Threadneedle Programming Guide

Jacky Mallett

Chapter 1: Introduction

This guide explains how to add agents to Threadneedle or modify the existing ones, as well as how to interact with other agents using the financial system provided by the simulation. Instructions are also provided on specific programming topics within Threadneedle: statistic creation and handling, adding to the main GUI using the Java FXML interface, specifying new Charts, overview of Threadneedle's architecture.

Users who just want to modify or create agents - which will typically be done by subclassing existing agents - can find instructions in Chapter 3. How to add statistics to the chart display can be found in section **??**, and general instructions on modifying the FXML user interface in section 3.1.

1.1 Design Philosophy

There are some overall design considerations embedded in the framework that it helps to be aware of, which have implications for programming and simulation design.

No Equations

Threadneedle is intended to provide as accurate a reproduction as possible of the financial control and information mechanisms used in modern economies. The status of all economic equations, and in particular any that incorporate financial information is treated as not proven.[1] Threadneedle is intended amongst other things as a tool for testing these equations under controlled conditions which can then be used to ascertain their accuracy or otherwise.

All agent actions must be performed by the agent

No economic actions are performed by default. Agents must call the payTaxes() and payDebts() methods each step if they wish to pay those bills.

This can be problematic when running simulations if programming errors occur, and the simulation is not collecting taxes or paying debts when it is expected that this is occurring. The alternative though is that agents have no choice about what they do, and when they do it. Both taxation and debt payments are sensitive to the order of evaluation of actions by the agent in each round, for example - for example, salaries being paid before debts are settled may cause the agent to default on a debt payment, debts before salaries may cause salaries not to be paid, and so this is left to the agent's programming. Be aware of it though as a potential source of economic bugs.

[1] In the Scottish Jurisprudence system there are three possible verdicts, guilty, not guilty and not proven. The not proven verdict is used in cases where sufficient evidence supporting either guilt or innocence has not been provided to the satisfaction of the jury.

Chapter 2: Directory Structure

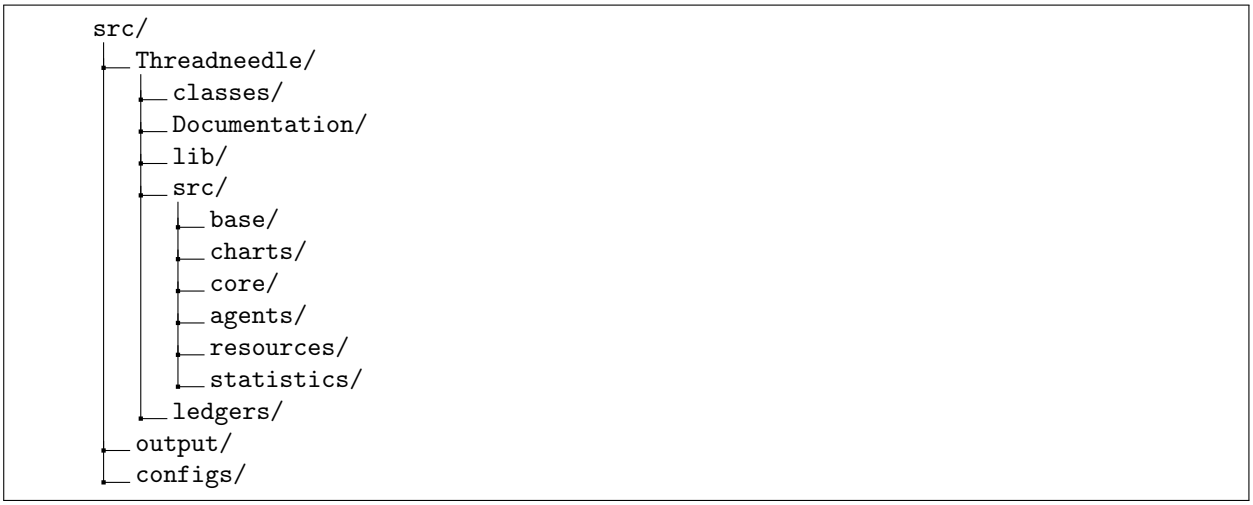

Figure 2.1: Threadneedle directory structure

.

Figure 2.1 shows the directories installed under Threadneedle containing the source code, documentation and example simulation files. Pre-defined agent classes can be found in the *src/agent* directory, and any new agents should also be placed in that directory. The *src/resources* directory contains the ascii fxml files used to generate the user interface, these can be modified and specific instructions on how to do this is provided in the FXML section below3.1.

