Object Oriented Programming and AB modelling

Alessandro Caiani[1]

Department of Economics and Social Sciences
(Università Politecnica delle Marche)
Ancona, Italy

AB-SFC Program Meeting
Monte Conero 2014

[1] a.caiani@univpm.it

Outline

Object-Oriented Programming [Minar et al., 1996]

A natural way of programming agents is to use an *Object-Oriented Programming language* (OOP) such as *Java*, *C++*, or *Python*.
Objects are program structures that hold **data and methods** operating on these data.

Members of a class

- Objects' data are attributes/characteristics stored in **fields** and can be of any type.
- Objects' method are **blocks of operations** that act on the data. They define the behavior of each object.

In OOP languages, objects are created from *templates* called **classes** which specify the type of data stored and the methods available to act on the data.

Object-Oriented Programming [Minar et al., 1996]

All the objects *instantiated* from the same class share the same methods and data structure, although their attributes values may differ.

The classes themselves are **arranged in a hierarchy**, with subordinate classes **inheriting** the methods and data of superior classes but **adding** additional ones or **replacing** some of the superior's attributes and methods with more specialized substitutes.

Class hierarchy

Modularity

The concept of modularity in OOP refers to organizing a structure where different elements of a software are divided into separate functional units.

Modularity leads to re-usability. If the components of the software are written in abstract way to solve general problems, they can be used in various applications. This allows to build a *hierarchy* of classes where classes can be ordered from general to specialized.
Hence OOP provides a practical way to **manage heterogeneity** among agents within and across the classes of agents populating our models.
OOP leads naturally to a useful **encapsulation**, with each agent clearly distinguishable within the program.

OOP and modularity [Minar et al., 1996]

In addition to being a natural way to implement multi-agent simulations, OOP is also a convenient technology for **building libraries of reusable software**.

Modelers can start to build models by directly instantiating useful classes from pre-existing libraries.

If there is no particular class that has the precisely needed behavior, one can take a preexisting class and specialize it, adding new variables and methods (or replacing some of them) via inheritance.

Modularity $\rightarrow$ **Easy to share** codes.

Outline

OOP and modularity

AB models may differ under several respect such as:

- the types of agents involved in the simulation;
- the types of markets (goods, credit, financial assets etc.) and stocks that are considered (deposits, cash, real capital, bonds, shares etc.);
- the sequence of events taking place within each period;
- the strategies followed by agents to take decisions, such as consumption, investment, pricing decisions;
- the mechanism through which transactions are cleared
- the scheme of amortization of capital
- ...

This raised an important issue related to the project design of our platform: in order to provide a general tool for AB-SFC modeler the platform should then allow to develop a high variety of models without having to intervene on its fundamental structure $\rightarrow$ Modularity

OOP and modularity II

We conceived our platform as a system composed of **rather independent functional blocks** communicating and interacting with each other while running the program.
Each functional block takes the **inputs to operate from some other blocks, elaborate them, and send its output to some other functional blocks** which use it as an input.
Each block does not need to know how the other blocks work in order to operate.
The only thing that matters is that functional blocks **agree to a a "contract" that spells out how the blocks interacts**. In OOP these contracts are called interfaces.
This allows to change the features of each functional block without having to intervene on the others and without compromising the functioning of the "system" and to develop each block without the need to know how the other blocks operate.

Interfaces: an example in JMAB I

Let's consider how a transaction between two agents is realized in JMAB:

- a **strategy of supply-agents** to determine their supply and a strategy to determine the price;
- a **strategy of demand-agents** to choose the counterpart and a strategy to set their demand;
- a **transaction mechanism** which compares desired demand and available supply, check that agents have enough resources to perform the transaction, and realizes the transfer.

To operate, **the transaction mechanism must** know what is the demand and supply, the prices, and the means of payment that will be used to clear the transaction and it have to **"ask to the agents involved in the transaction" these information**.

Interfaces: an example in JMAB II

But many types of agents may be involved in a certain type of transaction:

E.g. Households and consumption firms buying goods, capital and consumption firms selling their output, consumption and capital firms asking for loans, households and firms buying a deposit.
These **different agents usually have also different strategies** to set their demand/supply, the prices, and so on.

Therefore, the **transaction mechanism should be designed independently from the specific strategies followed by agents**, so that it can operate every time a certain type of transaction is realized (e.g. purchase of a good, emission of a new loan, purchase of a financial asset such as deposit, or a bond etc.) regardless the types of agents involved in the transaction.

