you should override this method if you want to get notifications about this event (and most of the time this is what you want)

As always messages are translated to remote procedure calls – you should write the algorithm regularly as you would write any agent zero algorithms (using the @WhenReceived annotation)

If you want to read the current system time you can call the Agent. getSystemTime method – notice that calling this method from within Agent.onMailBoxEmpty will retrieve the last system time (that just means that the function Agent.onMailBoxEmpty is getting called before the time is increased)

That’s about it; you now know everything you need in order to write a synchronized executed agent! The only thing left to know is how to tell Agent Zero that your agent should run in synchronized mode?

It’s also a simple task – just add the following directive to the @Algorithm annotation (the one before the declaration of your agent class) :

searchType=SearchType.*SYCHRONIZED*

That’s it! Your algorithm will now run in synchronized mode.

Most of the time you will want to run a synchronized algorithm on an ADCOP (A-synchronized Distributed Constraints Optimization Problem) if this is the case don’t forget to set the problemType directive in the @Algorithm annotation to ProblemType.*ADCOP* (problemType=ProblemType.*ADCOP*)

Chapter 2: implementation to look at

Let’s take a look on a real ADCOP, Synchronized executed algorithm written in Agent Zero:  
the highlighted lines demonstrates the concepts discussed in the previous page.

@Algorithm(name = "DSA",

searchType = SearchType.SYNCHRONIZED,

problemType = ProblemType.ADCOP)

**public** **class** DSAAgent **extends** SimpleAgent {

**private** Assignment values;

**private** **double** p;

@Override

**public** **void** start() {

values = **new** Assignment();

p = 0.5;

**int** value = random(**this**.getDomain());

**this**.submitCurrentAssignment(value);

send("ValueMessage", value).toNeighbores(**this**.getProblem());

}

@WhenReceived("ValueMessage")

**public** **void** handleValueMessage(**int** value) {

values.assign(getCurrentMessage().getSender(), value);

}

@Override

**public** **void** onMailBoxEmpty() {

**final** **long** systemTime = getSystemTime();

**if** (systemTime + 1 == 20000 && isFirstAgent()) {

finishWithAccumulationOfSubmitedPartialAssignments();

}

**Integer** newValue = calcDelta();

**if** (**Math**.random() > p && newValue != **null**) {

submitCurrentAssignment(newValue);

send("ValueMessage", newValue).toNeighbores(**this**.getProblem());

}

}

**private** **Integer** calcDelta() {

**int** ans = **this**.getSubmitedCurrentAssignment();

**double** delta = **this**.values.calcAddedCost(**this**.getId(), ans, **this**.getProblem());

**double** tmpDelta = delta;

**for** (**Integer** i : **this**.getDomain()) {

**double** tmp = **this**.values.calcAddedCost(**this**.getId(), i, **this**.getProblem());

**if** (tmp < tmpDelta) {

tmpDelta = tmp;

ans = i;

}

}

**if** (delta == tmpDelta) {

**return** **null**;

}

**return** ans;

}

}

Section 3: Nested Agents

Sometimes the algorithm that you will implement will need itself to run other algorithm and work with its results

The best example of this kind of algorithms is the Pseudo Tree algorithms family  
such an algorithm will first need to run some kind of Tree Analyzing algorithm (mostly DFS), get its result and use that in its execution

Nested agents feature designed to solve this need in a simple but powerful way

The following section will cover this feature, the usage of a ready-made nested agents and how to build your own nested agent tool

Chapter 1: nested agent usage basics

Let’s assume that we want to write an algorithm from the Pseudo Tree family – for example DPOP

Let’s see how the code for agent of such an algorithm should look like:

DPOPAgent should first run Distributed DFS to map the problem pseudo tree

The Distributed DFS (DDFS) algorithm should itself first run Distributed Leader Selection Algorithm (DLSA)

Then when the DLSA will complete the DDFS will use its results in order to produce a pseudo tree,

When he will complete the DPOP agent will use the resulted tree in order to produce an assignment.

