X10-based Agent Simulation on Distributed Infrastructure (XASDI) Application Development Guide

v0.9.0

IBM Research – Tokyo

2016/03/08

What is X10-based Agent Simulation on Distributed Infrastructure (XASDI)?

- A multi-agent agent simulation platform on top of the X10 language
- Java applications connected to ZASE can also attach to this base.

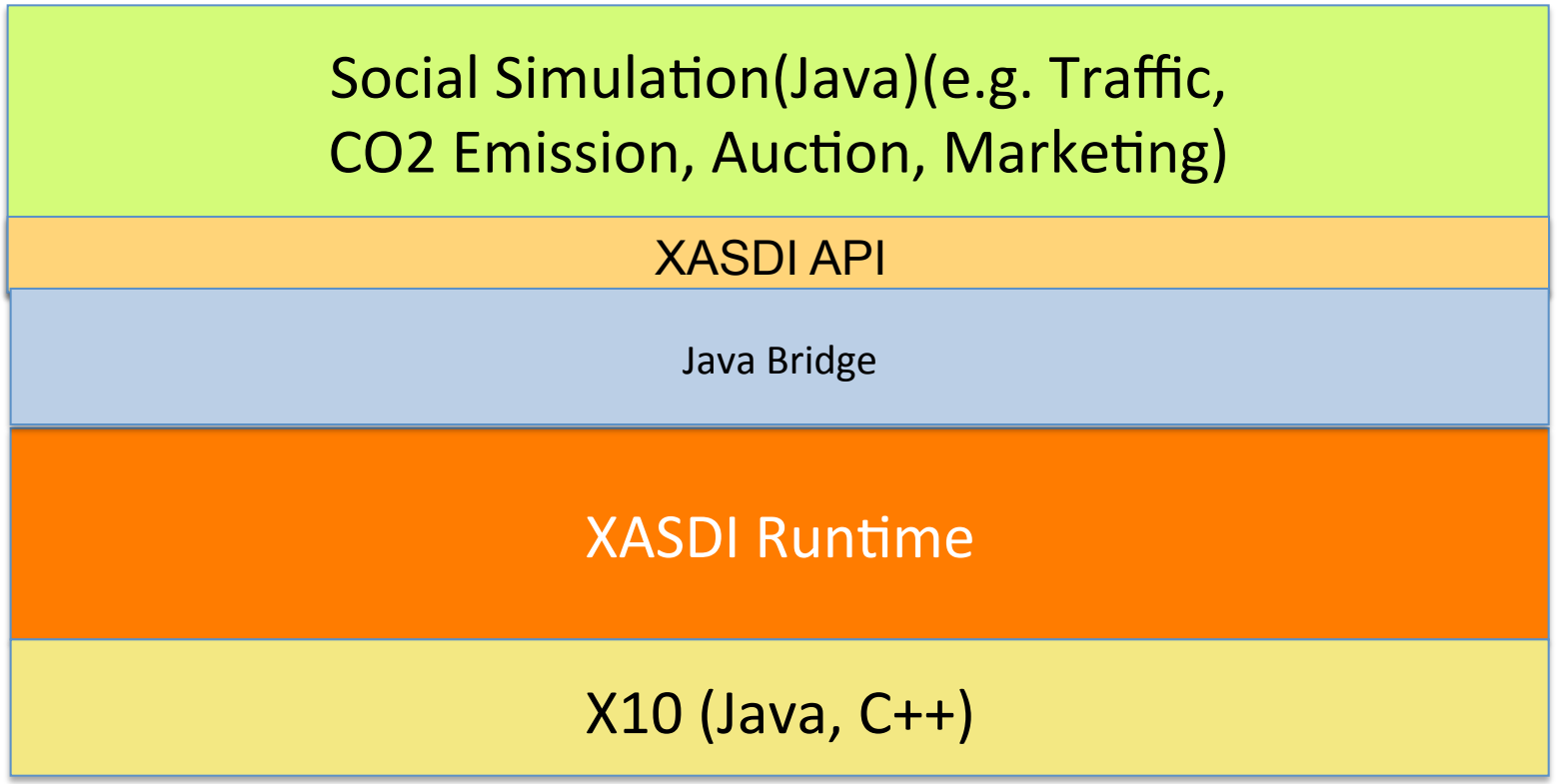
XASDI

- X10-based Distributed Agent Simulation Platform
  - X10 is the state-of-the-art PGAS (Partitioned Global Address Space) language that brings high productivity when implementing highly parallel and distributed applications on post-peta or exascale machines
    - X10 provides the functionality that can seamlessly integrate with legacy applications written in Java or C++.
- Programming Model
  - The agent programming model of XASDI is derived from our ZASE [Yamamoto, AAMAS2007] simulation platform
  - XASDI provides compatible API interface of ZASE to developers.

Gaku Yamamoto, et.al, “A Platform for Massive Agent-based Simulation and its Evaluation”, AAMAS 2007

XASDI Software Stack

- The following diagram illustrates the software stack of XASDI and its applications.



Directories

- You can see the following sub directories under the “xasdi” directory.
  - api (API Javadoc)
  - doc (Contains this file)
  - jar (Simulation base jar file)
  - MySample (Eclipse project directory of tiny sample program)
  - src (XASDI source code)

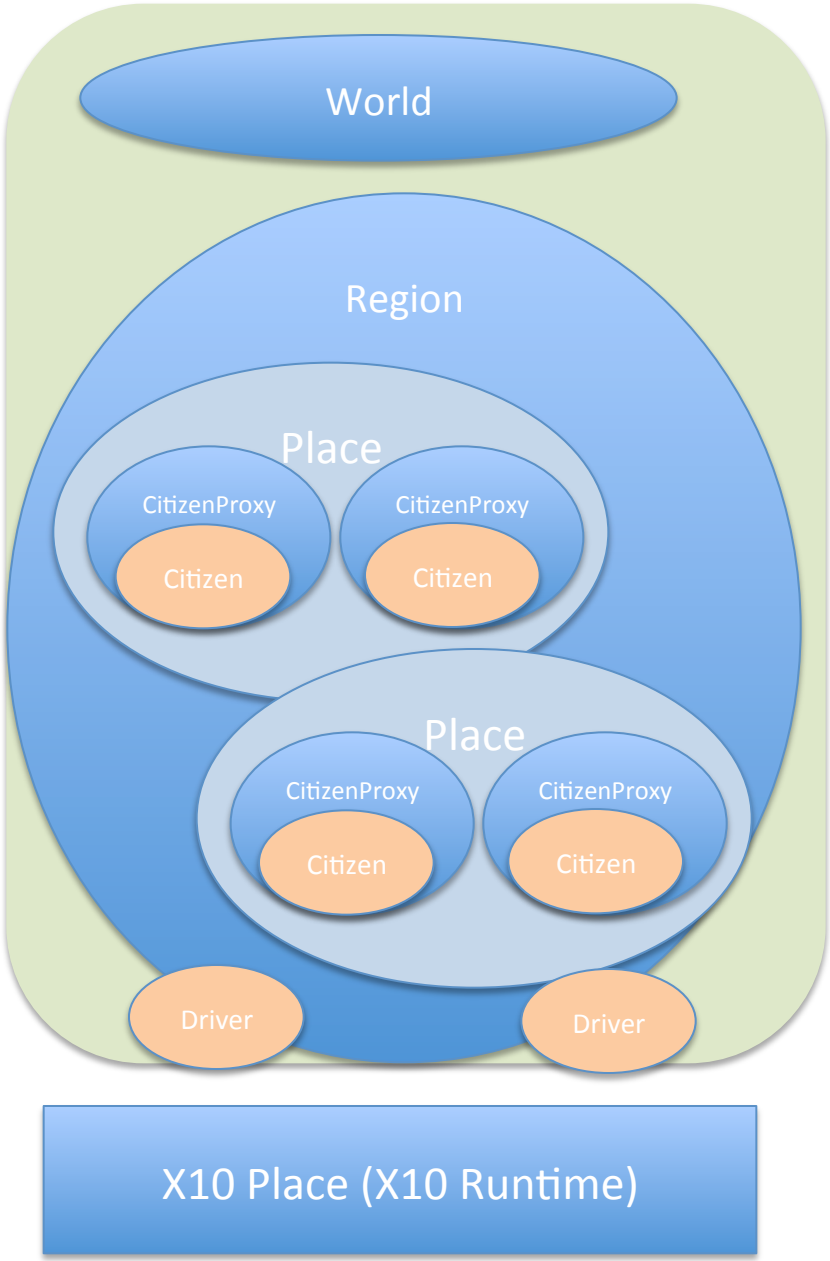
Installation

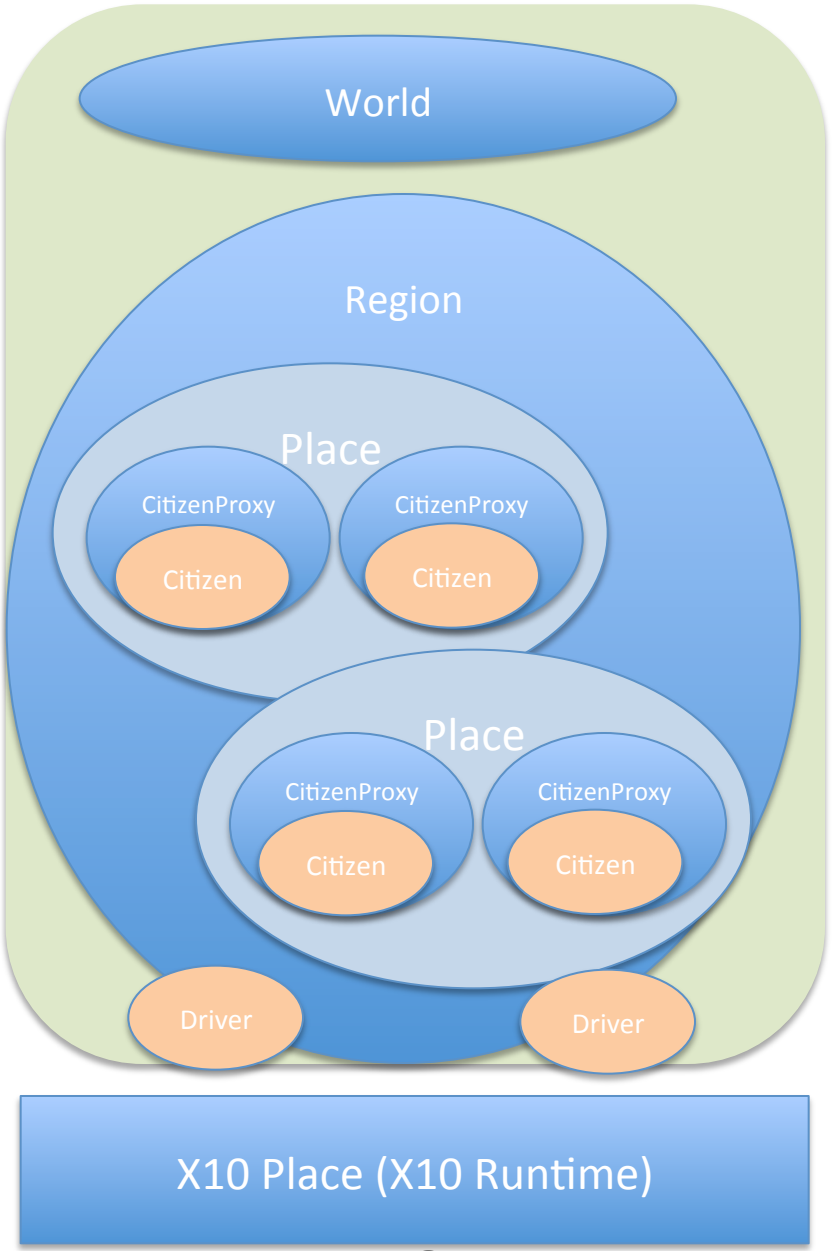
- Setup X10
  - Download pre-built binary of X10 2.5.4 for selected platform from http://x10-lang.org/releases/x10-release-254
  - After downloading, untar and see the INSTALL file for further instructions.

