# EOS Documentation

By Michael Adelson

## Model Organization

EOS intends to be, first and foremost, a model of an economy. Real-world economies are driven and influenced by millions of factors, from individual purchase decisions to high-level asset trading to political initiatives and weather patterns. Academic approaches to economics, however, often rely on massive assumptions and simplifications in order to keep their models manageable. EOS takes a middle road, neither forcing upon itself the impossible task of perfectly modeling a real economy nor building in assumptions in a way that constrains later refinement and specification.

### The Framework

The key to this flexible structure is EOS's core model framework, a set of "primitives" which we believe can be used to describe nearly every type of economic activity (with one notable exception—see Future Projects). The primitives are: Agents, Markets, and Goods.

#### Agents

Agents distinguish themselves because they represent decision-making units. In addition to making decisions, Agents own Goods and can die (be removed from the simulation). People, as well as corporate entities like firms and banks, can be modeled as Agents.

#### Markets

A Market is something which enables the exchange of two Goods, called the currency and the product. Offers are made using two parameters: a maximum quantity of product and unit price (in currency). The seller sets the minimum price at which he or she is willing to sell; the buyer sets a maximum. The Market's job is to somehow perform transactions based upon the various offers it receives. There are many types of real-world Markets; it is our hope that our most or even all of them can be put in terms of our Market primitive.

#### Goods

A Good is an object that represents a quantity of some commodity owned by an Agent. In the baseline, for example, every Agent owns a single Food Good which stores a variable quantity of food. In many ways, the Good primitive is a very general one. For example, it can represent both discrete (like a tractor) and continuous (like money) goods. However, Goods make one very important assumption about the commodities they represent: in Market transactions, two Goods vary only in their quantity. That is, from the Market's perspective, the 3.0 food stored in one Food Good is identical to the 3.0 food stored in another; thus there is no concept of differing quality. This is a common assumption used in economics, and it is very reasonable for many types of basic commodities, such as food, money, land, and natural

resources. Furthermore, this assumption is important for computational purposes: matching up and making buy offers and sell offers in a Market with two parameters (quality and price) is *much* more difficult than working with price alone. If in the future it becomes necessary to add variable-quality commodities to EOS, this can be done through modification of the Good primitive.

**Economy**

Although Economy.java is part of the framework package, the Economy class is really not one of EOS's primitives. Instead, Economies are responsible for coordinating and driving the simulation (see Economy under Code Structure).

### Goals—the Model

EOS's basic economic model is designed to achieve several goals which its predecessor, Minsim, did not. The first and foremost of these goals was that EOS should be simple to understand. By limiting the framework to a very small number of primitives, EOS should be easy to learn, add to, and explain to others. Through creating extremely general primitives, EOS also achieves the goal of being relatively complete, creating an economic modeling platform on which an enormous variety of industries, market structures, and strategies can be modeled.

### Future Projects

Although the baseline model has a lot of room for interesting experimentation, there are two major framework-level projects that remain to be done. They are explained here.

- Contracts: The addition of Contracts, representing agreements between Agents, would bring the model to a relatively complete state. Contracts would allow users to model loans, employment contracts, stock/joint firm ownership, all of which are crucial components of a complete economic model.
- Birth: The current model allows Agents to die, but not to be born. This means that populations can only decline, which makes long-term stability difficult to judge. Adding reproduction to EOS is an interesting challenge from a modeling perspective. Should Agents strategically choose when to have a child or should it happen randomly? How should children gain their initial allotment of Goods? Entrepreneurship (the "birth" of Firms) seems easier to model but requires the implementation of ownership contracts; Agents who acquire the requisite "startup costs" could query the Economy to create a new Firm; the Economy would establish the founding Agent as the owner.