This means that the DPOP Agent should contain a logic of 2 more algorithms,  
it should also contain a logic that produce synchronization with other agents that running those algorithms so all of them will move from algorithm to algorithm as a group or else messages that belongs to one algorithm will get sent to other

This is also not so modular – if someone else already writes a DLSA you will not be able to use it without fully modifying your agent and copy its code in it

And we not even start to talk about debugging such an algorithm – if there is a problem in which algorithm it happened?

The solution: **Nested Agents**

Nested agents feature allows you to separate your algorithm code into different agents and then combine them anyway you like for example when the previously discussed DPOP Agent will want to calculate a DFS Tree all he have to do is:

Tree tree = new PseudoTree();  
tree.calculate(this).andWhenDoneDo(new Continuation() {  
 @Override  
 public void doContinue() {  
 //WHAT TO DO WHEN TREE CALCULATION IS OVER  
 }});

What just happened?

Tree is a tool that contains a nested agent named DFSAgent   
\* this type of tools called nesteable tools

When calling calculate(this) the agent is actually transforming to DFSAgent and start its execution  
it’s like the DFSAgent nested inside DPOPAgent (hence the name..)

The DFSAgent messages will arrive only to other DFSAgents so DPOP is safe from getting unknown messages accidently  
the DFSAgent itself will have the code:

LSA lsa = new PseudoTree();  
lsa.calculate(this).andWhenDoneDo(new Continuation() {  
 @Override  
 public void doContinue() {  
 //WHAT TO DO WHEN LEADER SELECTION ALGORITHM IS OVER  
 }});

And so now the DFSAgent will get nested and the LSAAgent will became the active Agent,

You will notice that we are not actually instantiating a new agent, instaed we are instantiating the result of the wanted algorithm and then calling calculate – this reflects the algorithm need – the DPOP Agent don’t care how the tree will get calculated he only care about the final result.

You will also notice that we supplying the nesteable tool with an implementation of a continuation object

This is just java way to send a callback function (actually a callback closure…)

The last thing that you (as a user of nesteable tool) have to know is what exactly happened when you running calculate –

First the calculation will start the nested agent (calling its start function) then you will return back to the calling function, finish it and from now on (until the nested agent finish) your agent will behave as the nested agent.

When the nested agent will finish your continuation function will get called

This is very important to understand that the calculate method is non blocking one! The calling agent will have to complete its method (start / message handling method) and only then he will became the nested agent – so don’t put any code that uses the nesteable tool after the calculation – put it inside the continuation function.

Chapter 2: writing your own nesteable tool

We saw in the previous chepter how to use ready made nesteable tool, but what if we want to create our own implementation of Leader Selection Algorithm?

In that case we will want to build new nesteable tool – this is fairly simple:

Just create a new class (lets assume you called it MyLSA) with all the needed fields for the result of this algorithm (in that case we will have a field: selectedLeader)

Next you should write an inner class that extend SimpleAgent (lets assume you called it MyLSAAgent)

This class will have access to the field selectedLeader as it is an inner class of MyLSA  
now write in this class the LSA logic like any other agent

Finally implements the NesteableTool.createNestedAgent function to return new MyLSAAgent();

That’s it, here is a simple template following those steps:

**public** **class** MyLSA **extends** NesteableTool {

**private** **int** selectedLeader;

@Override

**public** **Integer** getSelectedLeader() {

**return** **this**.selectedLeader;

}

@Override

**protected** SimpleAgent createNestedAgent() {

**return** **new** MyLSAAgent();

}

**public** **class** MyLSAAgent **extends** SimpleAgent {

@Override

**public** **void** start() {

…

}

.  
 .

.

@WhenReceived("SomeMessage")

**public** **void** handleSomeMessage(**int** leader) {

selectedLeader = leader;

finish();

}

}

}