Class information for developing applications on XASDI

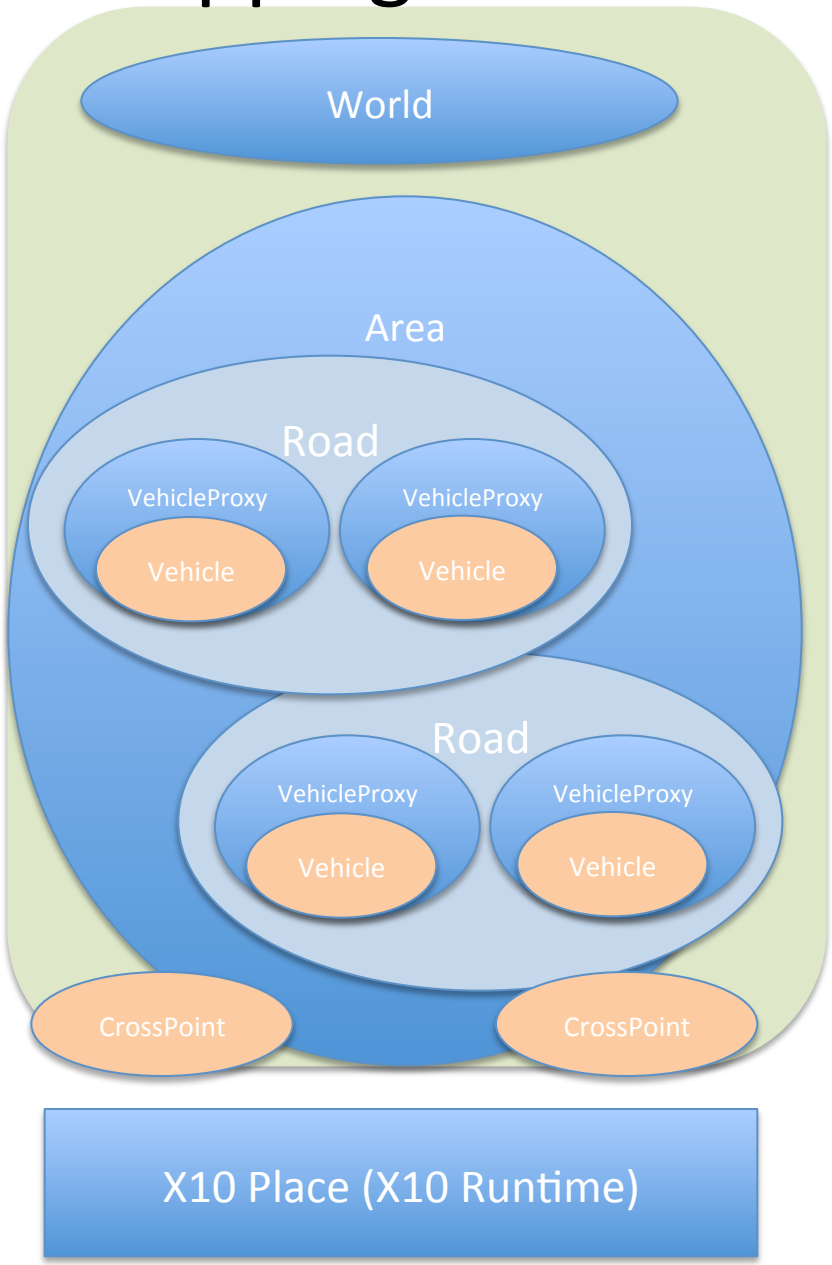
XASDI
API CLASSES

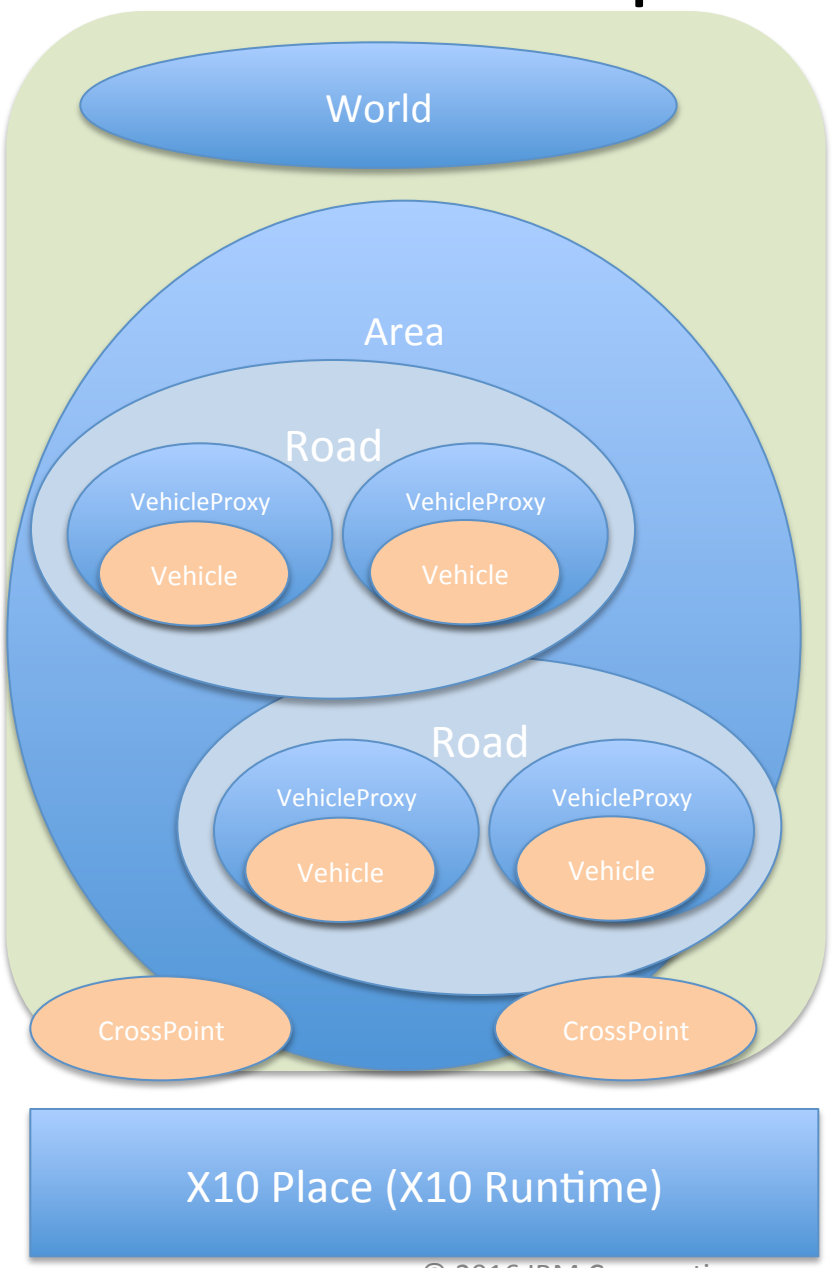
Overview





Mapping to Traffic Simulation for example





Important Classes

Running simulation

- LauncherProxy
- Region
- Driver
- Place
- CitizenProxy・Citizen
- World

Managing messages

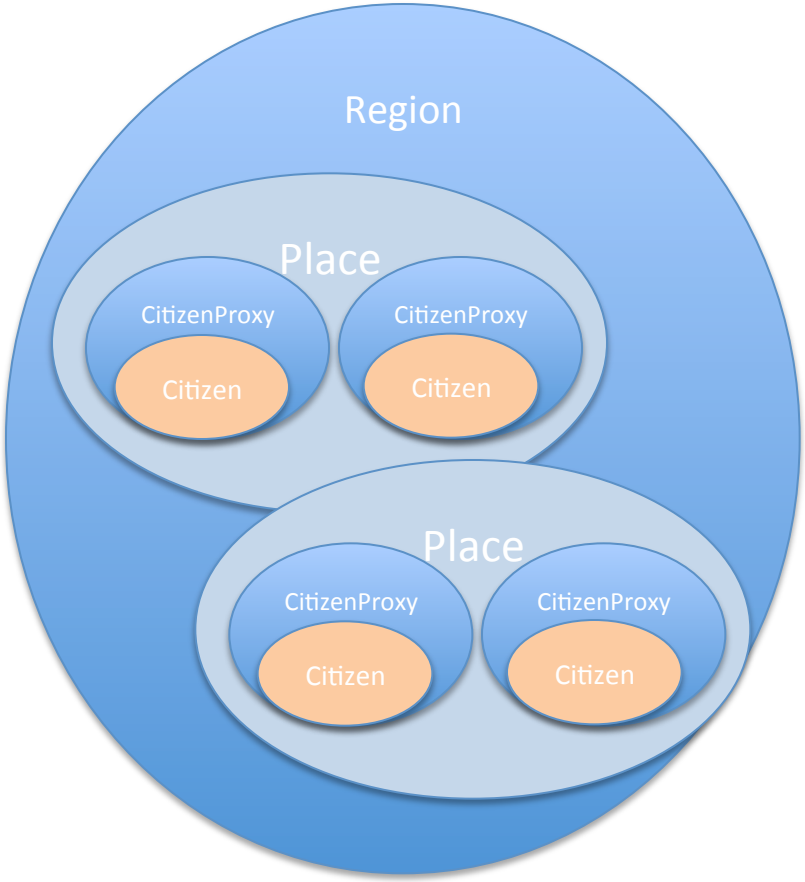
- Message
- MessageList
- MessageQueue
- MessageRepository
- MessageResolver

LauncherProxy

- A class for launching an agent simulation itself after creating relevant Regions, Drivers, Places, Citizens, World etc.
- The first Java object to be created.
  - Assign agent ID to agent objects.
  - Create important objects (Launcher and Region).
- The created agents and message objects are provided to the runtime of XASDI through Java APIs (xasdi_bridge).