## Code Structure

To correctly and efficiently modify EOS, it is necessary to fully understand the structure of the underlying code-base. Conceptually, we have split the code for EOS into three distinct levels of abstraction: framework-level, model-level, and strategy-level code. The three levels create natural lines of dependency and polymorphism in the Java classes: strategy-level classes extend model-level ones, which in turn extend those in the framework. Maintaining and properly

utilizing this three-level structure will be an important prerogative of future contributors, as it is an essential element of EOS's clarity.

**The Framework Level**

Code at the framework level is responsible for creating the API and language of the model. Thus, framework programs are interfaces and abstract classes. The EOS framework is meant to be highly general and to provide minimal restrictions to model- and strategy-level implementations.

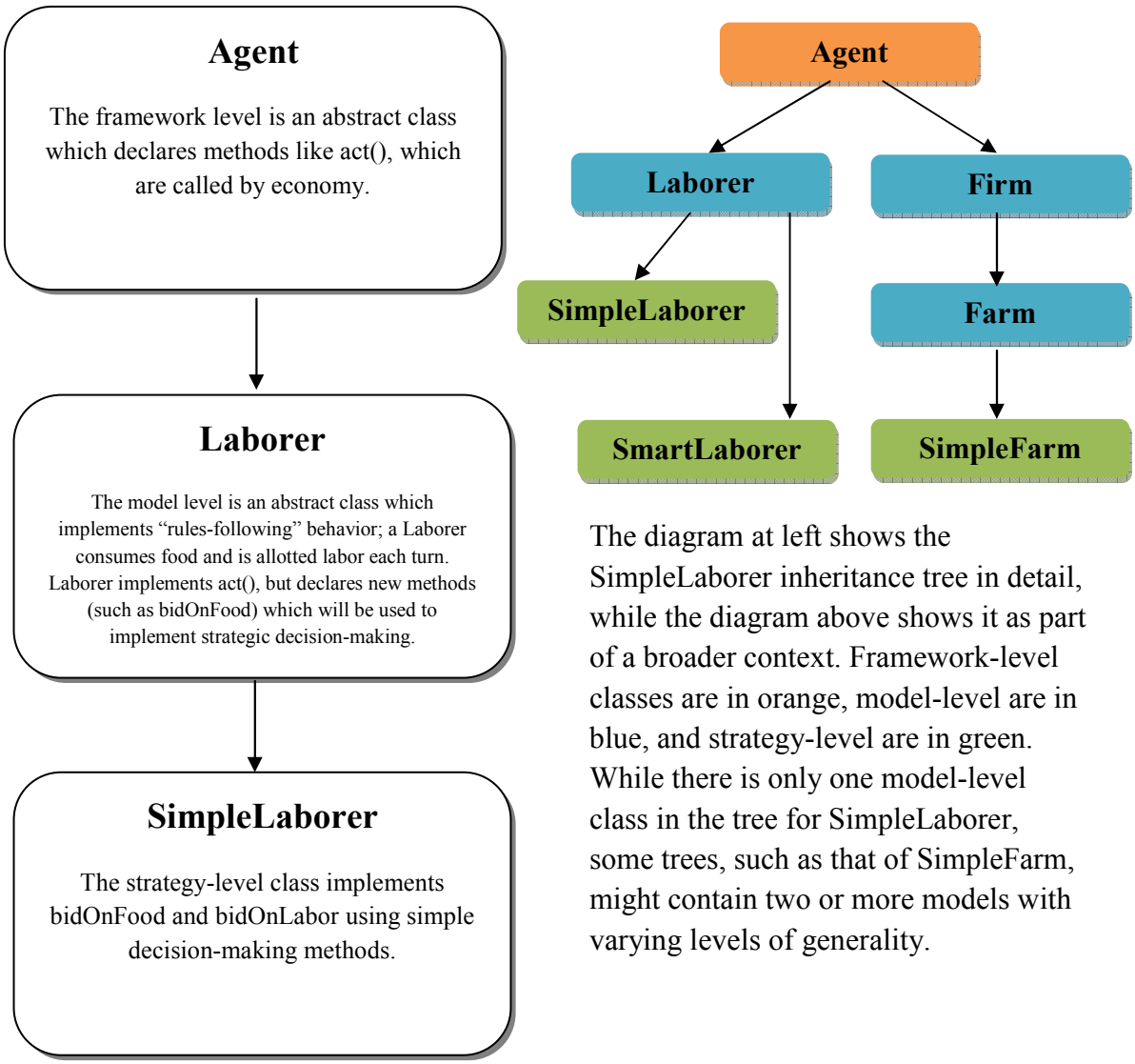
**The Model Level**

Model level code is responsible for enforcing "rules-following" behavior, such as forcing Laborers to eat each round or defining the production function for a Firm. Model-level code should be kept distinct from the framework because different approaches to modeling may implement very different behavior at the model-level while using the same framework.

**The Strategy Level**

Strategy level classes are specific, refined implementations of higher-level abstractions. In the case of Agents, strategy-level code will often contain the algorithms for the decision-making behavior, rather than the rule-following behavior of the model-level. For example, a SmartLaborer class might implement machine-learning algorithms to try to bid successfully in the Food and Labor markets, while a StupidLaborer class might bid randomly. Both, however, should extend the same Laborer model-level class which defines how much they must eat each turn, as well as how much labor they have to sell. A single simulation may contain many different strategy-level extensions of the same model for a particular type of Agent.

Here is a description of the levels used to implement SimpleLaborer, a basic "worker" Agent:




The diagram at left shows the SimpleLaborer inheritance tree in detail, while the diagram above shows it as part of a broader context. Framework-level classes are in orange, model-level are in blue, and strategy-level are in green. While there is only one model-level class in the tree for SimpleLaborer, some trees, such as that of SimpleFarm, might contain two or more models with varying levels of generality.

## Input and Output

EOS is designed to make setting input (customizing and running simulations) and getting output (simulation results) as easy and powerful as possible. In general, I/O is managed by the Economy class, which acts as a driver class for the simulation. Agents, Goods, and Markets are added to Economies on a class-by-class basis, which allows for a high level of customization. Output is in the form of CSV (comma-separated-value) text files, which can be opened in Excel and other programs for easy plotting.

