JMAB and Benchmark user guide

Alessandro Caiani and Antoine Godin

This version: 1.0 Thursday, 18 February 2016

TABLE OF CONTENTS

1 Introduction, installation and running JMAB

This document is intended to help the user to use JMAB and any subsequent model. For now only the benchmark model described in Caiani et al. (2015) is publicly available. The document is structured in the following way: this section introduces JMAB, how to install it, run it and collect data. Section 2 will provide a general description of JMAB, in terms of software structure and tic workflow. Section 3 and 4 will describe the various source folders and the class and interface they contains for JMAB and for benchmark respectively.

1.1 Java Macro Agent-Based toolkit (JMAB) and its origin

JMAB is built on top of JABM, the Java Agent-Based Modelling Toolkit developed by Steve Phelps (see here for more information). JMAB has kept JABM's structure but is adding the features necessary for the modelling of a stock-flow consistent macro-economy (i.e. more than one market, more than two populations, multiple strategies, balance sheets and transaction mechanisms). JMAB has extensively used the modularity principle of object oriented programming, by extending most of the original class from JABM. This explains why the code structures are very similar in JABM and JMAB.

Each specific model built on JMAB will have to develop distinct class, adapted to the particular context of the model. The benchmark model follows this rule. However, since each model is built on top of JABM/JMAB, the code structure will again be similar, but contain much less classes since most of the code is already present in JABM/JMAB. JABM will not be discussed all the relevant information can be found on the dedicated websites: http://jabm.sourceforge.net/ and https://github.com/phelps-sg/jabm.

1.2 Installation

Antoine Godin 18/2/2016 21:22
Comment [1]: To update

1-Install, if you haven't already, a Java Development Kit (JDK) on your PC:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

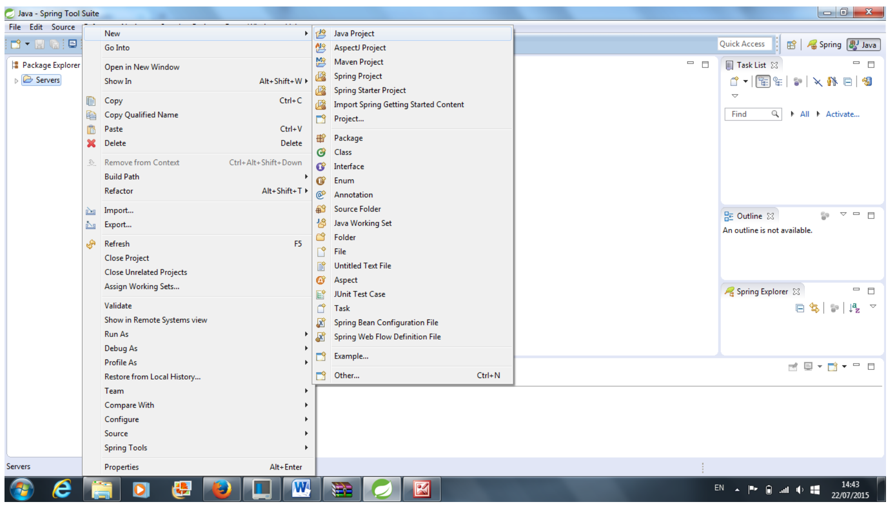
2-Install Eclipse or the Eclipse-based Spring tool suite (STS). Both are fine to inspect and run the code but the latter provides several additional features which help in writing and inspecting the .xml configuration files, plus graphical representation of the "Beans" dependencies structure. So you may find convenient to install the STS. If you have already installed Eclipse on your PC, you may also consider to update your Eclipse to the latest release of the Spring Tool Suite following the instructions on the STS website.

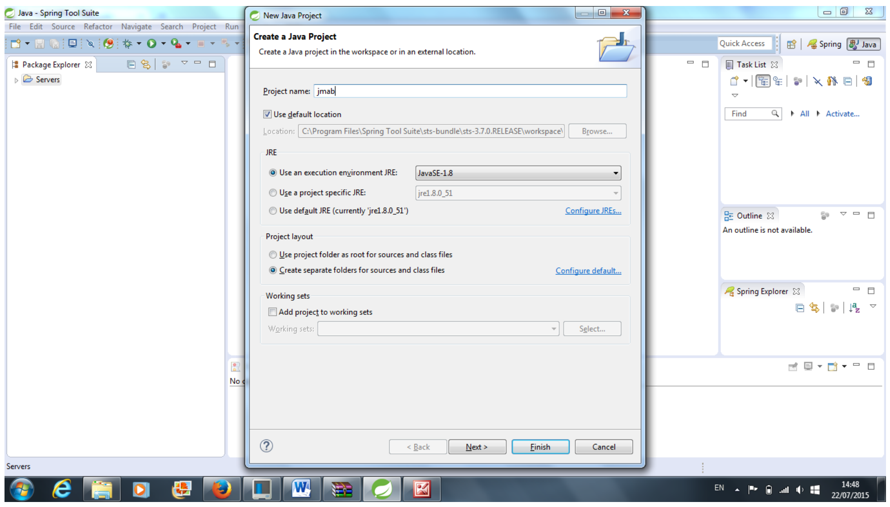
https://eclipse.org/downloads/

https://spring.io/tools

3-After installation is completed, launch Eclipse or the STS and specify your workspace directory.