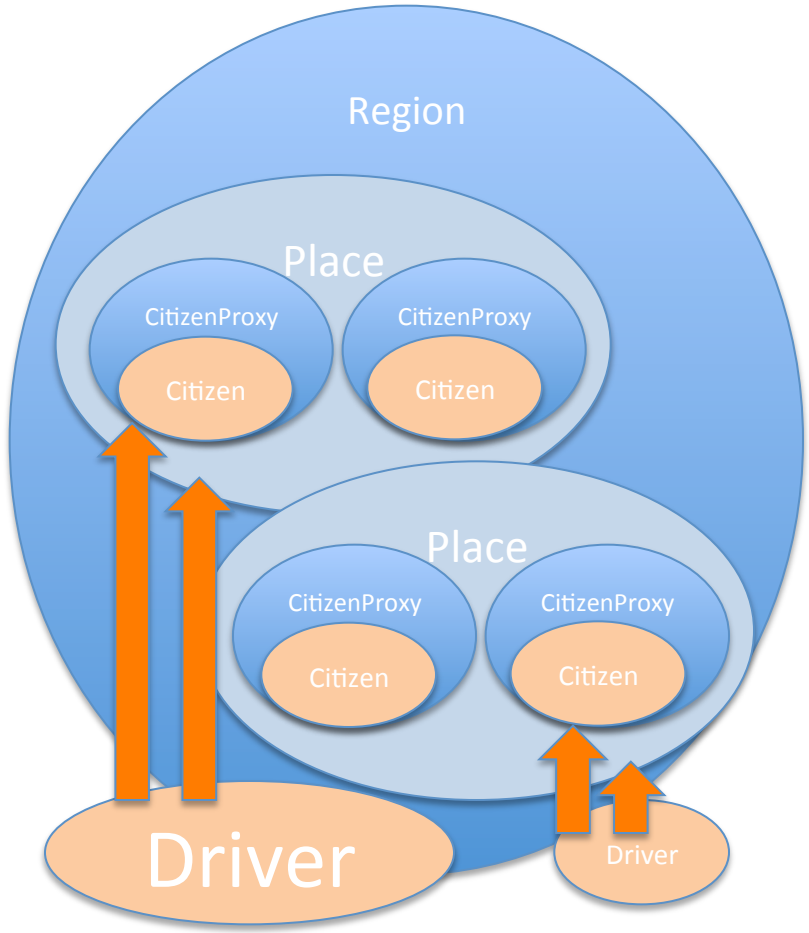
Region

- Possess agents (CitizenProxy and Driver), groups of agents (Place), mapping about objects and these IDs.
  - These objects are added, removed, and referenced at this class.



Driver

- A Driver class triggers the behaviors of CitizenProxies and Places
- Handle and execute Citizen objects.
  - At first handle Place objects, and then move **CitizenProxy objects.**
  - Driver objects can send and receive messages between other Driver and CitizenProxy objects.



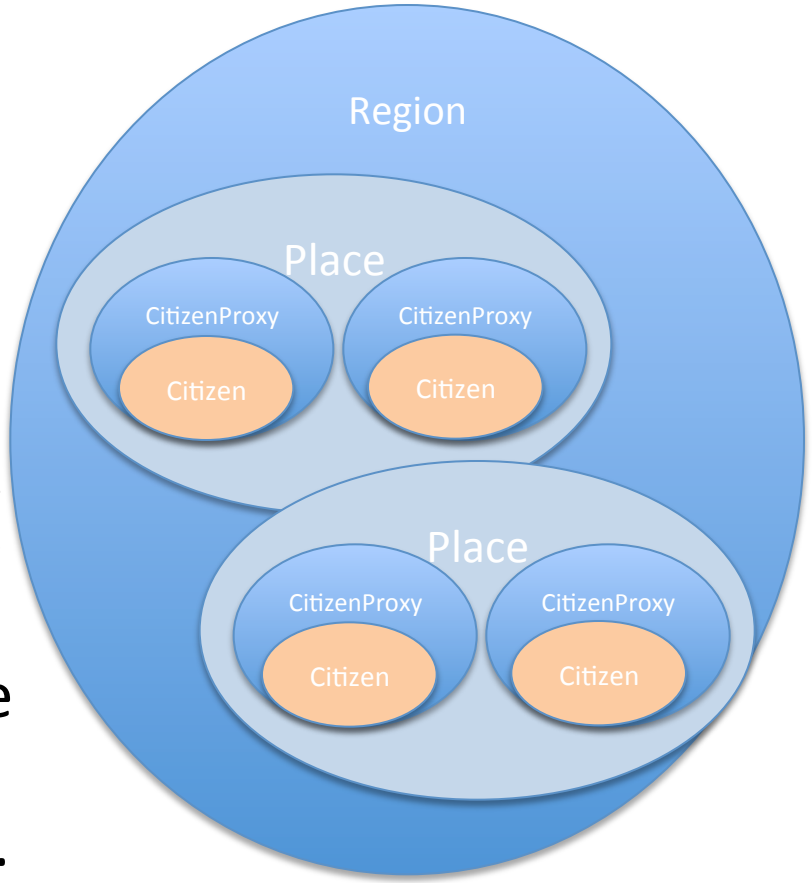
Place

- Group of CitizenProxy objects (Not X10 Place).
  - They have CitizenID and manage CitizenProxy objects.
  - CitizenID needs not to be a contiguous identifier.

CitizenID	CitizenProxy
0	obj1
1	obj2
…	…
N	objN

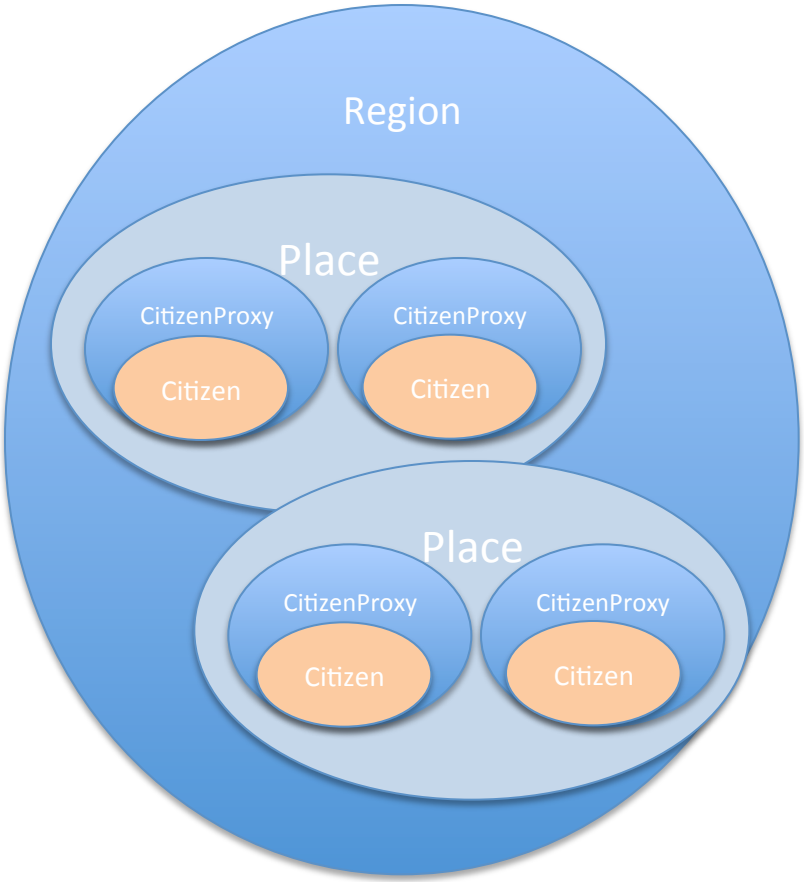
CitizenProxy

- An agent class for sending and receiving messages.
  - A list of messages with source and destination identifiers sent from this class will delivered to **MessageResolver**
  - Behavior when receiving messages is defined at subclass (It can ignore received messages).
- Each CitizenProxy object has one Citizen object handling their agent behavior or business logic.



Citizen

- A CitizenProxy object passes messages to the corresponding Citizen object that executes their agent behavior via the “execute” method.
- The “execute” method returns a “Message” as a response.



World

- It is used for maintaining global variables such as .. .
  - Set of log writers, simulation step counter, phase counter, mapping about agent and X10 Place ID...
  - Most variables can be referred using static methods.
    - e.g. World.world().getLogger();

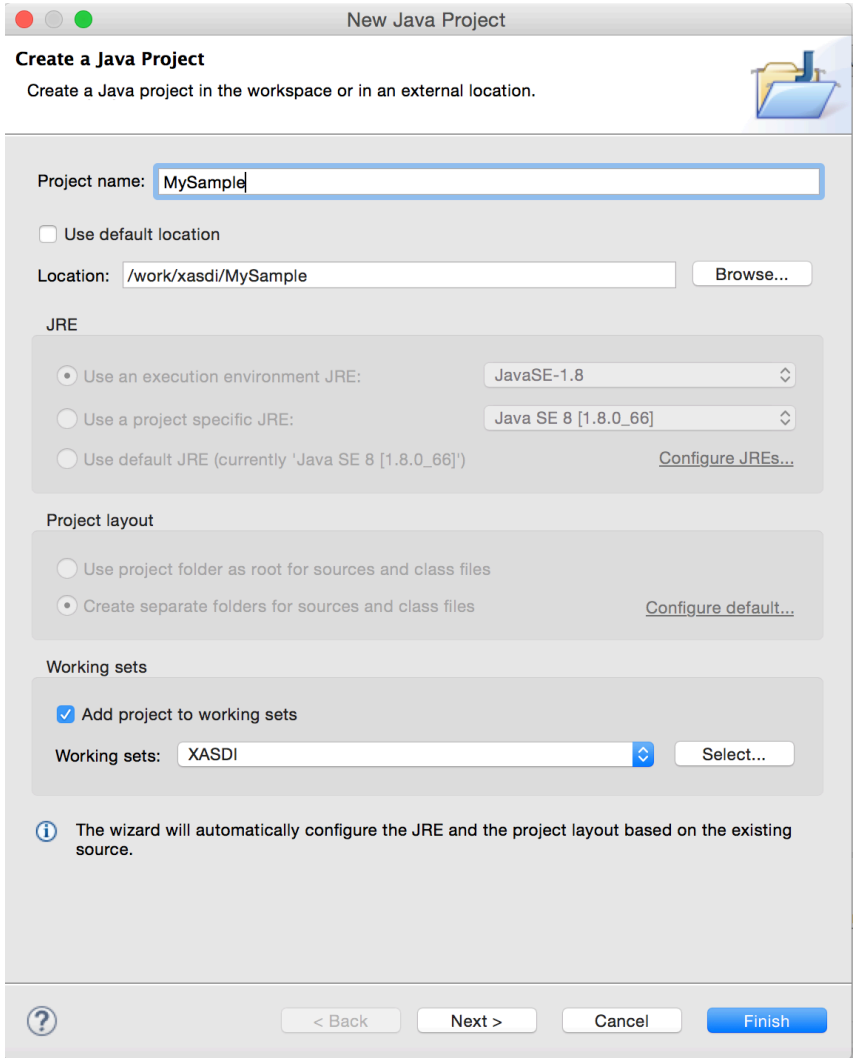
DEVELOPING SAMPLE APPLICATION WITH XASDI

Sample Program : MySample

- **Overview of Sample Simulation Program**
  - 10 agents send messages to other agents with unicast (10x10 messages in total) or broadcast (10 messages) style.
  - Simulation steps is 4
    - Step 0: Each agent sends an attribute message to the agent itself in order to change the attribute of the agent.
    - Step 1: Each agent sends an individual message to the agent itself in order to output the attribute value.
    - Step 2: The region sends broadcast messages to 10 agents (citizens).
    - Step 3: Each agent sends mutual messages to all the agents in an N to N manner.
- Project directory can be found at xasdi/MySample