### Economies

As the simulation driver, an Economy object is responsible for adding and keeping track of its constituent Agents and Markets. Economy methods can be complex to implement, since they involve numerous try-catch constructs as well as heavy use of the methods in the Java Class

class. Luckily, Economy methods are general; adding new types of Agents, Goods, and Markets does not require changes to the Economy class. Economies support customizable output via the addPrinter() method, which allows the user to have the Economy print data of a specific type to a specified file. Adding new "types" of printers is very simple (at least in the BaselineEconomy implementation), although the difficulty may vary depending on the difficulty of gathering the data required.

### Creating a Simulation Program

For anyone marginally familiar with Java, the easiest way to run a customized simulation using EOS is to write a "simulation program", which constructs an Economy, adds Agents, Markets, specifies output, and runs the simulation for the desired number of time steps. Simulation programs are short and simple. Here is an example:

```
/**
 * A sample simulation program for the baseline economic model. MySimulation
 * can be run from Eclipse, or by typing "java MySimulation" in the terminal/
 * command prompt in the directory where MySimulation is stored.
 */

package simulations;

import economies.*; // import the economies package (for BaselineEconomy)
import framework.*; // import the framework package (for Economy)

public class MySimulation {

    // create and execute the simulation
    public static void main(String[] args) {
        // the economy to be used
        Economy E = new BaselineEconomy();

        // initialize E

        // The names of Goods to be used
        String[] goods = {"goods.Food", "goods.Labor", "goods.Money"};

        // these arrays contain starting quantity values for the Goods in goods
        double[] qSL = {10, 0, 10}; // initial quantities for SimpleLaborers
        double[] qSF = {0, 0, 5}; // initial quantities for SimpleFarms

        // add 100 SimpleLaborers
        E.addAgents("agents.SimpleLaborer", 100, goods, qSL);

        // add 20 SimpleFarms
        E.addAgents("agents.SimpleFarm", 20, goods, qSF);

        // add a Food for Money Market
        E.addMarket("markets.CallAuctionMarket", "goods.Money", "goods.Food");

        // add a Labor for Money Market
        E.addMarket("markets.CallAuctionMarket", "goods.Money", "goods.Labor");

        // print the market prices at each timestep to prices.csv
        E.addPrinter("MarketPrices", "prices.csv", 1);

        // print the number of Agents alive every 5 timesteps to number.csv
        E.addPrinter("NumberAlive", "number.csv", 5);
```

```
        // run the simulation for 1000 timesteps
        E.run(1000);

        System.out.println("Simulation Complete!");
    }
}
```