Interfaces: an example in JMAB III

We define several interfaces representing the different roles agents can perform on the different markets: *Deposit Demander (Supplier)*, *Good Demander (Supplier)*, *Labor Demander (Supplier)*, *Bond Demander (Supplier)* etc.
Each interface contains the signature of the methods related to the role they represent: for example *"setGoodsDemand"* for a *Good Demander*, *"setLoanRequirement"*, *"decideLoanLength"*, *"decideLoanAmortizationScheme"* for a *Credit Demander*.

Interfaces: an example in JMAB IV

These methods are then implemented by the classes representing the different types of agents in our models: *Households*, *ConsumptionFirms*, *Capital Firms*, *Government*, *Banks*, *CentralBank* etc. For example *Households* are usually *LaborSupplier* and *GoodDemander*, they might also be *BondDemander*, *CreditDemander* etc.; *Firms* are usually *CreditDemander*, *LaborDemander*, *GoodSuppliers* etc.
Instead of developing as many transaction mechanisms as the different types of agents possibly involved in a certain type of transaction, **we can develop a unique transaction mechanism, which calls the methods defined by the demand/supplier interfaces** without having to know how they are implemented by the classes representing agents.

Modularity on Agents' Behaviors

- As we are **generally interested in analyzing different agents' strategies** (different pricing, investment, purchasing strategies for example) we keep the definition of these strategies separated from the definition of the classes. That is, **agents and their strategies are two independent blocks**.
  For this sake, we use again interfaces: e.g. a *InvestmentStrategy* defining a general method "*computeDesiredGrowth*" , a *FinanceStrategy* defining a method "*computeCreditDemand*" , *WageStrategy* defining a methods "*setAskedWage*" , a *TaxPayerStrategy*, a *SupplyCreditStrategy*, a *SelectWorkersStrategy*, a *SelectSellerStrategy* etc.
- This interfaces are then implemented by specific classes of strategies, representing concrete behaviors.

Modularity on Agents' Behaviors

- If one wants to analyze a new behavior by an agent, **he only have to develop the concrete class representing the new strategy without having to intervene in the class representing the agents that use that strategy**. The only thing to do is to inject the new strategy in the agents through the configuration file. This is called: **"Dependency Injection"**.

Outline

The logic of Dependency Injection

Roughly speaking, the **construction of the general attributes of classes and instances of agents is maintained separated from their configuration**, so that we can easily change agents' strategies, parameters value, simulations' initial conditions, without having to 'hard-code' them in the code representing the structure of the underlying simulation model.

That is, the general attributes of each agent (fields) are defined in the code representing the model, but the contents of these fields is defined outside the model and then injected into it through a separate XML file.

Dependency Injection and MC simulations

AB models are stochastic and executed as Monte Carlo simulations.

Therefore, we run the simulations many times to **check for across run volatility induced by stochastic terms and to perform sensitivity analysis.** Different draws of free parameters from a certain distribution, different distributions, or systematic increase in a certain parameter region ('parameter sweep').

Dependency Injection offers a way to express declaratively the configuration of the model without having to intervene in the code representing the model. The configuration information are stored in a separated file and then *injected* in the model when starting the simulation.

In this way we can write the code representing the agents **without having to make any assumption regarding how the properties** representing agents' attributes **are initialized**.

Dependency Injection and models' behavioral specification

Dependency Injection allows to **easily change the model configuration of behaviors**.

That is, we can easily change the configuration of: agents' routines, agents' learning algorithms, agents expectation formation mechanisms, pricing, investment, production rules, market protocols, market matching mechanisms, without having to hard-code anything in the model.

We only have to focus on the development of the new specific behavior we want to model (creating a new class) and then we can inject it through the XML configuration file $\rightarrow$ Modularity, Flexibility, Accessibility.

Sequence of Events

In most most cases, modellers have a specific sequence of events in mind, reflecting a particular vision (theory) of how the economy works.
A crucial feature for a platform aspiring to provide a **general, flexible and non theoretically-biased tool** should then be the neutrality regarding the definition of the sequence of events. Hence, in JMAB we adopted a different approach to define the sequence of actions during each period of the simulation, based on the use of specific objects called **events**.

Tick events

Events (General)

An event is an object sent from an object called *EventScheduler* to a number of objects called *EventListener*. The event may contain data used by listeners' to perform actions through their methods. It can thus be seen as a **message** being sent to all **listeners** that **subscribed** to an event.