Create Eclipse Project

- Launch Eclipse
- Locate the project directory to “xasdi” directory.
  - You should set work directory “xasdi” as well

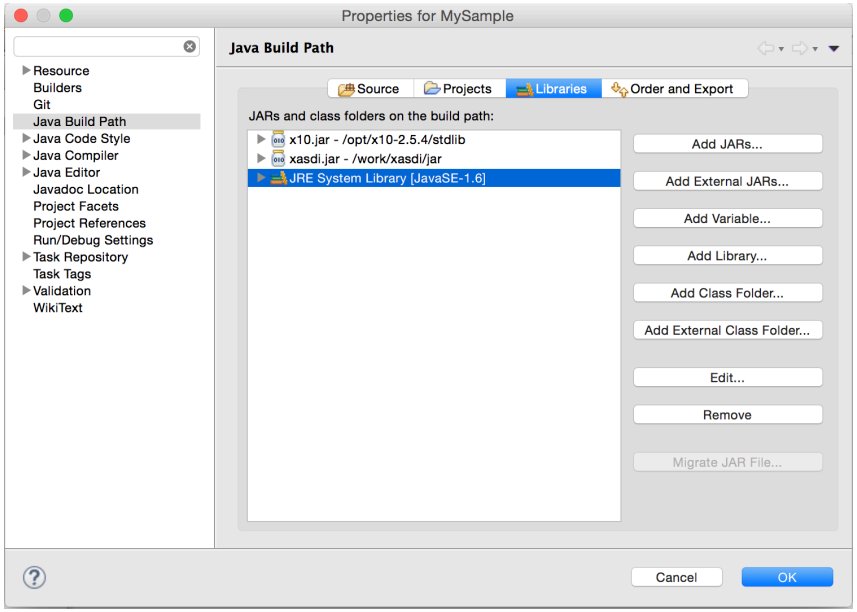


Required jar files

- (X10-2.5.4 install directory)/stdlib/x10.jar
  - X10 standard library for Java
- xasdi/jar/xasdi.jar
  - Base library of XASDI

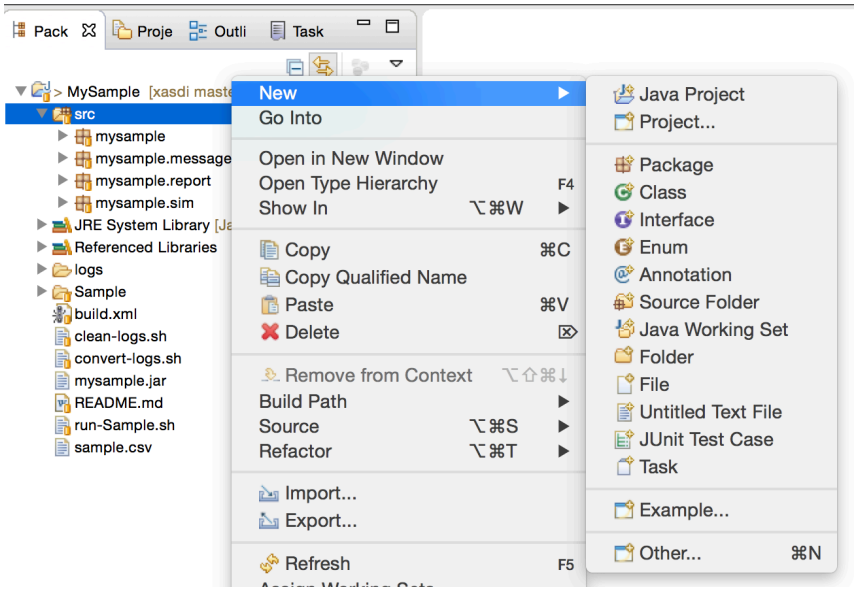
Import jar files

- Add required jar files into class path.
- ”Properties” → “Java Build Path” → ”Add External JARs ...”
- Add two jars in the previous slide.



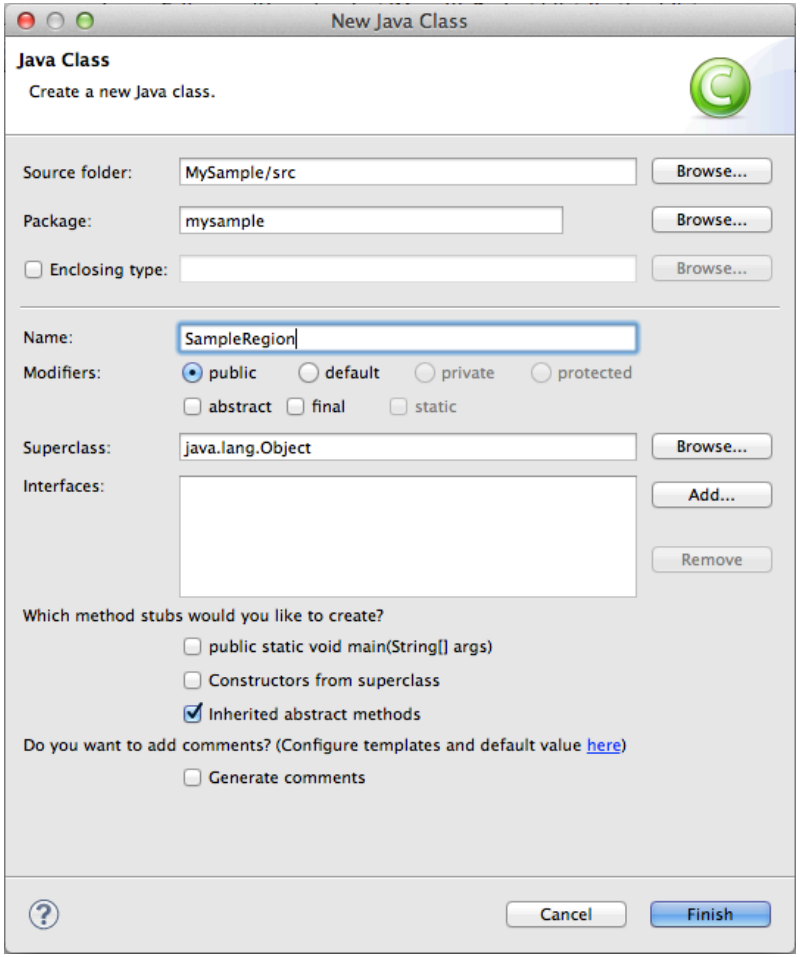
Add Java packages

- Select “src” directory.
- “new” → “Package”
- Create 4 packages.
  - mysample
  - mysample.message
  - mysample.report
  - mysample.sim



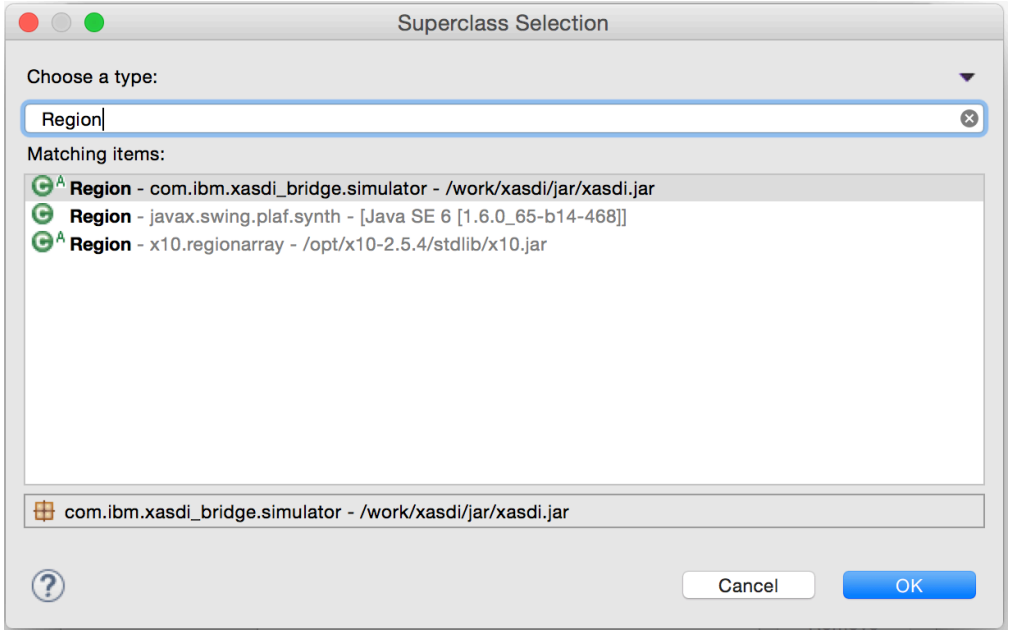
Add Java classes listed in p.27-p.29

- Add Java classes listed in p.27-p.29 into packages.
- Select a package → “new” → “Class”
- Input class name.



Selecting appropriate Inheritance classes

- Some classes inherit parent class in XASDI.
- Click “Browse...“ button, input super class and select proper class.



Write class content

- After clicking the “Finish” button, a constructor and methods are automatically shown.
- You have to add your source codes to these methods.
- After adding all Java classes, complement, add, and modify source code.
- * You can remove error marks alerted in Eclipse by importing classes in the simulation base (xasdi.jar and x10.jar).

Classes for MySample Application

- SampleRegion extends Region
- SampleDriver implements Driver
- SamplePlace extends Place
- SampleCitizen extends Citizen
- SampleCitizenProxy extendsCitizenProxy
- SampleCitizenFactory implements CitizenFactory
- SampleCitizenProxyFactory implements CitizenProxyFactory
- SampleResolver implements MessageResolver

Classes (mysample.message)