### Future Project—Text File Input

To make things even easier for non-programmers, an important future project would be to develop a simulation program that uses a specified text file input format to create and run EOS simulations. This would create a veritable EOS language, and would increase maintainability and self-documentation as well as ease of use.

## Making Additions and Changes

Inevitably, future EOS users will desire to modify, extend, and refine it in a variety of ways. This section describes the general process through which such changes should be made. The most important step is to figure out which levels (framework, model, and/or strategy) your intended change modifies. It is important to remember that many changes will require fairly large modifications at the strategy level to properly take effect; if a new Good is introduced, but Agents do not know that it exists, they will not attempt to buy or sell it. Here are some examples of potential changes and the levels they affect, assuming you are running EOS with the baseline code (see Appendix A).

| **Change** | **F** | **M** | **S** | **Explanation** |
|---|---|---|---|---|
| Create a barter economy with a food for labor market. | | | | This can be set in the simulation program; it does not require changing the code base. |
| Add a SmartLaborer class that uses machine learning. | | | X | SmartLaborer will still extend Laborer, it will just implement the bid() methods more intelligently. |
| Add a new type of Market with strict price controls. | | | X | ControlledMarket will simply be a different implementation of Market. |
| Changing the production function for a Farm. | | X | | This only requires modifying the convertToFood() method in Farm. |
| Adding traders. | | X | X | Must create a new Trader() model class that extends Firm, as well as a SimpleTrader() strategy-level implementation. |
| Add gemstones, a commodity | | X | X | Must create a strategy-level Gem class, which extends ContinuousGood, as well as a model- |

| | | | | |
|---|---|---|---|---|
| which can be bought and sold. | | | | level class Mine, which extends Firm, and a strategy-level class SimpleMine as a basic implementation. Furthermore, Laborer Agents must be modified so that buying and selling Gems becomes part of their strategy. |
| Add Contracts as described in Model: Future Projects, and use them to implement Firm ownership. | X | X | X | Must define a new framework-level API for Contract, and create a strategy-level implementation of an OwnershipContract. Firm must also be modified to use OwnershipContracts (model-level), while Firm and Laborer Agents must be modified at the strategy level to make use of them. |

Once you decide which pieces of code will need to be written/modified, writing the code itself using EOS classes and methods is fairly simple. Look at the sample code from the baseline implementation, as well as the comments describing it, to get an idea of how EOS code for various classes can be written. The example and guidelines below may also be helpful.

**An Example Modification—Adding Tractors**

Let's say, for example, that you want to expand the EOS economic model by adding tractors. A tractor, you decide, is something that farms can purchase to increase production. Tractors are assembled in factories and sold on the free market. You take the following steps to implement tractors in EOS.

1. You decide that tractors come in discrete (integer) quantities. Thus, you create a new Tractor class in the goods package which extends DiscreteGood.

2. Tractors are produced in factories, so you create a model-level TractorFactory class which extends Firm. TractorFactory defines a production function. For example, you might decide that, for L units of labor, the factory can produce 5*sqrt(L) tractors. TractorFactory also declares two abstract methods which will be implemented by strategy-level classes: buyLabor() and sellTractors().