Chapter 3: Agents

Every economically active entity in Threadneedle is created as a distinct, discrete agent. Although all agents inherit from the same base class (Agent), a distinction is made, primarily for future development purposes, between agents that are representing companies, such as Banks, Markets, etc., and those that represent individuals, such as Workers, Borrowers, Investors.

The inheritance tree for agents is shown in Figure 3.1. New agents which do not specialise existing agent types should inherit directly from either of the abstract classes, Company or Person.

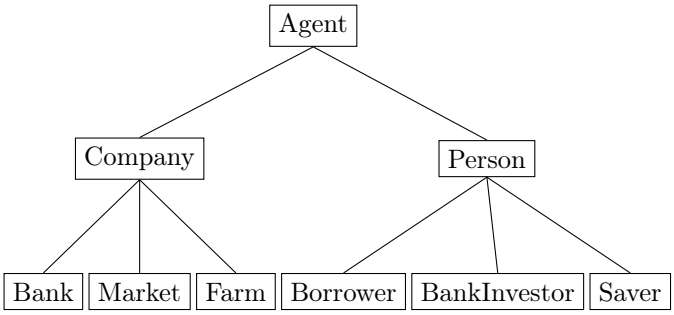

Figure 3.1: Agent inheritance in Threadneedle

Common functionality used by both companies and people is provided by the base Agent class. The Company agent additionally provides mechanisms for shares to be issued, and dividends paid on the shares, although this is not required; companies can also exist as purely private entities. The People class on the other hand provides support for labour employment, unemployment, and consumption of Needs and Wants as specified by individual simulations. Taxation can currently be applied as a flat rate on either class of agent, with a limit that can be set to provide a bound below which no tax is payable.

3.1 Creating new agents

Programming

Creating new agents is straightforward, but the steps provided below must be followed carefully in order to integrate them correctly within the Threadneedle framework and to allow simulation step evaluation to work correctly.

In order to interact correctly with the rest of the simulation, agents must be part of the *agents* packages in *src/agents* and declared *public*.

New agents must inherit from either the Company or Person abstract class, and implement constructors following a pre-defined template. Other functionality can be provided at will using the *evaluate()* method to implement actions for the Agent to take at each step.

1. Create new java class inheriting from Company or Person
2. Following the template in Figure 3.1 add methods:
    - Agent constructor for gui

- Agent constructor for json (saved configuration files)
- evaluate()

3. Create an icon for the agent and add it to the *image* directory
4. Add agent to the drag and drop panel by editing the threadneedle.fxml file following the template in Figure 3.2

As long as this is done correctly the simulation framework will automagically evaluate agent behaviours each round, keep charts updated, roll over any statistics used by the agent, and provide access to the new agent for other agents and through the batch/command line interface.

An example of the minimum code needed to instantiate a new agent is shown in Figure 3.1.

Listing 3.1: Example Agent

```
package core;                          //All agents are part of the core package

import statistics.*;                   // Required if agent is using statistics
import static statistics.Statistic.Type
import static base.Base.*;             // Required for access to base functions

import com.google.gson.annotation.Expose; // Required to save simulation


public class ExampleAgent extends Person
{
  @Expose public long mySalary = 10;   // Exposed variables will be saved

  /**
   * Evaluation function for ExampleAgent will be called once every
   * simulation step.
   *
   * All agent behaviours should be called from here.
   */

   protected void evaluate(boolean report, int step)
   {
      super.evaluate(report, step);     // Required to invoke framework

      // Assuming that funds are available these methods will make
      // any outstanding payments for this step.

      payDebts();                       // Make payments on all outstanding loans
      payTaxes();                       // Pay any taxes

   }

  /**
   * Constructor from GUI and CLI.
   *
   * @param name     Unique and identifying name for agent
   * @param deposit  Initial bank deposit
   * @param govt     Government
   * @param bank     Bank where agent's account will be created
   */

   public ExampleAgent(String name, Govt govt, Bank bank, long deposit)
   {
      super(name, govt, bank, deposit);

      // Agent specific instantiation code can be placed here, but
      // must be duplicated in the JSON file constructor below
   }

  /**
   * No parameter constructor for loading from JSON simulation save files.
   * All @Expose'd variables will be initialised by GSON, and it is
   * the responsibility of the developer to set anything else correctly.
   */
  public Borrower()
  {
    super();
  }

  // Other methods may be provided or overridden as required for the agent.

}
```

Adding Agents to the User Interface

In order to add an agent to the drag and drop user interface, once the source code has been created and added to the *src/core* directory, it is necessary to modify the fxml files which describe the user interface. FXML is a powerful, but somewhat pernicty XML based UI description language which needs to be treated carefully.