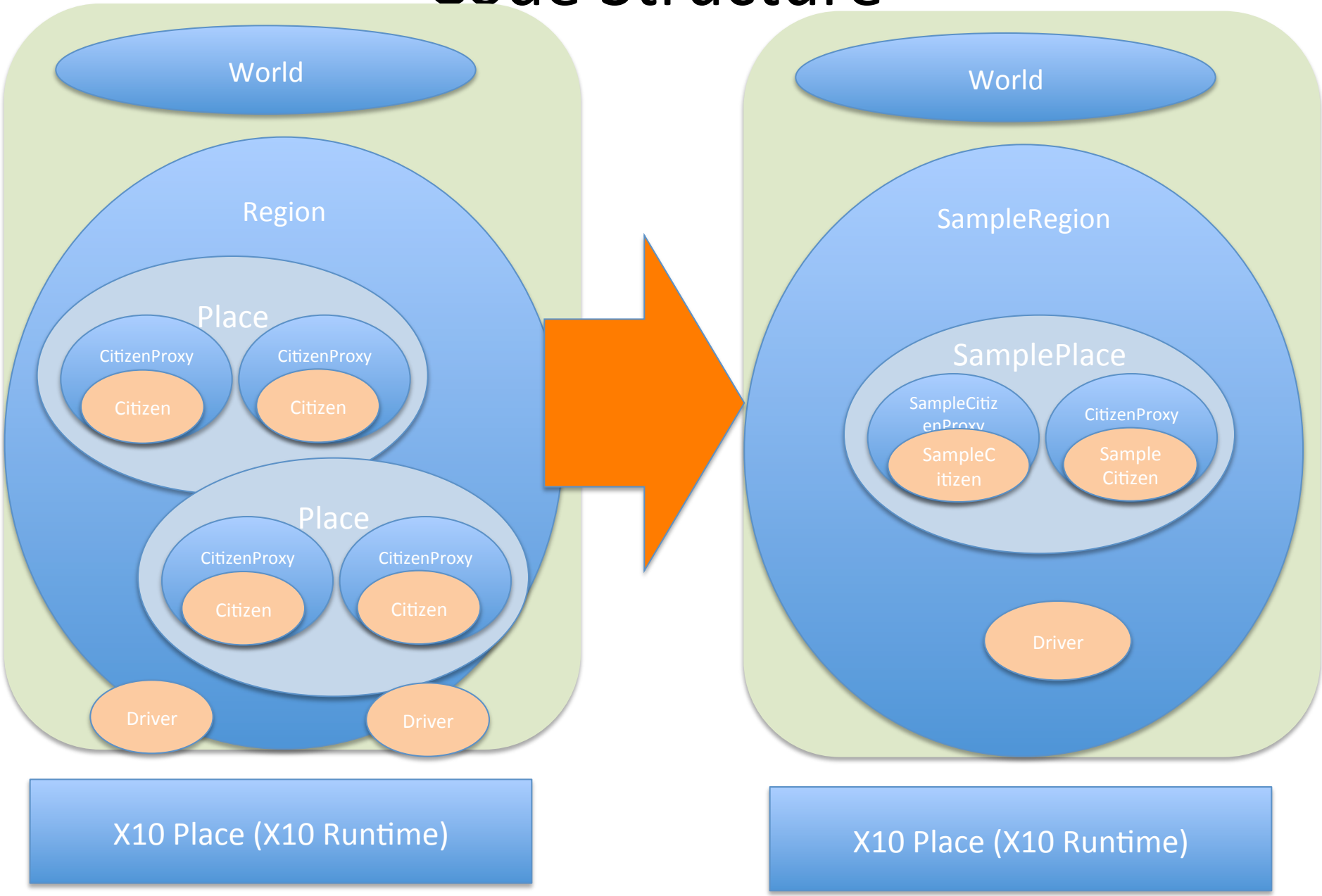
- SampleMessage extends Message
- SetAttributeMessage extends SampleMessage
- IndividualMessage extends SampleMessage
- BroadCastMessage extends SampleMessage
- DirectionMessage extends SampleMessage
- MutualMessage extends SampleMessage

Classes (mysample.report, sim)

- package mysample.report
  - SampleLogConverter
- package mysample.sim
  - SampleLauncher implements Launcher

Sample Code

Code Structure



SampleRegion

```
package mysample;

import com.ibm.xasdi_bridge.citizen.CitizenFactory;
import com.ibm.xasdi_bridge.message.MessageQueue;
import com.ibm.xasdi_bridge.simulator.Region;

public class SampleRegion extends Region {

	public static final Integer FACTORYID = new Integer(0);

	public SampleRegion(long id) {
		super(id);
	}

	@Override
	protected CitizenFactory getCitizenFactory() {
		return super.getCitizenFactory(SampleRegion.FACTORYID);
	}

	@Override
	public void sendMsgQ(MessageQueue msgq) {
		// do nothing
	}

}
```

- You have only to define a constructor and CitizenFactory.
- Parent class (Region) will manage all Citizen, Place and Driver agents.

SampleDriver

```
package mysample;

import java.util.HashMap;
import mysample.message.*;

import com.ibm.xasdi_bridge.CitizenID;
import com.ibm.xasdi_bridge.message.MessageList;
import com.ibm.xasdi_bridge.simulator.CitizenProxy;
import com.ibm.xasdi_bridge.simulator.Driver;
import com.ibm.xasdi_bridge.simulator.Region;

public class SampleDriver implements Driver {

	SamplePlace place;
	SampleRegion region ;

	public SampleDriver(SamplePlace place, SampleRegion region) {
		this.place = place;
		this.region = region;
		for(int i=0; i<10; i++){
			int args = 0;
			CitizenID cid = new CitizenID(i);
			SampleCitizenProxy proxy =
(SampleCitizenProxy)region.createCitizen(SampleRegion.FACTORYID, cid, args);
			if(proxy != null){
				Region.getCitizenSet(0).add(cid);
			}
		}
	}
```

- The SampleDriver is responsible for creating 10 agent (SampleCitizenProxy) objects in a constructor.
  - Agent ID is from 0 to 9.
- In the execute method, the SampleDriver object kick offs the behavior of the CitizenProxy object.

SampleDriver

```
@Override
public void execute(long time) {
        HashMap<Long, CitizenProxy> proxies = region.getCitizenProxies();
        if(time == 0){
                for(long i=0; i<proxies.size(); i++){
                        SampleCitizenProxy proxy = ((SampleCitizenProxy)proxies.get(i));
                        proxy.setAttribute(proxy.getID() , (int) i);
                }
        }else if(time == 1){
                for(long i=0; i<proxies.size(); i++){
                        SampleCitizenProxy proxy = (SampleCitizenProxy)proxies.get(i);
                        proxy.individual(proxy.getID());
                }
        }else if(time == 2){
                BroadCastMessage msg = new BroadCastMessage(place.getID());
                region.sendMessage(msg);
        }else if(time == 3){
                for(long i=0; i<proxies.size(); i++){
                        SampleCitizenProxy from = (SampleCitizenProxy)proxies.get(i);
                        for(long j=0; j<proxies.size(); j++){
                                SampleCitizenProxy to = (SampleCitizenProxy)proxies.get(j);
                                MutualMessage msg = new MutualMessage(from.getID(), to.getID());
                                from.mutual(msg);
                        }
                }
        }
}
```

The “execute” method in SampleDriver

- Each agent (CitizenProxy) sends 4 types of messages at the “execute” method.
  - Step 0: Sends a message to itself and changes its attribute.
  - Step 1: Sends a message to itself, but only writes log file without changing its attribute.
  - Step 2: Broadcasts one message to all agents.
  - Step 3: Send different messages to each agent.

- (Tips) It is not required to change other methods than the “execute” method.

SamplePlace

```
package mysample;

import com.ibm.xasdi_bridge.simulator.Place;
import
com.ibm.xasdi_bridge.simulator.Region;

public class SamplePlace extends Place {

	private static final long
		serialVersionUID = 1L;

	public SamplePlace(Region region, long id) {
		super(region, id);
	}

}
```

- In this sample application, all agent are grouped into one Place so that the Place class is not used.
- If you would like to use a Place class for grouping multiple places, you can have variables in the Place class.

SampleCitizen

```
package mysample;

import mysample.message.*;

import com.ibm.xasdi_bridge.CitizenID;
import com.ibm.xasdi_bridge.citizen.Citizen;
import com.ibm.xasdi_bridge.citizen.CitizenSet;
import com.ibm.xasdi_bridge.log.Log;
import com.ibm.xasdi_bridge.log.Logger;
import com.ibm.xasdi_bridge.Message;
import com.ibm.xasdi_bridge.simulator.Region;
import com.ibm.xasdi_bridge.simulator.World;

public class SampleCitizen extends Citizen {
        int attribute;

        public SampleCitizen(CitizenID cid) {
                super(cid);
        }
        @Override
        public void onCreation(Object arg) {
                attribute = 0;
                CitizenSet set = Region.getCitizenSet(0);
                set.add(getID());
        }
        @Override -→ Obsolete
        public void onExecute(long time, int phase) {
                // do nothing
        }
```

- SampleCitizen objects execute its core agent behavior methods according to received messages' types.
  - Create reply message
  - Write Log
- (Tips) The "onExecute" method is now obsolete, instead you can write your agent behavior in the *onMessage* method.

SampleCitizen

```
@Override
public Message onMessage(Message msg) {
        SampleMessage reply = (SampleMessage)msg;
        switch(msg.getType()){
                case SampleMessage.INDIVIDUAL:
                reply = individual((IndividualMessage)msg); break;
                case SampleMessage.BROADCAST:
                reply = broadCast((BroadCastMessage)msg); break;
                case SampleMessage.ATTRIBUTE:
                reply = null; attribute((SetAttributeMessage)msg); break;
                case SampleMessage.DIRECTION:
                reply = null; direction((DirectionMessage)msg); break;
                case SampleMessage.MUTUAL:
                reply = mutual((MutualMessage)msg); break;
        }
        return reply;
}
```

- When a message is arrived, “onMessage” method is called.
  - This method is create a reply message (if any) and return.

SampleCitizen

```
private IndividualMessage individual(IndividualMessage msg){
        IndividualMessage reply = new IndividualMessage(1);
        int value = attribute;
        reply.selected = value;
        this.writeLog("I have received an individual message", value);
        return reply;
}

private BroadCastMessage broadCast(BroadCastMessage msg){
        BroadCastMessage reply = new
                BroadCastMessage(msg.getGroupID());
        int value = BroadCastMessage.VALUE;
        this.writeLog("I have received a broadcast message", value);
        return reply;
}

private void attribute(SetAttributeMessage msg){
        this.attribute = msg.attribute;
        this.writeLog("I have received a setattribute message", attribute);
}
```

- The methods - listed in the left hand demonstrate 3 different agent behaviors, unicast (individual), broadcast, and changing the agent attribute.
- You can use the proprietary logging methods, but it is optional.

SampleCitizen

```
private void direction(DirectionMessage msg){
        int cid = (int)msg.getReceiverID().getLocalID();
        this.writeLog("I have received a direction message", cid);
}

private DirectionMessage mutual(MutualMessage msg){
        int value = MutualMessage.VALUE;
        this.attribute = value;
        this.writeLog("I have received a mutual message", value);
        CitizenID dest = msg.getSenderID();
        DirectionMessage reply = new DirectionMessage(dest);
        return reply;
}

private void writeLog(String str, int value){
        try{
                Logger logger = World.world().getLogger();
                Log log = logger.getFreeLog(0);
                log.setInt(0, value);
                log.setString(1, str);
                log.write();
        } catch(Exception e){
                e.printStackTrace();
        }
}
```

- When a message is received, message information is written to common log files.

SampleCitizenProxy

```
package mysample;

import mysample.message.*;

import com.ibm.xasdi_bridge.CitizenID;
import com.ibm.xasdi_bridge.Message;
import
com.ibm.xasdi_bridge.simulator.CitizenProxy;
import com.ibm.xasdi_bridge.simulator.Region;

public class SampleCitizenProxy extends
CitizenProxy {

        public SampleCitizenProxy(Region region) {
                super(region);
        }
```

- SampleCitizenProxy can send messages to SampleRegion class.
  - These objects has a pointer of Region object.

SampleCitizenProxy

```
public void individual(CitizenID id){
        IndividualMessage msg =
                new IndividualMessage(0);
        msg.setSenderID(id);
        sendMessage(msg);
}

public void setAttribute(CitizenID id ,int i){
        SetAttributeMessage msg =
                new SetAttributeMessage(i);
        msg.setSenderID(id);
        sendMessage(msg);
}

public void direct(CitizenID dest, DirectionMessage msg){
        msg.setReceiverID(dest);
        sendMessage(msg);
}

public void mutual(MutualMessage msg){
        sendMessage(msg);
}
```

- It has “sendMessage” method to send a message.
- These 4 methods are called according to the message type.