4-In the "Package Explorer" window on the left create a new Java Project by right-clicking and then selecting New-Java Project. Type "jmab" in the Project name field. Then press Finish. A jmab project should now appear on your Package Explorer.





Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: jmab

[✓] Use default location

Location: C:\Program Files\Spring Tool Suite\sts-bundle\sts-3.7.0.RELEASE\workspace\ Browse...

JRE

(●) Use an execution environment JRE: JavaSE-1.8

( ) Use a project specific JRE: jre1.8.0_51

( ) Use default JRE (currently 'jre1.8.0_51') Configure JREs...

Project layout

( ) Use project folder as root for sources and class files

(●) Create separate folders for sources and class files Configure default...

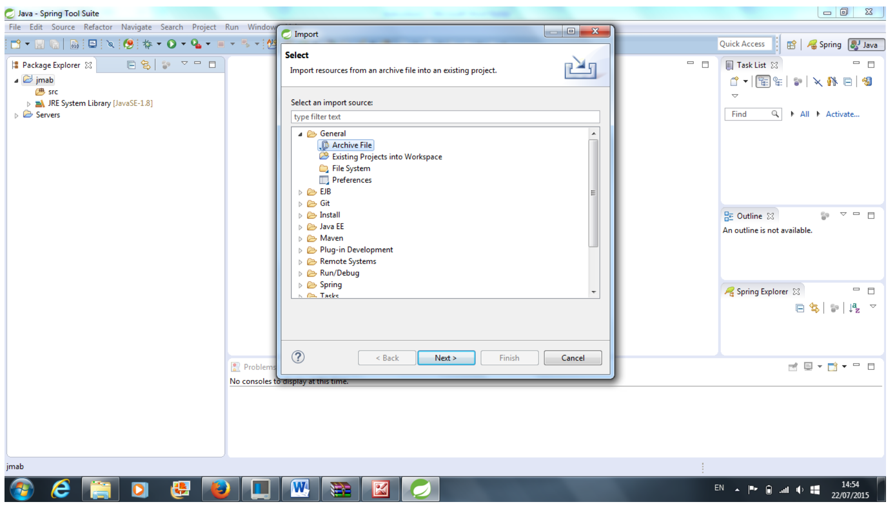
Working sets

[ ] Add project to working sets

Working sets: Select...

< Back Next > Finish Cancel

5-Now select you jmab project, right-click and choose Import...-General-Archive File. Then press Next.



Import

Select

Import resources from an archive file into an existing project.

Select an import source:

type filter text

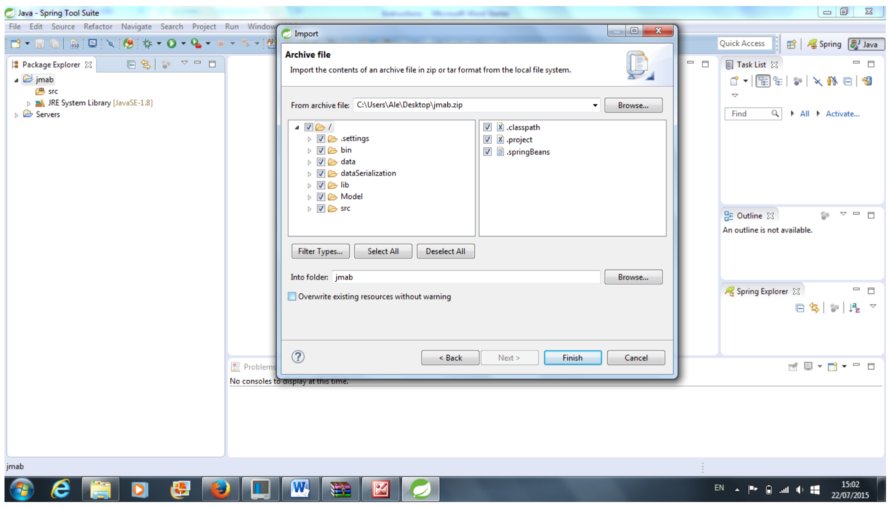
- General
  - Archive File
  - Existing Projects into Workspace
  - File System
  - Preferences
- EJB
- Git
- Install
- Java EE
- Maven
- Plug-in Development
- Remote Systems
- Run/Debug
- Spring
- Tasks

jmab
- src
- JRE System Library [JavaSE-1.8]
- Servers

An outline is not available.

No consoles to display at this time.

4-In the "From Archive file" field specify the path where you saved the jmab.zip archive. Please verify that all the contents of the archive are checked and that the project folder chosen is jmab before pressing Finish. After this, a window should open asking "Overwrite '.classpath' in folder 'jmab'?". Press "Yes To All". Your jmab project should then display both the source code and the configuration files of our platform and model.



Import

Archive file

Import the contents of an archive file in zip or tar format from the local file system.