3. You next create a basic strategy-level implementation of TractorFactory by adding a new class, SimpleTFactory, to the agents package. SimpleTFactory extends TractorFactory, and implements buyLabor() and sellTractors() using decision-making strategies inspired by those in SimpleFarm. It might seem like you are done at this point, but there are actually two more steps to go. At this point, tractors can be successfully built because SimpleTFactories will purchase labor in the Money for Labor Market and use it to produce Tractors. However, there are no customers for these wonderful machines and they will rust on the shelves without a few more modifications!

4. Since Tractors increase Farm productivity, you change the convertToFood() method in Farm to take two inputs (tractors and labor) instead of one. You decide that each worker who uses a tractor is 50% more effective, so you change the production function from 5*sqrt(L) to 5*(sqrt(L-t) + sqrt(1.5*t)), where t = Math.min(L, number of tractors). You also change Farm's bidOnLabor() method to bidOnInputs(). Instead of modifying the existing Farm and SimpleFarm classes, you could of course also create a "parallel" farm class tree by writing, for example, a class TractorFarm which differs from the original farm class in the ways described above. In the long run, this approach may be preferable as it maintains old, but still functioning, versions of the code.

5. Finally, you modify SimpleFarm (or write a new class SimpleTFarm) to purchase and use tractors by expanding the bidOnInputs() method to make purchases in the Money for Tractor Market.

**Guidelines and Recommendations for Design**

When adding and modifying EOS code, we recommend that certain guidelines are followed to insure that the code remains maintainable and understandable.

- Comment extensively. At a minimum, every method should be explained through a comment with the form /**…*/ immediately preceding the method signature (this will appear blue in Eclipse). Comments should explain how the method should work as well as which inputs will cause it to throw exceptions.

- Do conscientious error-checking. When EOS methods receive improper inputs (for example, if a Market receives an offer with negative price), they should throw RuntimeExceptions rather than performing incorrectly. With so many interdependent pieces of code, this is the only way to maintain a development environment in which debugging is efficient.

- Maintain a log of changes by adding edit comments to a program's header comment (see CallAuctionMarket.java for examples). This supplements the SVN repository by making it easier to see how the code has changed over time.

- Don't change baseline code if you don't have to, especially if doing so would render useless other dependent classes. When adding to EOS, try to add new classes rather than change old ones whenever possible. This makes it easier for future users to pick and choose which modifications they want to include.

Goals—Code Structure

EOS's code structure was developed to meet the following design goals which we feel are necessary to insure its future usefulness.

- EOS is infinitely *extensible*. The general framework allows EOS's code base to extend in any number of directions.
- EOS is infinitely *refinable*. In creating the baseline implementations, we have made many assumptions about the nature of consumption, production, and decision-making. However, none of these assumptions are hard-wired into EOS. Thus, it is always possible to refine the correctness and complexity of EOS's model.
- EOS is *easy to understand*. We believe that the small size of the framework, as well as the adherence to the design motif of framework, model, strategy make the EOS system easy to understand. Our extensive commenting of each class and method should also help in this regard.
- EOS is *easy to use, regardless of one's level of programming experience.* The heavy commenting and thorough error-checking in EOS's base classes make its code easy to use as a base for new development. Furthermore, the user-friendliness of Java and the simplicity of the simulation programs required to run custom EOS simulations should make it accessible for users with limited programming experience.

# Appendices

### Appendix A: The Baseline Code

The so-called "baseline" model is consists of the initial code base created for EOS. The baseline implements a very basic economy; there are farms which buy labor and sell food, and laborers which sell labor and buy/consume food. While this model is extremely limited, it was an important step as it allowed us to test EOS's features in a restricted environment. Here is a graphical representation of the baseline:

# The Baseline Code