SampleCitizenProxy

```
@Override
public void receiveMessage(Message msg) {
        Message reply = citizen.onMessage(msg);
        sendMessage(reply);
}
```

- This class instance invokes the Citizen object in order to send a reply message after receiving an incoming message.

SampleCitizenFactory

```
package mysample;

import com.ibm.xasdi_bridge.CitizenID;
import com.ibm.xasdi_bridge.citizen.Citizen;
import
com.ibm.xasdi_bridge.citizen.CitizenFactory;

public class SampleCitizenFactory implements
CitizenFactory {

        @Override
        public Citizen newInstance(CitizenID id) {
                return new SampleCitizen(id);
        }

}
```

- When “newInstance” method is called, one SampleCitizen object is instantiated.

SampleCitizenProxyFactory

```
package mysample;

import com.ibm.xasdi_bridge.simulator.CitizenProxy;
import
com.ibm.xasdi_bridge.simulator.CitizenProxyFactory;
import com.ibm.xasdi_bridge.simulator.Region;

public class SampleCitizenProxyFactory implements
CitizenProxyFactory {

	@Override
	public CitizenProxy newInstance(Region region) {
		return new SampleCitizenProxy(region);
	}

}
```

- When “newInstance” method is called, one SampleCitizenProxy object is instantiated.
  - SampleCitizen and SampleCitizenProxy should be created at the same time.

SampleResolver

```
package mysample;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import mysample.message.BroadCastMessage;
import mysample.message.DirectionMessage;
import mysample.message.IndividualMessage;
import mysample.message.MutualMessage;
import mysample.message.SetAttributeMessage;

import com.ibm.xasdi_bridge.CitizenID;
import com.ibm.xasdi_bridge.Message;
import com.ibm.xasdi_bridge.citizen.CitizenSet;
import com.ibm.xasdi_bridge.citizen.MessageResolver;
import com.ibm.xasdi_bridge.simulator.Region;
```

- A message resolver is in charge of resolving a list of destination agents (CitizenProxy) from a given message
- Import all message classes and some parent classes.

SampleResolver

```
public class SampleResolver implements MessageResolver {

        @Override
        public Collection<CitizenID> resolve(Message msg) {
                if(msg instanceof BroadCastMessage){
                        BroadCastMessage bmsg = (BroadCastMessage)msg;
                        long pid = bmsg.getGroupID().getLocalID();
                        CitizenSet cset = Region.getCitizenSet(pid);
                        if(cset == null){
                                System.err.println
                                        ("CitizenSet is null: placeid="+pid);
                                return null;
                        }else{
                                return cset.getCollection();
                        }
                }else if(msg instanceof IndividualMessage){
                        List<CitizenID> cset = new ArrayList<CitizenID>();
                        cset.add(msg.getSenderID());
                        return cset;
                }
```

- You have only to implement the “resolve” method.
- The return type is a list of destination agent identifiers (CitizenID).

SampleResolver

```
        else if(msg instanceof DirectionMessage){
                        List<CitizenID> citizenSet
                                = new ArrayList<CitizenID>();
        citizenSet.add(((DirectionMessage)msg).getReceiverID());
                        return citizenSet;
                }else if(msg instanceof SetAttributeMessage){
                        List<CitizenID> cset = new ArrayList<CitizenID>();
                        cset.add(msg.getSenderID());
                        return cset;
                }else if(msg instanceof MutualMessage){
                        List<CitizenID> citizenSet
                                = new ArrayList<CitizenID>();

        citizenSet.add(((MutualMessage)msg).getReceiverID());
                        return citizenSet;
                }else{
                        System.err.println
                                ("Message is null or unsupported.");
                        return null;
                }
        }
}
```

- Destination of messages are determined by their types

SampleMessage

```
package mysample.message;

import com.ibm.xasdi_bridge.Message;

public class SampleMessage extends Message {

	private static final long serialVersionUID = 1L;

	public static final int INDIVIDUAL = 1;
	public static final int BROADCAST = 2;
	public static final int ATTRIBUTE = 3;
	public static final int DIRECTION = 4;
	public static final int MUTUAL = 5;

	public SampleMessage(int type) {
		super(type);
	}

}
```

- A message class – extending the Message class - used for this application.
- 5 constant values are represented as message types – each of which is used at a constructor method.

IndividualMessage

```
package mysample.message;

public class IndividualMessage extends
SampleMessage {

	private static final long serialVersionUID = 1L;

	public int selected = 0;

	public IndividualMessage(int selected) {
		super(INDIVIDUAL);
		this.selected = selected;
	}

}
```

- Each agent sends this message only to itself.
- If selected is 0, it is an original message
- If selected is 1, it is a reply message.

BroadCastMessage

```
package mysample.message;

import com.ibm.xasdi_bridge.PlaceID;

public class BroadCastMessage extends SampleMessage {

        private static final long serialVersionUID = 1L;

        public static final int VALUE = 3141;
        private PlaceID id;

        public BroadCastMessage(PlaceID id) {
                super(BROADCAST);
                this.id = id;
        }

        public PlaceID getGroupID(){
                return id;
        }

}
```

- An agent sends this message to all agents.
  - It has PlaceID, but no agents belong to any Place in this application. Therefore, it will be sent to all agents in the same Place.

SetAttributeMessage

```
package mysample.message;

public class SetAttributeMessage extends
SampleMessage {

	private static final long
serialVersionUID = 1L;

	public int attribute;

	public SetAttributeMessage(int attribute) {
		super(ATTRIBUTE);
		this.attribute = attribute;
	}

}
```

- Set the agent attribute
- When an agent receives this message, it change its attribute to that variable.

DirectionMessage

```
package mysample.message;

import com.ibm.xasdi_bridge.CitizenID;

public class DirectionMessage extends
SampleMessage {

	private static final long
serialVersionUID = 1L;

	public DirectionMessage(CitizenID dest) {
		super(DIRECTION);
		this.setReceiverID(dest);
	}

}
```

- A Message class could have a list of CitizenIDs. to specify origin and destination ids.
- The destination id is set in this sample application.

MutualMessage

```
package mysample.message;

import com.ibm.xasdi_bridge.CitizenID;

public class MutualMessage extends
SampleMessage {

	private static final long serialVersionUID = 1L;

	public static final int VALUE = 1732;

	public MutualMessage(CitizenID orig,
CitizenID dest) {
		super(MUTUAL);
		this.setSenderID(orig);
		this.setReceiverID(dest);
	}

}
```

- Two CitizenIDs represent an origin and destination identifiers in this sample.
  - A destination object can create a reply message.

SampleLogConverter

```
package mysample.report;

import java.io.*;
import com.ibm.xasdi_bridge.log.*;

public class SampleLogConverter {
        public static void main(String[] args) {
                System.out.println
                        ("now converting");
```

- This class is used for converting binary log to text data format.

SampleLogConverter

```
try {
        PrintWriter pw = new PrintWriter(new FileWriter("sample.csv"));
        File dir = new File(args[0]);
        File[] logs = dir.listFiles();
        for(int i=0;i<logs.length;i++) {
                try {
                        if(logs[i].isDirectory())continue;
                        LogSet set = LogSet.read(logs[i].getAbsolutePath());
                        while(set.hasNext()) {
                                Log log = set.next();
                                LogDefinition def = log.getLogDefinition();
                                switch(def.getID()) {
                                case 0: convert(pw,log);break;
                                default:
                        System.out.println("unknown");
                        break;
                                }
                        }
                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
        pw.close();
} catch(Exception ee) {
        ee.printStackTrace();
}
}
```

- Convert binary Log files into a CSV text file.

SampleLogConverter

```
    static void convert(PrintWriter w, Log log) throws Exception {
        int value = log.getInt(0);
        String message = log.getString(1);

        w.println(message + "," + value);
    }
}
```

- The “convert” method converts binary data to text (ASCII) data.

SampleLauncher

```
package mysample.sim;

import java.io.File;
import java.util.Properties;

import mysample.SampleCitizenFactory;
import mysample.SampleCitizenProxyFactory;
import mysample.SampleDriver;
import mysample.SamplePlace;
import mysample.SampleRegion;
import mysample.SampleResolver;

import com.ibm.xasdi_bridge.log.ColumnType;
import com.ibm.xasdi_bridge.log.DefaultLogger;
import com.ibm.xasdi_bridge.log.LogDefinition;
import com.ibm.xasdi_bridge.log.Logger;
import com.ibm.xasdi_bridge.message.MessageRepository;
import com.ibm.xasdi_bridge.simulator.Launcher;
import com.ibm.xasdi_bridge.simulator.Region;
import com.ibm.xasdi_bridge.simulator.World;

public class SampleLauncher implements Launcher {

        static final String LOGFILE = "logFile";
        private SampleRegion sampleRegion = new SampleRegion(0);
        private SampleResolver sampleResolver =
                new SampleResolver();
```

- Simulation starts from this SampleLauncher class that implements the *Launcher* interface.
- It creates most of objects such as CitizenProxy, Driver, Place, and so forth.

SampleLauncher

```
@Override
public void prepare(Properties prop) {
        //      defines Log
        String filename = prop.getProperty(LOGFILE, "sample");
        Logger logger = World.world().setLogger(new DefaultLogger());
        logger.setFile(new File(filename));
        LogDefinition def = new LogDefinition(0,0);
        def.addColumn(ColumnType.INT);
        def.addColumn(ColumnType.STRING);
        logger.addLogDefinition(def);
        logger.enableLog(0);

        //      defines SampleCitizen(Proxy)Factory, SampleRegion
        SampleCitizenProxyFactory sampleCitizenProxyFactory =
                new SampleCitizenProxyFactory();
        sampleRegion.addCitizenProxyFactory
                (SampleRegion.FACTORYID, sampleCitizenProxyFactory);
        sampleRegion.setNumberOfPhases(1);
        sampleRegion.setMessageRepository(new MessageRepository());
        sampleRegion.getMessageRepository().setResolver(sampleResolver);

        SampleCitizenFactory sampleCitizenFactory =
                new SampleCitizenFactory();
        sampleRegion.addCitizenFactory
                (SampleRegion.FACTORYID, sampleCitizenFactory);
        sampleRegion.createCitizenSet(0);
```

- The “prepare” method creates a set of principal objects.
  - Region, Log, Citizen, CitizenProxy, Factory...

SampleLauncher

```
        //      defines a place, a driver, a simulator
        SamplePlace samplePlace =
                new SamplePlace(sampleRegion, 0);
        sampleRegion.addPlace(samplePlace);
        SampleDriver sampleDriver =
                new SampleDriver(samplePlace, sampleRegion);
        sampleRegion.addDriver(sampleDriver);
        World.world().setDeltaTime(1);
        World.world().setSimulationTime(0, 4);
        World.world().addRegion(0, sampleRegion);
}
```

- Create objects about Place and Driver.
- Set simulation time and Region object to the World class.