From archive file: C:\Users\Ale\Desktop\jmab.zip

- [✓] /
  - [✓] .settings
  - [✓] bin
  - [✓] data
  - [✓] dataSerialization
  - [✓] lib
  - [✓] Model
  - [✓] src

- [✓] .classpath
- [✓] .project
- [✓] .springBeans

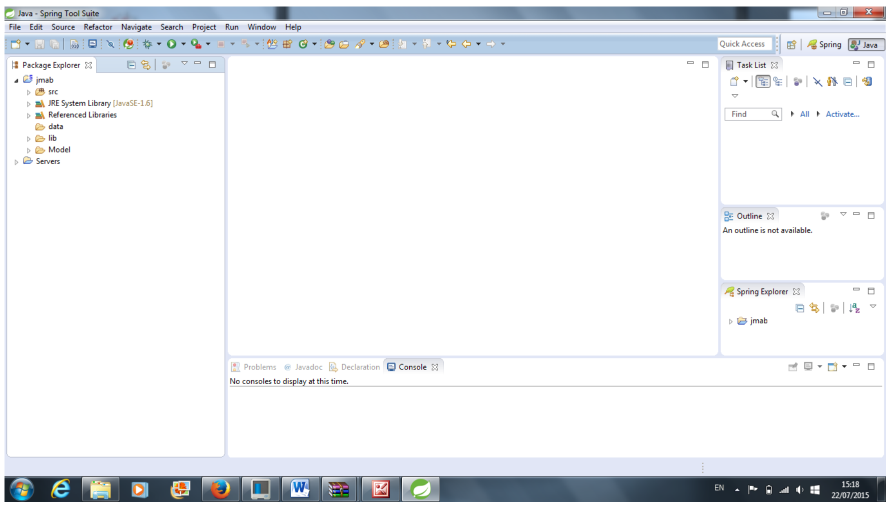
Into folder: jmab

[ ] Overwrite existing resources without warning

An outline is not available.

No consoles to display at this time.

5-Before proceeding, please check that the "data" folder appears in your jmab project. If this is not the case, please create it manually by right-clicking on the jmab project in Eclipse/STS Package Explorer-New-Folder. The folder "data" is the where all the .csv files generated by the simulations will be stored. At the end of the installation procedure your jmab package should look like in the picture hereunder.



An outline is not available.

No consoles to display at this time.

1.3 Create a Java Application to run or debug the benchmark model

1. Select “Run Configurations” from the “Run” menu on the upper toolbar. Select “ Java Application” and then press the “New” button.

2. In the “Main” window specify

- The name of your new configuration, for example “Model_Baseline”.
- The Project: benchmark
- The Main class: benchmark.Main

3. In the “Arguments” window specify (the Xmx and Xms allocate more RAM to the java process, where 2G assigns 2 gigabytes of RAM):

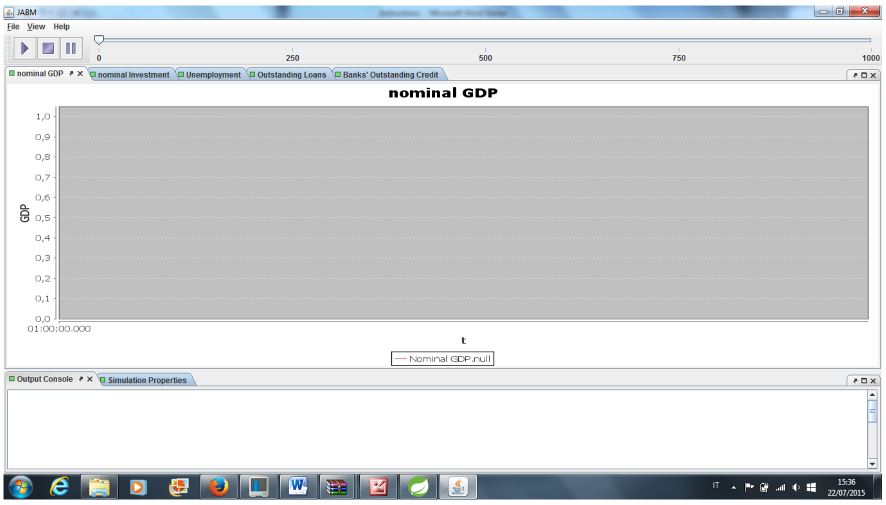
- VM arguments: -Djabm.config=*pathToConfigurationFile*.xml –Xmx2G –Xms2G
- Where the *pathToConfigurationFile.xml* is the path to the configuration file (e.g. Model/mainBaseline.xml).

4. Then press “Apply” and “Run” to launch the simulation.

If everything goes smoothly, the following script should appear on your Eclipse/STS Console, specifying that the various configuration files are being loaded to instantiate the beans of the simulation:

```
Loading XML bean definitions from file [C:\Program Files\Spring Tool
Suite\sts-bundle\sts-
3.7.0.RELEASE\workspace\jmab\Model\mainBaseline.xml]
Loading XML bean definitions from file [C:\Program Files\Spring Tool
Suite\sts-bundle\sts-
3.7.0.RELEASE\workspace\jmab\Model\modelBaselineMonteCarlo.xml]
Loading XML bean definitions from file [C:\Program Files\Spring Tool
Suite\sts-bundle\sts-3.7.0.RELEASE\workspace\jmab\Model\reports.xml]
```

5. After a few moments a Java Console automatically opens: click Play to start the simulation.



6. The graphic interface only provides a representation of some variables (e.g. Nominal GDP, Unemployment, Banks' Total Credit). All the other outputs of the simulation are generated in the "data" folder as separated csv files, 1 for each variable. Click "Refresh" on the data folder to see them.

