Computational Modeling in Practice: Building Models in Repast[1]

Kyle A. Joyce

Department of Political Science
University of California, Davis

August 6-8, 2014

[1]Some of this material is drawn from David Siegel.

Overview

- Object-Oriented Programming
- Java
- Repast

Object-Oriented Programming

- Break up problems into separate aspects.
  - Agents become an object, the space agents occupy can become an object, and so on.
  - The main program sets up the interface, initializes the objects, and tells them to do their thing.
- Objects have associated data and processes (methods) that dictate what they are and how they behave.

Objects as Structure

- The use of objects structures the model.
- Agents need to be specified; they must have properties (variables and parameters) and methods (strategies and utilities).
- An order of operations must be specified.
- A data-collecting scheme must be implemented.

Modularity

- Objects are also modular.
- Code for different actors, etc. can be separated.
- Modularity enables easy expansion—just drop in a new object.
- Modularity also helps debugging.

Which Language?

- Java, C++, and Python are popular choices.
- All free and platform independent.
- Lots of packages mimic this functionality (e.g., NetLogo).
- Programming in a lower level language allows for more control over what's going on, and more flexibility.
- Can always add libraries to add functionality.

To Get Started

- You probably want an IDE such as eclipse (http://www.eclipse.org/)
- Java runtime environment (JRE)
- Any libraries that you might want to use: Repast, JUNG, MASON, etc.

Objects

- We will be defining objects in different files (e.g., agent1.java, mainmodel.java, space1.java, etc.).
- When the model is complied each file will become a class.
- Each object will have the following:

  ```
  package myproject;
  import necessary external packages;
  public class myclass() {
        body of code
  }
  ```
- Each executable object—in this case, mainmodel—must have a function **public static void** main(String[] args) {} defined within it. This function is what executes when the program runs.

Common Syntax

- Each command ends with a ; but code that tells you where to go does not (e.g., for).
- Sections of code are delineated by {}.
- A . indicates location. For example:
  - Math.exp() means that the exp() function is defined within the Math package.
  - agent.imbored references the imbored variable within the agent object.
  - System.out.println(“ ”) illustrates nested locations: println is located in the out package, which is located within the System package. This function prints whatever is in “ ” to the screen followed by a new line.

Importing

- Any class within the same package does not need to be imported to use.
- Other packages need to be given their own import package line in the code.
- For example:
    - java.util.* contains definitions of Lists, ArrayLists, Vectors, and related utilities.
    - java.awt.Color contains color definitions and functions for making other colors.
    - java.io.* contains input and output functions.

Defining Variables

- Variables must be defined before they are used.
- Variables are defined by statements like:
  - **double** x; (within a class)
  - **public** int y; (for a variable to be seen by other objects)
  - **private boolean** flag; (for a variable used only by the object in which it is located)
- Variables can be initialized to a value when they are defined, but do not have to be. For example:
  - **public int** y; y=7; or
  - **public** String z="Still Bored?";

Common Data Types

- **double**: 64 bits, default real number format.
- **float**: 32 bits, like double, but smaller.
- **int**: holds integers up to about 2 billion.
- **long**: holds integers up to about 9 quintillion.
- **boolean**: either **true** or **false**.
- **char**: single character, encapsulated by ‘ ’.
- String: multiple characters, encapsulated by “ ”.
- Object: Any object.

Algebraic Expressions

- Once variables are defined, they can be changed via expressions like:
  - x=z*y+(r-2)/7. Common rules of order apply. Be careful mixing **int** and **double**!
  - Math.exp(x), Math.sqrt(z), Math.pow(y,2) and other functions exist within standard packages.
  - z++ increments z.
  - x=x/3. Old value of x becomes new one.
  - z=y%5. % is the mod operator, and calculates the remainder after division.

Defining Classes and Objects

- **public class** agent1() begins code after import statements.
- **public** agent1() occurs inside of main brackets.
- To create a new instance of a particular object within another class, use **new**. For example, if you want to create a new agent1 variable within mainmodel, type: agent1 agent = **new** agent1();

Using Objects

- Once you have created a new instance of an object, in this case agent of type agent1, you can treat this object like any other variable. So, in effect, your agent1 class becomes a type of variable just like **double**, and you defined a new variable agent with the **new** command.
- If agent1 contains variables and methods, you can use them just like Math.exp(). For example: y=agent1.bored/agent1.thisisfun; or agent1.move().