SampleLauncher

```
    @Override
    public void start(Properties prop) {
            System.out.println("start...");
            World.world().start();
    }

    @Override
    public Region getRegion() {
            return sampleRegion;
    }

    @Override
    public MessageRepository getMessageRepository() {
            return sampleRegion.getMessageRepository();
    }

}
```

- An application will start from the “start” method.
- Overided methods such as “getRegion” and “getMessageRepository” so that the underlying runtime of X10-based Agents Executive Infrastructure for Simulation can refer variables in Region and MessageRepository.

DEBUGGING SAMPLE APPLICATION

Copy property XML file

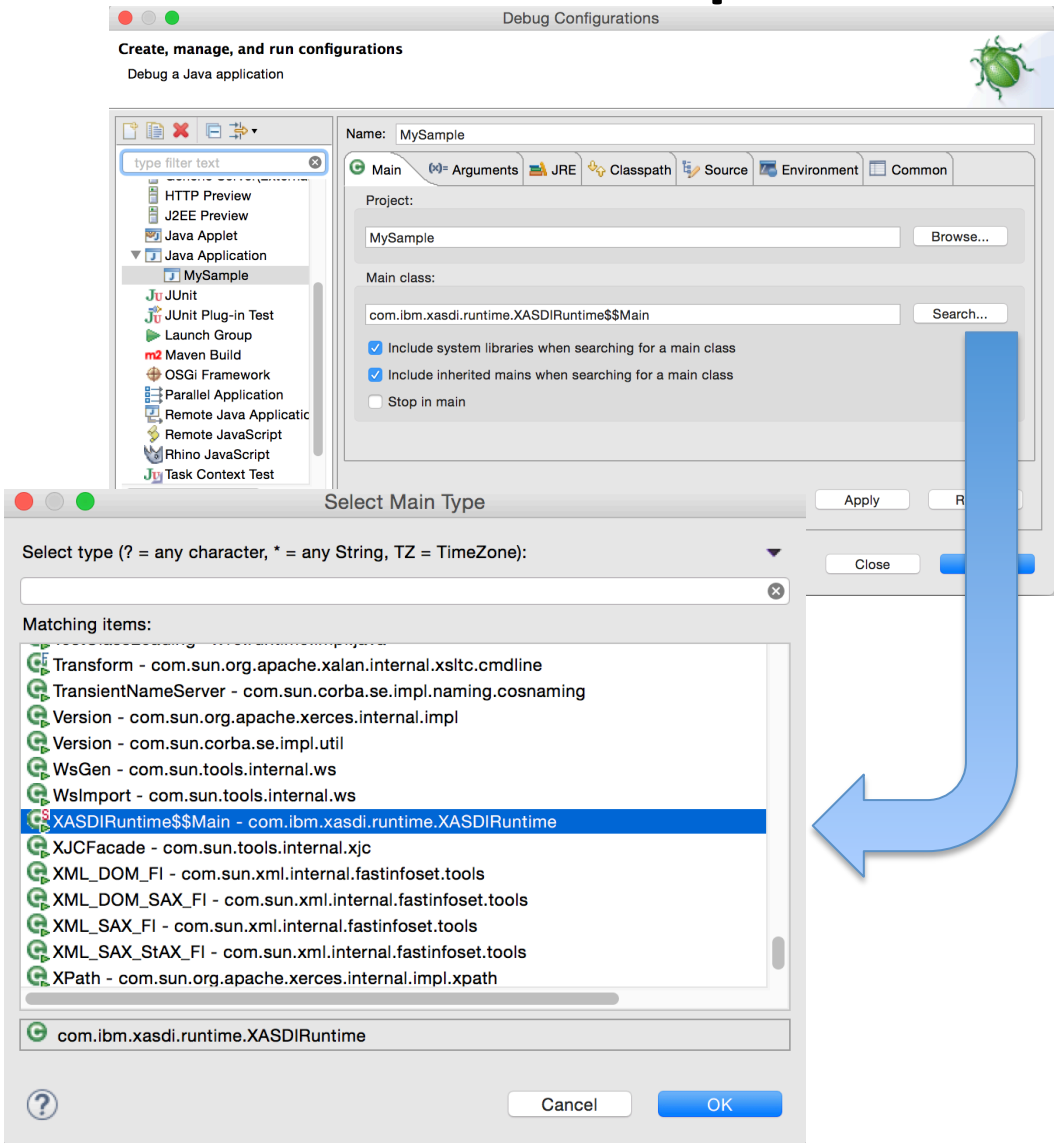
- First, create directories “Sample” and “logs”.
- Simulation program on X10-based Agent Simulation on Distributed Infrastructure (XASDI) needs property XML file and CSV files.
  - The easiest way is to copy these files from extracted archive directory MySample/Sample/ to MySample/Sample/.

Setting up Projects in Eclipse

- Use a debugger in Eclipse (4.2.x or later).
- Confirm two jar files named “xasdi.jar” and “x10.jar” are in the class path of the MySample project.

Debug configurations in Eclipse

- Run → Debug Configurations…
- Select the Main class
  - Project: MySample
  - Main class : **com.ibm.xasdi.runtime.XASDIRuntime$$Main**