1.4 Sensitivity experiments

Antoine Godin 18/2/2016 21:21
Comment [2]: To update

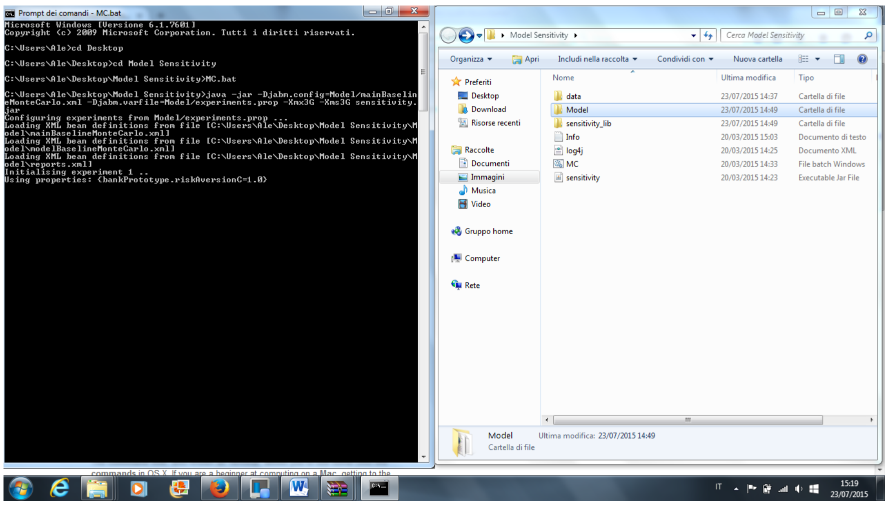
In order to perform the sensitivity experiments presented in the paper, we used the jar executable contained in the Model Sensitivity archive. This is by far the fastest solution to run Monte Carlo Simulations and it allows setting up the sensitivity experiments through the use of a separated properties file defining the ranges of variation and the increment for each parameter.

To set up a sensitivity experiment open the "experiments" property file contained in the" Model" folder with a text editor and de-comment the line(s) referring to the parameter(s) of interest. The 3 numbers following the parameters names (expressed in the "class.field" format, as they appear in the source code) are the minimum value, the increment and the maximum value for the parameter. Note that if you un-comment more than one variable, the Monte Carlo simulation will combine all possible cases of variable values (i.e. exponential number of possible sets).

To launch the experiments open your Command Line, change the directory to the "Model Sensitivity" folder, and then launch

- the MC.bat file - as in the example hereunder - in case you are using Windows OS,
- the MonteCarlo.sh file in case you are using Mac OS .

To stop the simulations just use the type CTRL+C in the Command Line.



```
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Users\Ale>cd Desktop

C:\Users\Ale\Desktop>cd Model Sensitivity

C:\Users\Ale\Desktop\Model Sensitivity>MC.bat

C:\Users\Ale\Desktop\Model Sensitivity>java -jar -Djabm.config=Model/mainBaselin
eMonteCarlo.xml -Djabm.varfile=Model/experiments.prop -Xmx3G -Xms3G sensitivity.
jar
Configuring experiments from Model/experiments.prop ...
Loading XML bean definitions from file [C:\Users\Ale\Desktop\Model Sensitivity\M
odel\mainBaselineMonteCarlo.xml]
Loading XML bean definitions from file [C:\Users\Ale\Desktop\Model Sensitivity\M
odel\modelBaselineMonteCarlo.xml]
Loading XML bean definitions from file [C:\Users\Ale\Desktop\Model Sensitivity\M
odel\reports.xml]
Initialising experiment 1 ..
Using properties: {bankPrototype.riskAversionC=1.0}
```

Nome	Ultima modifica	Tipo
data	23/07/2015 14:37	Cartella di file
Model	23/07/2015 14:49	Cartella di file
sensitivity_lib	23/07/2015 14:49	Cartella di file
Info	20/03/2015 15:03	Documento di testo
log4j	20/03/2015 14:25	Documento XML
MC	20/03/2015 14:33	File batch Windows
sensitivity	20/03/2015 14:23	Executable Jar File

1.5 Data

Csv files names should be self-explanatory. The number at the end of the file name refers to the number of the simulation run.

The first number in each row of a csv file represents the simulation period. The following numbers refer to the value of the variable for each agent in that period.

Some files refer to aggregate or average values of a variable (in the whole economy or in a specific sector, as specified by the name). In these cases the abbreviations “Agg” (aggregate) or “Av” (average) appear in the name of the .csv file.