Functions

- Defined similarly to objects, functions allow you to perform complex tasks through a simple function call.
- The general syntax is: **public** `data_type` calcpi() {} where `data_type` is the type of value that the function returns (**double**, etc.).
- If the function returns a value, a **return** `value;` statement must reside within the {}. If not, use **void** for the `data_type` and do not include a **return** statement.
- For example:

```
public double getbroadcastProbability() {
    return broadcastProbability;
}
```

Arguments

- Both objects and functions can take arguments within their ().
- For example, to define a function that multiples two doubles:
  ```
  public double mult(double x, double y) {
      return x*y;
  }
  ```
- Passing arguments is not necessary for public variables within the same class, but is for private variables, or variables defined only within a given section of code.

Lists and Arrays

- Often we need to consider multiple variables at once. Arrays, generally used for sequences of doubles or integers, and Lists, generally used for objects, satisfy this need.
- Arrays: **int**[] x = **new** int[10]; Individual elements in the array are accessed by x[j], and the size of the array must be specified.
- Lists: ArrayList agentList = **new** ArrayList(); This one comes up a lot, as it is a list of agents in the model. You can add and take away agents dynamically.

Conditional Statements I

- Often in coding we want to tell the computer to execute code only if some condition is met. Two ways of doing this are commonly used: **if** and **switch**.
- **if** statements take the form: **if** (x==y){}. The double equal sign is a Boolean operator: it returns **true** if x equals y, and **false** otherwise. One could also use Boolean variables directly: **if** (condition) {}.
- Some other Boolean operators:
  - != is not equal to.
  - >= is greater than or equal to.
- For example:

```
if (dsurf != null)
    dsurf.dispose();
```

Conditional Statements II

- **if** statements can also be attached to **else** statements to make a string of conditions that are checked in order. Be careful specifying which **else** applies to which **if**!
- For example:
  if (flag1) {1;}
  else if (flag2) {2;}
  else {3;}
- This runs 1 if flag1 is true, runs 2 if flag1 is false, but flag2 if true, and runs 3 if they are both false.

Conditional Statements III

- The switch usually examines an integer and does one of a sequence of blocks of code depending on what value the integer takes.

```
switch (x) {
      case 1:
        z=z/2;
        break;
      case 2:
        z=7;
        break;
}
```

- **case** means "if x is 1"; **break** can be used in other ways as well, and jumps out of whatever command—in this case, **switch**—in which the program is presently in.

Loops

- Usually, we will want to tell the computer either to do something a specified number of times (the **for** loop), or to tell the program to do something until some condition is met (the **while** loop).

For Loops

- **for** loops take the form:

  ```
  for (int i=0; i < 10; i++) {
      x = x+1;
  }
  ```
- The portion in () defines a variable *i* and initializes it to 0, sets up a Boolean condition for when to continue (as long as $i < 10$), and instructs the program how to change *i* with each execution of code in the loop (here x=x+1). The output of this code would be to increase x by 10.
- The condition is checked before each loop.

While Loops

- **while** loops take the form:

```
while (x < 10) {
      x = x+1;
}
```

- At the beginning of each loop, the program checks to see if $x < 10$ is true. If it is, the loop executes. While this code does the same thing as the **for** loop, the **while** loop allows for more complex conditions to be used.
- Note that this code would provide a different result if $x \neq 0$ at first.

Extends and Implements

- In addition to letting us define objects, Java lets us define sub- and super-classes of objects. Thus, we might create a car, and then create a hatchback that has all the same properties of a car, plus some.
- To avoid redoing work, we can define a class as follows: **public class** hatchback **extends** car. This allows us to use code from both the new class as well as the old car class when we create a new hatchback variable.
- Sometimes instead of a full class to extend there will only be an interface; there we use the syntax **public class** agent1 **implements** Drawable. This particular implementation comes up often when using Repast in order to show agents graphically.

Learning Java Resources

- New to Java Programming Center
- The Java Tutorials
- Java in One Day

External Packages

- Java has many built in functions but some tasks can be complex (and potentially time consuming) such as building a GUI or generating graphics.
- In such cases, one can import external packages and use their objects and methods as one would use any Java command.
- One useful package for agent-based modeling is Repast (specifically Repast 3).
- See the document on the course website for configuring Eclipse, Java, and Repast.