The UI fxml files are found in the *src/resources* directory, with the general convention that the resource file name is the lower case name of the class that instantiates it. To add a new agent, open the *src/resources/threadneedle.fxml* and add a new agent description to the *<children>* of the *LeftMenu*. The recommended approach is to copy and paste from an existing agent and modify carefuly. Example code is shown below for the Farm agent in Figure 3.2

Listing 3.2: Agent FXML

```
<SimNode fx:id="Farm" onDragDetected ="#mouseDragDetected"
                      onMouseReleased="#mouseDragDropped"
                      onMouseDragged ="#onDragOver">
  <image>
     <Image url="@images/F.png" preserveRatio="true" smooth="true"/>
  </image>
  <type>Farm</type>
  <tooltip>Food producer - auto-adds Food market</tooltip>
  <properties product="Food" initialDeposit="100" labourInput="5"/>
</SimNode>
```

The image file (@images/F.png) in the above example, provides the Icon displaying the agent in the UI. The *<type>* of the agent is the agent's class name, and *<tooltip>* is the rollover help text for the agent, and the *<properties>* tag is an arbitrary set of default values which can be provided for the Agent and be passed through to the Agent at instantiation.

Chapter 4: Statistics

Threadneedle provides a simple statistics class which allows the simulation and individual agents to maintain a round by round record of selected variables. Caution is advised with the statistics capability, since it can create scaling issues for the simulation if too many statistics are being maintained for large numbers of agents. (These will typically manifest as memory issues.) Statistics can be visualised using the Charts package as described below.

4.1 Adding Statistics

By convention, all statistics defined in Threadneedle are prefixed with *s_*. There are four different types, set when the statistics is created, which determines the operation performed by the *add()* method when used on this statistic:

COUNTER All values added to statistic are summed

AVERAGE Computes the average of values added

SINGLE Single value, sucessive adds() in same round overwrite value

NUMBER Cumulative total of add/subtracts maintained between rounds

Statistics are managed using *static* methods within the Statistic class. To create a new Statistic assign a reference name for the statistic and specify the type:

Listing 4.1: Statistic

```
s_mkt_price = new Statistic(name, Statistic.Type.AVERAGE);
```

The constructor will check the existing statistics and if a statistic with that exact name already exists in the simulation, then a reference to that statistic is returned, rather than a new object being created. This allows statistics to be shared between agents if required.

In order to provide a common interface between loading simulation files and creation using the GUI, statistics initialisation is put into its own method which is invoked by both the constructors.

4.2 Linking Charts to Statistics

Charts are defined in the fxml file: *src/resources/chartcontroller.fxml*. Statistics that are used by charts should be first defined in fxml file, and accessed using the *Statistic.getStatistic(name, group, type)* interface, which in addition to providing support for statistics also allows separate statistics to be part of a group shown on a single chart. For example:

Listing 4.2: Statistic fxml

```
<StepChart fx:id="production" title="Production" enabled = "true"
           prefHeight="250.0" prefWidth="250.0">
   <statistics>
       <FXCollections fx:factory="observableArrayList"> </FXCollections>
   </statistics>
</StepChart>
```

The fx:id of the statistic in *chartcontroller.fxml* must match the group specified in getStatistic(). The group specifier allows multiple different statistics to be shown on the same chart, by using the same group. The Statistic type and name are as explained above.

Chapter 5: Experiment Tagging

Git tags are used to associate particular builds with the experiment reports. This provides a way to cross-check experimental results with software changes, in case the outcomes of an experiments changes with later builds. Assuming the currently checked out branch is the desired one to tag, then the commands to do this are:

Create annotated tag:	git tag exp-b1 -m 'Builder Experiment 1'
List known tags:	git tag -n
Checkout tag:	git checkout exp-b1
Delete tag:	git tag -d exp-b1

Note that in order to propagate the tag into other repositories, the command, *git push –follow-tags* must be used.