The files aggBBS, aggCBBS,aggCFBS, aggKFBS, aggGBS, aggHHBS refer to the Balance Sheets of the banking, central bank, consumption firms, capital firms, government, and households sectors. Each line contains the period number followed by a sequence of numbers reporting the value of each type of stock for that sector (liabilities appear with a minus sign). The order in which stock values appear reflects the index assigned to the correspondent type of stock in agents' stock matrix, which is defined in the modellone.StaticValues Interface:

```java
public static int SM_CASH=0;
public static int SM_DEP=1;
public static int SM_CONSGOOD=2;
public static int SM_CAPGOOD=3;
public static int SM_LOAN=4;
public static int SM_BONDS=5;
public static int SM_RESERVES=6;
public static int SM_ADVANCES = 7;
```

The TFM.csv file provides a synthetic representation of the nominal flows generated during each period of the simulation, which can be used to derive the aggregate Transaction Flow Matrix (TFM). Similarly to the logic of construction of the .csv files for sectoral balance sheets, the order in which flows appear in each line, depends on the index assigned to each type of flow in the modellone.StaticValues interface:

```
public static int TFM_CONS=0;(Households’ consumption)
public static int TFM_WHH=1; (Households’ Wage)
public static int TFM_DOLEHH=2;(Households’ dole)
public static int TFM_THH=3; (Households’ taxes)
public static int TFM_IDHH=4; (Households’ interests on Deposits)
public static int TFM_DIVHH=5; (Households’ Dividends)
public static int TFM_CONSCF=6; (C firms’ sales)
public static int TFM_WCF=7; (C firms’ wages)
public static int TFM_TCF=8; (C firms’ taxes)
public static int TFM_INVCF=9; (C firms’ investment change)
public static int TFM_CACF=10; (C firms’ capital amortization)
public static int TFM_IDCF=11; (C firms’ interests on deposits)
public static int TFM_ILCF=12; (C firms’ interests on Loans)
public static int TFM_DIVCF=13; (C firms’ dividends)
public static int TFM_RECF=14; (C firms’ retained earnings)
public static int TFM_WKF=15;(K firms’ wages)
public static int TFM_TKF=16; (K firms’ taxes)
public static int TFM_INVKF=17; (K firms’ sales in inventories)
public static int TFM_IDKF=18; (K firms’ interests on deposits)
public static int TFM_ILKF=19; (K firms interests on loans)
public static int TFM_DIVKF=20; (K firms’ Dividends)
public static int TFM_REKF=21; (K firms’ Retained Earnings)
public static int TFM_TB=22; (Banks’ taxes)
public static int TFM_IDB=23; (Banks’ interests on deposits)
public static int TFM_IBB=24; (Banks’ interests on bonds)
public static int TFM_IAB=25;(Banks’ interests on advances)
public static int TFM_ILB=26; (Banks’ interests on loans)
public static int TFM_DIVB=27; (Banks’ Dividends)
public static int TFM_REB=28; (Banks’ Retained Earnings)
public static int TFM_WG=29; (Government’s wages)
public static int TFM_DOLEG=30; (Government dole)
public static int TFM_TG=31; (Government’s taxes)
public static int TFM_IBG=32;(Government’s interests on bonds held by
banks)
public static int TFM_FCBG=33;(Government Profits from CB)
public static int TFM_IBCB=34;(Central Bank’s interests on Government
Bonds)
public static int TFM_IACB=35;(Central Bank’s interests on advances)
public static int TFM_FCB=36; (Central Bank’s profit)
public static long TFM_DINVENTCF = 37; (Variation in Consumption Firms’
Inventories)
public static long TFM_DINVENTKF = 38; (Variation in Capital Firms’
Inventories)
public static int TFM_SIZE=39;
```

2 General description

2.1 Software Structure

The software used the SpringFramework to initialize the various java objects needed for the simulation. The SpringFramework is based on the concepts of “beans” which represent the objects to be created and contains the value or the references for all the objects contained in each object.

The combination of the three xml files (mainBaseline.xml, modelBaseline.xml, and reports.xml) containing the bean definition of each object of the simulation determines the structure of the software. Figure 1 shows a simplified version of this structure. Each arrow in Figure 1 represents a “has an object of the type” relationship. The origin of the link is empty while the end of the link contains an arrow and a number representing the number of objects contained: 1, 2 or n.

The SimulationController is the object managing the whole simulation. It contains an object of the type MacroSimulation, which is the core simulation. It also contains a list of all the reports, which will generate either csv files or plot time series on the DesktopSimulationManager window. Finally, the simulation manager contains a list referencing all the objects (called listeners) that subscribe to the diffusion canal used to send events (see hereunder).

The MacroSimulation object contains a list of TicEvents, which will be send to the agents or markets via the simulation controller. These tics are defined within the MacroSimulation class because they are specific to each implementation of a model. The MacroSimulation object also contains a list of MarketSimulation objects which will be used to simulate each specific market (consumption good market, deposit market, credit market, and so on) and an object of the type MacroPopulation which represent the set of all the existing agents, grouped into populations.

The MacroPopulation object contains a list of Population objects which contain all the agents of a certain type (i.e. households, consumption good firms, banks, governments, etc.). Each specific agent is of a class extending the class SimpleAbstractAgent. This mother class contains a set of variables common to all agents: the StockMatrix (see after), a map containing strategies, a map containing lagged values (of the type PassedValues) and a map of expectations. These three maps contain objects specific to each agent’s type (i.e. the banks may contain strategies for interest settings, and credit supply, passed values about non performing loans and profits and expectations about deposits while households contain strategies to select consumption good suppliers and wage setting, passed values about consumption prices and employment and expectations about consumption prices).

The stock matrix contains all the assets (real or financial) and liabilities (only financial) that each agent owns. It consists of two lists (assets and liabilities) of lists of Items. If an agent does not own a specific asset (i.e. reserves for households or consumption goods for banks), the list relative to that asset will be empty. All the objects contained in the stock matrix are of a type extending the abstract class AbstractItem and thus contain at least a value, a quantity an age and a reference to its position in the stock matrix. Furthermore, each Item contains a reference to its asset holder and its liability holder. For real goods (such as consumption or capital goods) the liability holder refers to the producer.