Repast Overview

- Repast is a toolkit that provides packages that aid in automating many of the tasks involved in programming a model.
- In general, you will program the rules for your agents and their environments, and Repast packages will produce graphics, aid data collection, provide a scheduling mechanism, etc.
- Before Repast can help you, you must link to its methods. To illustrate how this works, we will use the Sugarscape model as an example.
- Note: Sugarscape is also available in the NetLogo Models Library, so you can compare the NetLogo and Java/Repast implementations of the model (see Sugarscape 1 in the NetLogo Models Library).

Beginning a Repast Model

- Repast has several classes and functions that you must define in your code.
- After specifying your own package and importing the relevant Repast packages, you must tell the executable class to use the core commands of a Repast model.
- We do this by including **extends** SimModelImpl in the definition of our main class: **public class** SugarModel **extends** SimModelImpl { (See line 73 in SugarModel.java)
- Within the main body of the code we also include: **public** SugarModel() {} (See lines 190-195 in SugarModel.java). The body of this may be empty.

Variables

- Variables are usually created next. Several must be created for Repast to work, although their names may be different from those used here.
- **private** Schedule schedule; This creates the object that holds the sequence of each time period's actions (See line 76 in SugarModel.java).
- **private** ArrayList agentList = new ArrayList(); This creates the list of agents (See line 81 in SugarModel.java).
- **private** DisplaySurface dsurf; This creates the display (See line 139 in SugarModel.java).
- Other variables corresponding to graph and histogram objects, spaces, other lists, and so on should also be created here.

Build Model

- The next function that must be included takes the form: **private void** buildModel() {} (See line 207 in SugarModel.java)
- Inside the body of this function you must create all the individual agents. Usually agent's initial parameters are set at this time as well. (See lines 216-218 and lines 407-428 in SugarModel.java)
- If there are spaces, grids, or like objects, these should also be created here (See lines 213-214 in SugarModel.java).
- Data recorders should be included here as well (See lines 221-252 in SugarModel.java).

Build Display

- Next, you must build the display. This is done in the function: **private void** buildDisplay() {} (See lines 263-328 in SugarModel.java).
- Inside the body of this function you add all of the objects that you want to be displayed on the screen, including graphs and histograms, in addition to the main display.
- Adding the main display involves creating a new display, and then adding that display to the already-created dsurf. Probable indicates that you can click on the agents; an event listener is added at the end to make something happen.

Build Schedule

- The schedule dictates the order, and it must fall within: **private void** buildSchedule() {} (See line 333 in SugarModel.java).
- Within this function we often create another class that **extends** BasicAction (See lines 351-378 in SugarModel.java).
- This class must contain a function **public void** execute() {}, though it need not contain anything else (See line 352 in SugarModel.java).
- Inside the execute() function we put everything that happens during a “tick”, including updating displays (See line 369) and telling our agents (See lines 359-362), and spaces (See line 363) what to do.
- Finally, we end by scheduling the class (See lines 381 and 397 in SugarModel.java).

Connecting to the Display

- In a typical model, we will have several parameters we want to explore, and we would like those parameters to be changeable via the GUI.
- To do this we need to include several functions:
  - For every parameter, we need two functions: one that gets its value and one that sets its value. If the parameter is a double named x, then these must be called **public double** getx() and **public void** setx(**double** x). Be careful: make sure the names match! (See lines 496-566 in SugarModel.java)
  - We also need a function called public String[] getInitParam() {} . Inside this function are only two lines: String[] params = {“x”,“y”,...}; where x and y are your parameter names, and **return** params; which returns the string you just defined. (See lines 568 to 573 in SugarModel.java)

Begin

- Having built our model, we need to tell the simulation to begin. We do this with: **public void** begin() {} (See lines 582-592 in SugarModel.java)
- Inside this we call all three building functions (buildModel(), buildDisplay(), buildSchedule()), and tell all of our graphics to display (dsurf.Display(), graph.Display(), and bar.Display()).

Setup

- This final function does all of the cleanup between runs of the simulation (See lines 600-663 in SugarModel.java).
- There are three basic steps accomplished here:
  1. Eliminate all old surfaces (See lines 608-618), schedules (See lines 601), and the like.
  2. Create new lists (See lines 602-603), graphs (See lines 633 and 638), a schedule (See line 623), and a display surface (See line 624).
  3. Assign initial values to the parameters (See lines 641-647).