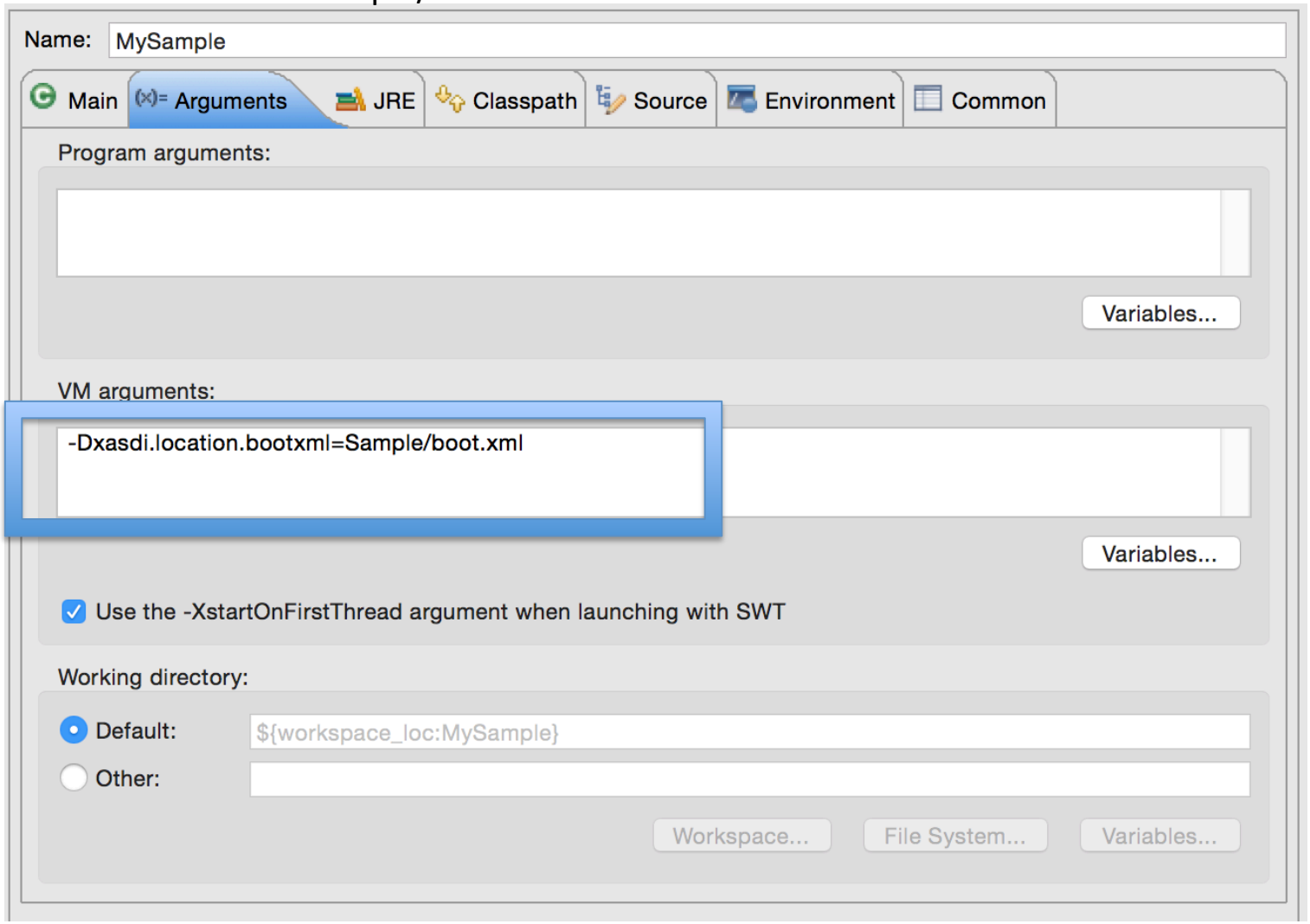


VM arguments

- Set the Java VM arguments

  -Dxasdi.location.bootxml=Sample/boot.xml

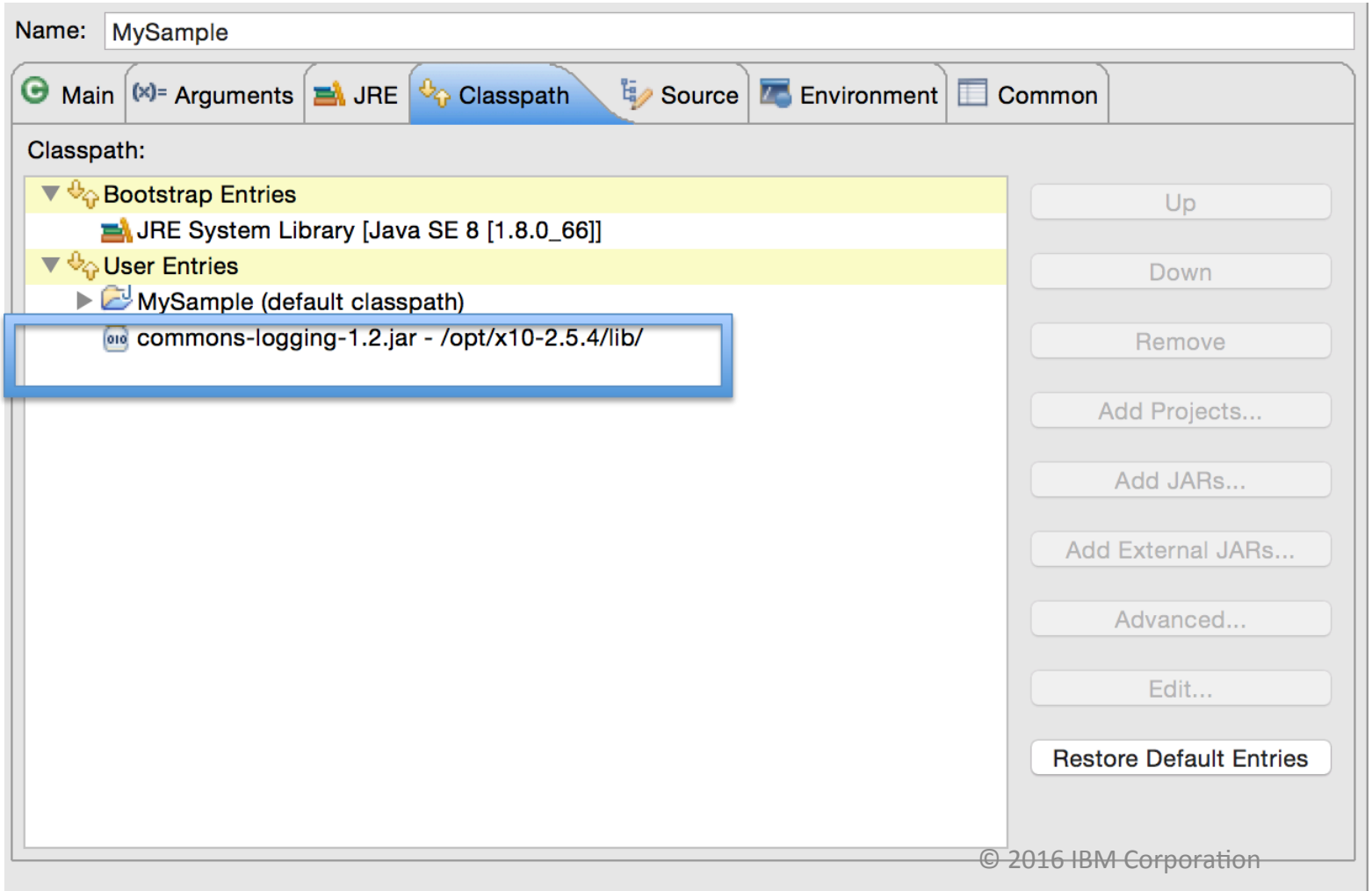
Java VM Arguments



VM arguments

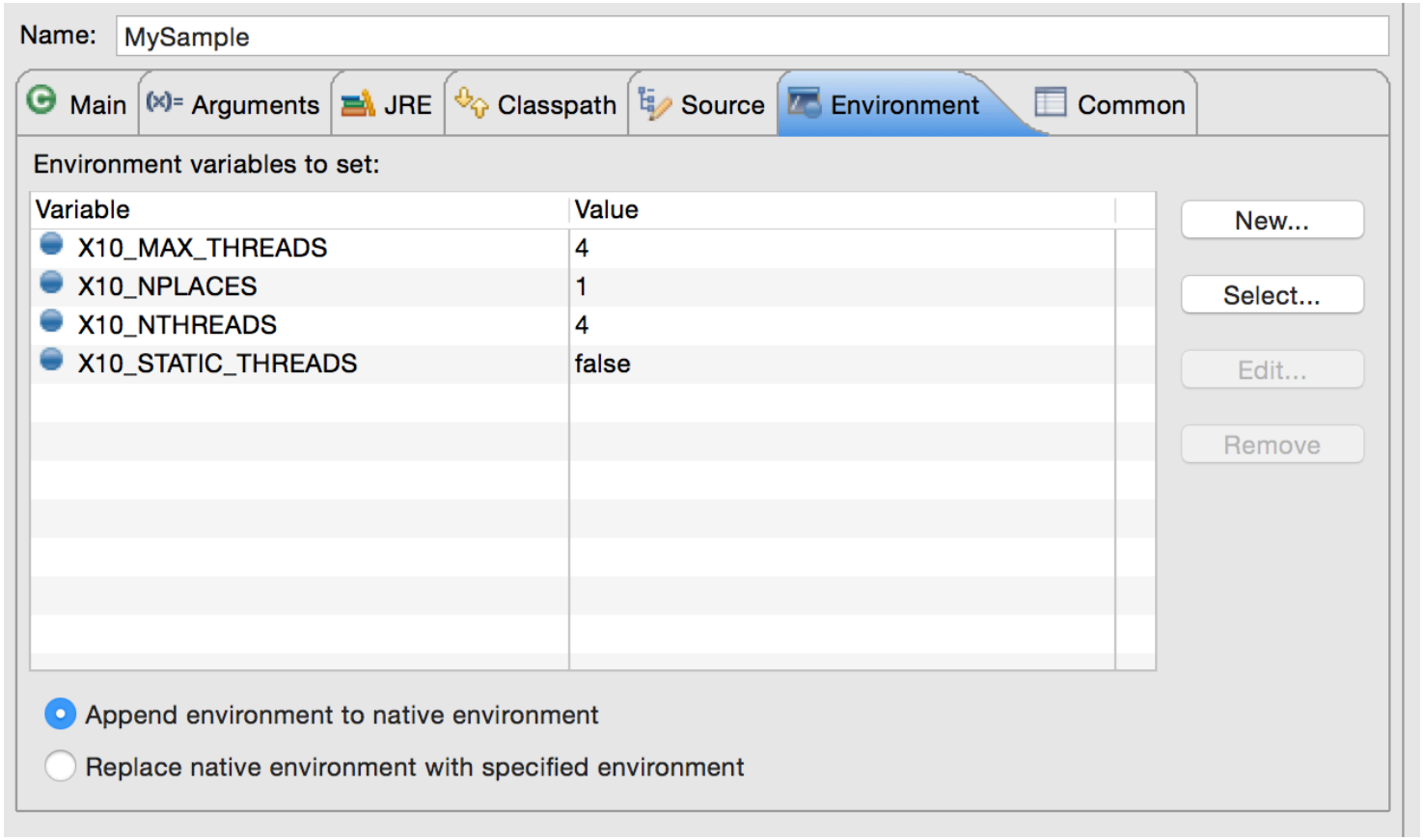
- Add a required jar from x10 lib directory
  - (X10-2.5.4 install directory)/lib/commons-logging-1.2..jar

External JAR

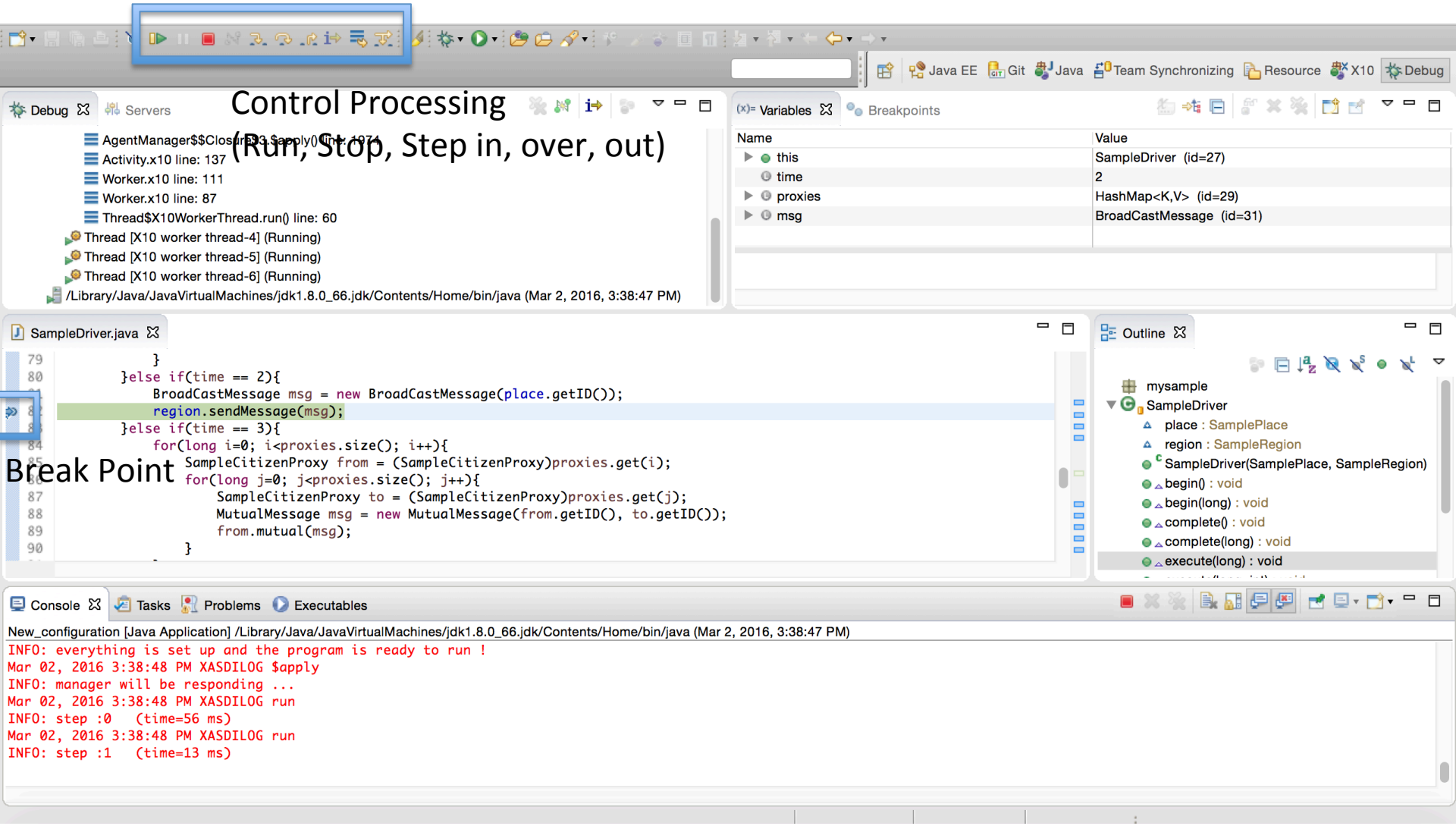


Environment variables

- Set the environment variables (* $TH: the number of CPU cores)
  - X10_MAX_THREADS: $TH
  - X10_NPLACES: no need to specify (default is 1)
  - X10_NTHREADS: no need to specify (default is the number of CPU cores)
  - X10_STATIC_THREADS: no need to specify (default is false)



Debugging with Eclipse



Start the debugging by setting breakpoints in the Java source codes.

Debugging with Eclipse

- At the end of simulation, log files are generated at MySample/logs

RUNNING SAMPLE APPLICATION

Copy shell script

- First, copy a shell script “run-Sample.sh” from extracted archive directory MySample/ to MySample/ directory.

Setting simulation properties with boot.xml

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="directory">./</entry>
<entry key="simTime">4</entry> <!-- setting the simulation steps -->
<entry key="simName">MySample</entry>
<entry key="nThreads">2</entry>
<entry key="nPlaces">1</entry>
<entry key="nPhase">1</entry>
<entry key="launcherName">mysample.sim.SampleLauncher</entry>
<entry key="workFile">Sample/citizenPlaceMap.csv</entry> <!-- mapping file -->
<entry key="driverFile">Sample/driverPlaceMap.csv</entry> <!-- mapping file -->

<entry key="logFile">logs/sample</entry>
</properties>
```

Mapping file for multi-nodes

- Two files, citizenPlaceMap.csv and and driverPlaceMap.csv, are needed.
- The first column indicates the agent identifier and the second column indicates the identifier of the X10 place.
- If you would like to run your simulation on multiple nodes, you can specify the identifier greater than 0 for some set of agents.

```
0,0
1,0
2,0
3,0
4,0
5,0
6,0
7,0
8,0
9,0
```

Running simulation script

- Then, execute this script.
  - **./run-Sample.sh**
  - * Tips: If you would like to run your simulation on multi-nodes, please specify the environmental variable, X10_NPLACES to the number greater than 1.
- After running this program, a set of log files can be found in logs/ directory.