Each MarketSimulation object contains 3 objects: a mixer, a transaction mechanism and a market population. The market population contains references to the populations containing the potential demanders and suppliers (i.e. households and firms for the consumption good market, firms and banks for the credit market). The mixer is the mechanism specifying how demanders and suppliers will be interacting: how demanders are selected and in which order, and with whom will they interact (i.e. the whole supplier population or only a subset). Furthermore the mixer determines when the market is closed (i.e. when there are no more demanders or suppliers). Finally, the transaction mechanism realizes the transaction, once a demander and a supplier have been selected. This consists in determining the price and quantity of the transaction (depending on various factors such as availability of funds or supply quantities, amount demanded, etc.) and then updating the stock matrix of all the agents involved in the transaction (these can be the two active agents but also any other passive agents whose balance sheets are impacted due to clearing mechanisms).

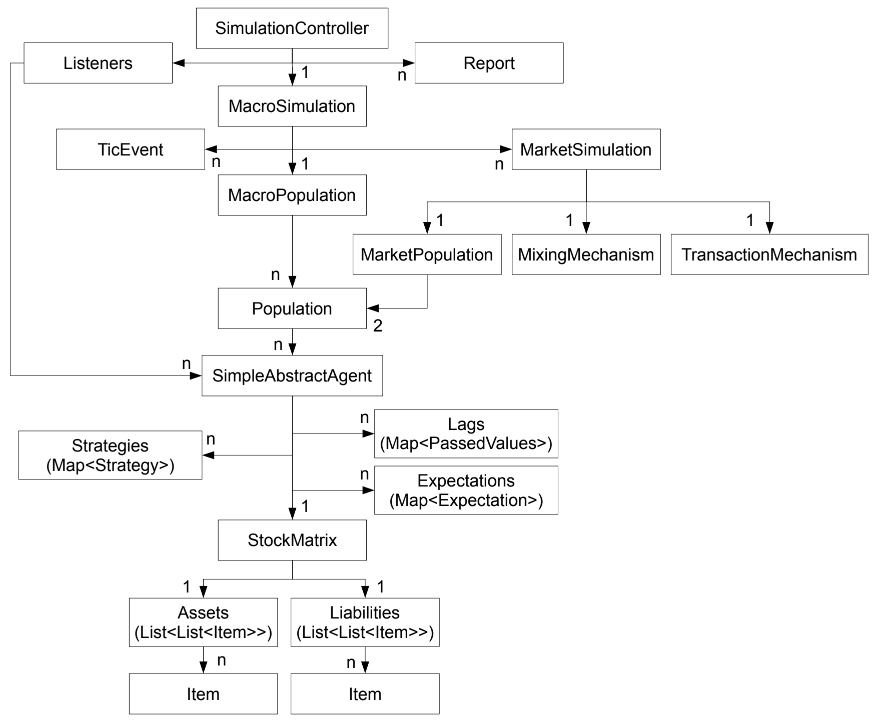


Figure 1 Software Structure

2.2 TicEvent effects

Figure 2 shows the two type of effects a TicEvent can trigger. The TicEvent can be of four types: MacroTicEvent, MarketTicEvent, MacroVariableTicEvent and MicroMultipleVariableTicEvent. The last two refer to reports and will not be detailed here. MacroTicEvent are general events that will trigger agents, as shown at the tope of Figure 2. Each agent listening to the SimulationController will compare the value of the tic with its own list and determine whether it needs to do something (i.e. update its expectation, determine its credit demand or the interest rate it will charge). These tic do not lead to agent interactions (except for interest or tax payments but these are actions based on previous interactions).

The MarketTicEvent will not trigger agents but MarketSimulation objects. As for agents, each market compares the value of the tic with its own list of values and determines whether it needs to do something or not. One the market has been activated; it asks the mixer to start the interactions. The order in which parties in the market are activated depends on each mixer implementation. We use various versions of the market mixer (called RandomRobinMixer) that activate demanders in a random order. The mixer also determines a set of suppliers from which the demander will select its counterpart. Once the demander and the set of supplier have been determined, the mixer sends an AgentArrivalEvent to the demander, which triggers a procedure out of which the supplier is being determined (usually via the usage of strategies). Once the supplier has been selected, the demander asks to the MarketSimulation object to commit the transaction. The market simulation uses the transaction mechanism to execute the transaction. It is the transaction mechanism which will update accordingly the stock matrix and thus ensures the stock-flow and flow-flow consistency of the simulation.