Assorted Functions

- Repast also uses a couple of simple functions:
  - **public** Schedule getSchedule() {
        return schedule;
    } (See lines 666-668 in SugarModel.java)
  - **public** String getName() {
        return "mainmodel";
    } or whatever the name of your executable class is (See lines 671-673 in SugarModel.java).
- The coding of your agent or space classes usually contain the variables specific to a given agent or space, as well as the rules these objects use to change over time (See the files SugarAgent.java and SugarSpace.java).
- If you want to display agents, the agent classes must implement Drawable:
  public class SugarAgent **implements** Drawable {

Executable Function

- The main class must be: **public static void** main(String[] args) {} to execute.
- Within the executable function in your main class (See lines 675-681 in SugarModel.java), there must be three lines:
  1. SimInit init=new SimInit(); (See line 676)
  2. SugarModel model = new SugarModel(); (See line 677)
  3. init.loadModel(model,“”,false); (See line 678)
- These initialize the model and set it to run in non-batch mode.
- Once this is all done, you can add your own code.

Data Collection

- Repast makes it very easy to collect data using the **DataRecorder** object and write the data to a file.
- All that we need to do is create the **DataRecorder**, add data sources to it, and schedule when we want to record the data and write the data to a file.
- To illustrate how the **DataRecorder** works we will continue with the Sugarscape model, which already has a data recorder programmed.

Data Collection

- First, we need to import the data recorder library (See line 40 in SugarModel.java).
- Sometimes we only want data to be collected at certain points during the modeling process. As a result, it is helpful to create a boolean variable that determines whether or not the model will record the data.
- In SugarModel.java we need to change the write variable to true (See line 118) to record data from the model. Note: this already done in SugarModel.java
- The reason we do not always want to record the data when we are still developing the model is because is slows down the model.

Data Collection

- Creating a data recorder simply requires us to add the following code:
  recorder = new **DataRecorder** ("data/sugar_data.csv", this);
  (See line 222 in SugarModel.java)
  Note: the data for SugarModel.java will be saved in a folder called data in the workspace folder.
- The first argument is where we want the file saved and the name we want to save the file as.
- The second argument is the model associated with the recorder. The argument "this" means that the recorder is created within the model.
- Note: we create the data recorder and tell it what data we want to collect in **buildModel()**.

Data Collection

- Once we have created the **DataRecorder** we need to decide what data to collect.
- The actual sources of data can be of two types: numbers (ints, floats, longs, doubles) and objects (Strings, etc.).
- After deciding what data to collect we either call an object or the value of some variable, which are wrapped in one of two interfaces.
- The following code collects numeric data: recorder.**addNumericDataSource**("name of variable", new NumDataSource());
- The following code collects data from an object: recorder.**addObjectDataSource**("name of object", new ObjDataSource());
- See lines 226-251 in SugarModel.java.

Data Collection

- Now that we have created a data recorder and told the data recorder what data to collect we need to schedule the recording of data in **buildSchedule()**.
- We schedule the recording of data in a class that extends **BasicAction**.
- The following code tells the schedule to record the data: recorder.record(); where recorder is the variable name we have assigned to **DataRecorder**.
- See lines 373-374 in SugarModel.java.

Data Collection

- Finally, we have to schedule when the data gets written to a file (data is stored in memory until then).
- This is done using the following code: schedule.**scheduleActionAtEnd**(recorder, “writeToFile”);
- See lines 393-394 in SugarModel.java.

Data Collection

- The Sugarscape model run was scheduled to run forever (i.e., it has no stopping rule).
- I added a stopping rule using the following code: schedule.**scheduleActionAt**(100, this, “stop”);
- See line 397 in SugarModel.java.
- The first argument tells the model to run for 100 time steps (“ticks”).
- The second argument refers to the model.
- The third argument tells the schedule what action to take when the 100th time step is reached.

Data Collection

- At this point, if we run the model a .csv file will be saved in the data folder.
- The top of the file contains the start of the simulation and the parameter settings.
- Next, the data appear with one row for each tick and the columns contain the variables we asked the data recorder to record.
- Note: each time you run the model a new data file will be saved (at some point you will want to delete the old files).
- Note: the structure of the .csv file is slightly different for batch runs (more on this below).