In the case of JMAB, we created a class symbolizing tick events. Each one of these ticks represent a sub-period of one period of the simulation.
There are two types of tick event: **market** and **agent** and each tick event identified by a **unique id**.
These tick events are then sent to agents by the MacroSimulation ("*OrderedEventsSimulation*").

Here is how it works

The simulation has a list of ticks events stored in one of its fields ("events"). For each of these:

1. the simulation sends the tick event
2. the agents listening to the broadcasting channel (one for each type of tick event) receive the message
3. depending on the type of event and on value of the tick id
   - each agent uses zero, one or more methods to update its data
   - each market invokes agent interactions or not

Benefits

- the sequence of events is not imposed by the platform, but it is determined by the modeller
- the definition of the order of events is kept totally separated from the definition of the agents' methods. Hence we can change the ordering without having to change the blocks representing agents.
- it allows for the possibility of modelling different frequencies of economic processes (e.g. more than one market interaction per period).
- different accounting, production, and finance timings
- it gives the possibility of dealing with asynchronous decisions by agent (agents can decide whether to react or not to a specific tick event).
- next developments: agents sending events, not only reacting.

Conclusion

The platform does not superimpose any structure to the model, leaving the modeller full flexibility in defining the sequence and timing of events

Fundamental representation of an agent

Mother class

Each agent, may it be of the type household, firm, bank or government, is endowed with a stock matrix, representing its real and financial stocks, subdivided into assets and liabilities. This *StockMatrix* is inherited from the superior *SimpleAbstractAgent* class.

The structure of the stock matrix:

- two lists representing assets and liabilities containing
  - as many slots as there are types of stocks (each one identified by a specific index), containing
    - Lists of **Items** representing stocks of a certain type (e.g. capital, durable consumption goods, cash, loans, deposits, bonds, shares etc. held by an agent

The Stock Matrix and agents' Balance Sheet

Item

All stocks are object of the abstract class *Item*, which is then specialized by a number of concrete classes representing specific types of stocks. All **real goods** contain generic and specific information regarding, for example, its owner, its producer, its price, its age, or its productivity. All **financial assets** contain generic and specific information regarding, for example, the liability holder and asset owner, its value or its interest rate.

The **stock matrix is then deeply related to the representation of agents' balance sheet** (though being a broader concept), as it contains all the information required to compute it.

Indeed, we can easily access the accounting value of the various stocks, aggregating them for each type of asset/liability, and then summing them up in order to derive the total value of assets and liabilities, with the net wealth as balancing item.

Benefits

- flexible and efficient tool to store information and manage heterogeneity
- deeply related to the representation of agents' balance sheet
- network representation of stock-matrix interdependencies via asset-liability duality of each financial item
- helps in ensuring the stock-stock consistency. Every time a stock is updated (e.g. a share of a loan is repaid), also the balance sheets of agents holding it as an asset/liability are automatically updated.

Flow-flow and Stock-flow consistency

Both consistencies are guaranteed by the **transaction mechanisms** such as *AtOnceMechanism* (for purchases of real/financial goods), ConstrainedCreditMechanism (for loans), DepositMechanism, ReserveMechanism (for banks' cash advances asked to the CB).

- The transaction mechanism first of all assesses whether the chosen supplier has enough supply to satisfy agent's demand. If not we have a supply-constrained transaction.
- In the case of a purchase of a good or financial assets:
  - the mean of payment accepted by the supplier is identified (e.g. cash or deposit transfer)
  - the transaction mechanism assesses whether the buyer has enough liquidity in form asked by the seller. If not, an internal transfer is made (e.g. from agent's deposit to cash or vice-versa). If still not enough the transaction is financially-constrained.
- The transaction mechanism updates the Stock Matrices of agents' (e.g. seller's inventories and buyers stock of that good, seller's and buyer's cash/deposits) ensuring that the flows arising out of the transaction respect *Copeland's quadruple entry system*.

From micro to meso to macro

JMAB...

1. provides a **comprehensive representation of the complex dynamic network linking agents' balance sheets** and its evolution through time as transactions between different agents occur. This allows:
   - network analysis
   - fragility analysis
2. provides an **overall balance sheet and flows of funds representation** of the economy allowing for
   - more traditional analysis at the macroeconomic level
   - synthetic measures to assess the state of the economy and its future evolution

Conclusion

JMAB **allows the researcher to browse through all the possible levels of aggregation according to his needs**, thus providing a fully scalable vision of the functioning of the simulated economy

References I

N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. *Santa Fe Institute Working Paper Series*, 96-06-042:–, 1996.