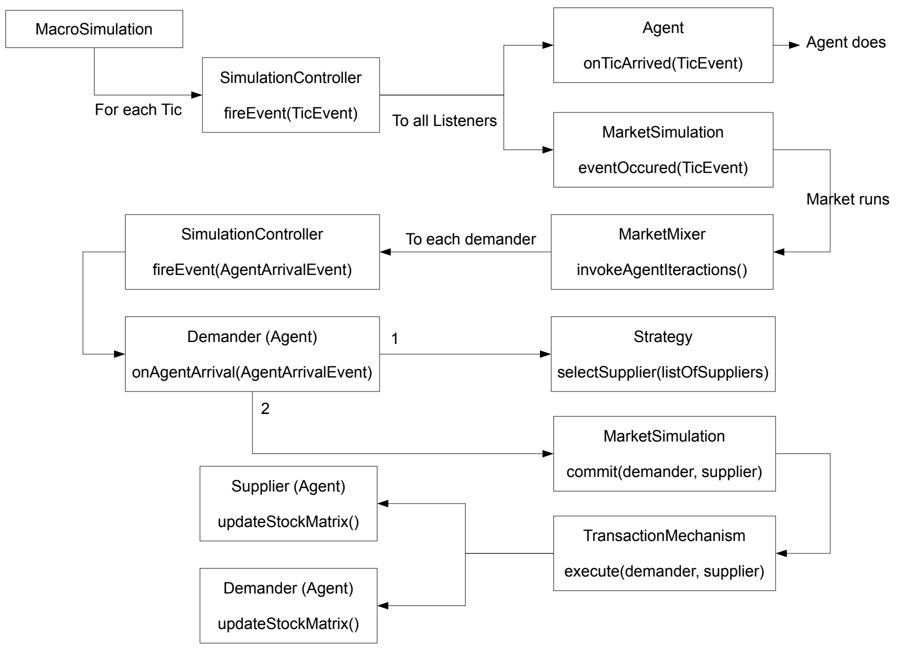


Figure 2 Tic path

3 JMAB folder description

JMAB contains the following source folders, each following the name convention: jmab.xxx. Each folder is discussed in the following sub-sections.

- agents
- distribution
- events
- expectations
- stockmatrix
- init
- mechanism

- mixing
- population
- report
- simulations
- strategies

3.1 Simulations

The folder Simulations contains the classes and interfaces that manage the overall simulation procedure. These classes extend the original simulation classes from JABM contained in the root folder (net.sourceforge.jabm). The simulation classes can be divided into two broad categories: those that deal with the management of the macro-simulation and those that deal with the management of a single market simulation (within a macro-simulation).

The MacroSimulation object contains a list of TicEvents, which will be send to the agents or markets via the simulation controller (see the Event folder description). These tics are defined within the MacroSimulation class because they are specific to each implementation of a model. The MacroSimulation object also contains a list of MarketSimulation objects, which will be used to simulate each specific market (consumption good market, deposit market, credit market, and so on) and an object of the type MacroPopulation which represent the set of all the existing agents, grouped into populations (see the Population folder description).

Each MarketSimulation object contains 3 objects: a mixer, a transaction mechanism and a market population. The market population (see the Population folder description) contains references to the populations containing the potential demanders and suppliers (i.e. households and firms for the consumption good market, firms and banks for the credit market). The mixer (see the Mixing folder description) is the mechanism specifying how demanders and suppliers will be interacting: how demanders are selected and in which order, and with whom will they interact (i.e. the whole supplier population or only a subset). Furthermore the mixer determines when the market is closed (i.e. when there are no more demanders or suppliers). Finally, the transaction mechanism (see the Mechanisms folder description) realizes the transaction, once a demander and a supplier have been selected. This consists in determining the price and quantity of the transaction (depending on various factors such as availability of funds or supply quantities, amount demanded, etc.) and then updating the stock matrix of all the agents involved in the transaction (these can be the two active agents but also any other passive agents whose balance sheets are impacted due to clearing mechanisms).

3.2 Events

The Events folder contains the classes and interfaces related to the messages being sent from the macro-simulations to all the listeners (agents, market simulation and reporters objects). The events can be divided into 4 categories: events related to tic triggering actions from the agents objects , events related to tic triggering actions for the market simulation objects, events related to the triggering reports and specific events such as dying agents events or serialization events (allowing to save the simulation at a specific instant).

MacroTicEvent are general events that will trigger agents. Each agent listening to the SimulationController will compare the value of the tic with its own list and determine whether it needs to do something (i.e. update its expectation, determine its credit demand or the interest rate it will charge). These tic do not lead to agent interactions (except for interest or tax payments but these are actions based on previous interactions).

The MarketTicEvent will not trigger agents but MarketSimulation objects. As for agents, each market compares the value of the tic with its own list of values and determines whether it needs to do something or not. One the market has been activated; it asks the mixer to start the interactions. The order in which parties in the market are activated depends on each mixer implementation. We use various versions of the market mixer (see the Mixing folder description) that activate demanders in a random order. The mixer also determines a set of suppliers from which the demander will select its counterpart. Once the demander and the set of supplier have been determined, the mixer sends an AgentArrivalEvent to the demander, which triggers a procedure out of which the supplier is being determined (usually via the usage of strategies, see the Strategies folder description). Once the supplier has been selected, the demander asks to the MarketSimulation object to commit the transaction. The market simulation uses the transaction mechanism to execute the transaction. It is the transaction mechanism which will update accordingly the stock matrix and thus ensures the stock-flow and flow-flow consistency of the simulation.

MacroVariableTicEvent and MicroMultipleVariableTicEvent are event related to reports, and will be described at length in the Report folder description.