Parameter Sweeps and Batch Runs

- Single runs of the model (or illustrative runs) can be useful for diagnosing problems and understanding initial model dynamics but they cannot be used to describe results from the model.
- In order to be able to describe results from the model we need to do simulations (or batch runs).
- In addition, we want our results to be robust to changes in the parameter settings.
- In order to demonstrate robustness we do parameter sweeps.

Parameter Sweeps and Batch Runs

- Each time you run a model in non-batch form a GUI and displays appear.
- The presence of a GUI and displays can dramatically slow down simulations of the model.
- As a result, it is a good idea to turn them off.
- We can accomplish this by creating a boolean variable that when set to true turns the GUI and displays on and when set to false turns them off (See line 120 in SugarModel.java).

Parameter Sweeps and Batch Runs

- Once we have created this variable we need to add it to some of Repast's built in methods: buildSchedule() (See lines 368-372 in SugarModel.java), begin() (See lines 587-591), and setup() (See lines 651-662).

Parameter Sweeps and Batch Runs

- We use the initial value of parameters to create the actual model.
- The values of these parameters can be seen in the GUI when we run the model in non-batch form by adding them to the function: public String[] getInitParam() (See lines 568-573 in SugarModel.java).
- Each of these parameters must have get and/or set methods associated with them. If get and set are both present the parameter is readable and writable. If only the get method is present the parameter is read-only.
- See lines 496-566 in SugarModel.java.
- Note: when we created our loadGUI parameter, it was added to public String[] getInitParam() and we created get and set methods for the parameter.

Parameter Sweeps and Batch Runs

- Repast makes it very easy to conduct batch runs using a parameter file.
- The parameter file defines a parameter space and describes how the model should search that space.
- A parameter file has the following format:

```
runs: x
Parameter {
   value_definition
}
```

- Here, x represents the number of simulations you want to conduct and Parameter is the name of some model parameter accessible through get and set methods.

Parameter Sweeps and Batch Runs

- The `value_definition` is composed of one or more keywords and corresponding values that must always appear together:
  - start: the starting numerical value of a parameter
  - end: the ending numerical value of a parameter
  - incr: the amount to increment the current value of the parameter
- Other useful keywords for `value_definition` are:
  - set: defines a single numerical value as a constant for the entire batch run
  - `set_boolean`:defines a boolean value as a constant for the entire batch run
- More than one parameter can be specified and parameters can also be nested.

Parameter Sweeps and Batch Runs

- I created a parameter file for the Sugarscape model called sugarscapeparams.txt, which is located in the main sugarscape folder in the workspace.
- In order run the model in batch form we need to modify: public static void main(String[] args) { .
- Since you are going to want to run the model in both batch and non-batch form, you will simply add a new line of code to tell it to run with model with the settings specified in the parameter file.
- So comment out the line: init.loadModel(model, “”, false);
- And add the line:
  init.loadModel(model, “nameofparameterfile.txt”, true);
  where nameofparameterfile for the Sugarscape model is sugarscapeparams (See line 680 in SugarModel.java).

Parameter Sweeps and Batch Runs

- Now, when we hit the Run button the model will run in batch mode without the GUI and displays because we set the parameter `loadGUI` to false in sugarscapeparams.txt.
- You can track the progress of the simulations in the console.
- Once the simulations are finished the data will be written to the specified file(s).

Parameter Sweeps and Batch Runs

- The structure of the file generated by a batch run is similar to the one generated by a non-batch run but there are some differences.
- Any parameters that were varied during the course of the simulation are not listed at the top of the file but are included as columns in the file with the settings listed in the parameter file.
- Additionally, there is a line at the bottom of the file that lists the ending time of the simulation.

Parameter Sweeps and Batch Runs

- Now that we have our results, we can analyze them.
- I normally organize the data in Stata and then do the analysis is R but you should choose whatever software works best for you.
- You can find a Stata .do file and an R script in the data folder that will organize (Stata) and then plot (R) the data.
- Note: before you read the data into whatever software you are using, you need to delete the information at the top and bottom of the data file, taking note of the parameter values.

Repast Tutorials

- A nice collection of resources to help you get started with Repast can be found here.