3.3 Population

Population contains the classes that extend the JABM population class (net.sourceforge.jabm.population). The reason being that JMAB and all the derived models will contain more than one type of population. The MacroPopulation class contains a list of Population objects, each of them containing a list of agents (see the Agent folder description). The MarketPopulation object allows storing the references to two Population objects, a list of buyers and a list of sellers respectively.

The population folder contains also a set of interface and abstract classes related to the handling of entry and exit of agents (so far the models have been keeping constant populations).

3.4 Mixing

Mixers are objects used by the MarketSimulation objects, in the course of the interaction on a specific market. These mixers specify the way in which agents interacts, that is who meets whom and in which order. The specific way in which these interactions are implemented depends on the class being used. As explained previously, all mixers activate demanders in a random order. They also determine a (sub-) set of suppliers from which the demander will select its counterpart. These classes are then divided in two broad categories: one step markets and two-step markets.

All one-step market mixers classes implement the MarketMixer interface which specifies two methods: a logical method “closed” determining whether all the interactions have been ran and a method “invokeAgentInteractions” which triggers interactions between suppliers and demanders on the market. In these markets, the demander selects its preferred suppliers from the subset offered by the mixer and then realises a transaction immediately. The different classes implementing this type of markets differ in the size of the set of suppliers with whom the demander interacts (entire supplier population - RandomRobinBuyerMixer- vs. a selection of them - SelectionRandomRobinBuyerMixer -). Furthermore, the Dynamic versions of the classes allow for the population of buyers and sellers to be populated during each period (for example when an agent can be a supplier or a demander, depending on its status – housing market or interbank market).

All two-step market mixers classes implement the TwoStepMarketMixer interface which specifies three methods: the logical “closed” method and two interactions methods: firstInteraction and secondInteraction. These markets are those were there is a first interaction (e.g. suppliers first send a brochure of the goods they produce to some demanders) and then transaction occurs at a later stage of the period (after the buyers has obtained a credit, for example). The classes implementing these two-steps markets are the following: AbstractTwoStepMarketMixer, BrochureRandomRobinBuyerMixer, BrochureSelectionRandomRobinBuyerMixer. The two non-abstract classes differ in determining the number of suppliers with whom the buyer will interact. The BrochureSelectionRandomRobinBuyerMixer constrain the number of suppliers to the nbSellers argument.

3.5 Mechanism

The Mechanism folder contains the interfaces and classes that deal with transaction mechanisms. These are essential to ensure the stock-flow consistency at the micro level. The transaction mechanism realizes the transaction, once a demander and a supplier have been selected, via the mixing process. A transaction consists in determining a price and a quantity of the selected good (depending on various factors such as availability of funds or supply quantities, amount demanded, etc.) and then updating the stock matrix of all the agents involved in the transaction (these can be the two active agents but also any other passive agents whose balance sheets are impacted due to clearing mechanisms).

Three interfaces structure the various classes implementing these transaction mechanisms: the Mechanism interface declaring the methods common to all mechanisms, the CreditMechanism interface which extends Mechanism and declares the methods related to credit creation, and the GoodMechanism, also an extension of Mechanism, which declares the methods related to goods (consumption or capital) transactions. While this distinction between good and credit mechanism is essential, there are more than two types of mechansisms. We will rather distinguish them between exchange transaction (those involving an exchange of an item –real good or financial asset- for money –either cash or via a deposit transfer-), labor transaction (related to hiring of employees), credit creation, and reallocation transfers (i.e. changing deposits from one bank to another).

3.5.1 Exchange transaction mechanisms

Exchange transaction mechanisms are the simplest form of transaction that can occur in JMAB. It involves the exchange of an item (real good or financial asset) for money (cash or deposit transfer). For now, only the AtOnceMechanism has been implemented for real goods (consumption or capital). This represent the case of a demander buying all the goods he needs from the selected supplier, provided the supplier has enough stock and provided the supplier has enough liquidity. The mechanism thus computes the minimum quantity that can be traded (i.e. the minimum between quantity demanded, quantity supplied and available liquidity divided by price). Once this has been set, the mechanism updates the stock matrix objects (see the Agents and StockMatrix folder descriptions). This is done by changing the asset holder of the traded goods, adding them in the buyer stock matrix and removing them from the seller's one. The amount of the sale is then removed from the paying stock (deposit account, cash) of the buyer and added to the receiving stock of the supplier. If a clearing mechanism is needed (i.e. in the case of deposit transfers), it is ensured as well.

The BondMechanism implements the exchange transaction of bonds in exchange for a deposit transfer. The mechanism is very similar to the AtOnceMechanism: it checks for the minimal quantity of bonds that can be exchanged based on demand, supply and liquidity availability. Once the quantity has been determined, bonds are exchanged (i.e. added to the asset side of the buyer and to the liability side of the seller) and deposits are transferred. Clearing is ensured by the mechanism, as always.

3.5.2 Credit mechanisms

Credit mechanisms are a bit different in the way that they create new items in the stock matrix rather than re-shuffling items between buyers and sellers. There are two version of credit mechanisms implemented for now: ConstrainedCreditMechanism which deals with the case of credit constraints and the ReservesMechanism which deals with advances. Both credit mechanisms are done in the following way: the amount lent is equal to the minimum between the demand for credit and the supply of credit (can be constrained at the demander-level, i.e. constraint for a specific borrower, or at the supplier-level, i.e. constraint for a specific lender). Once this minimum amount has been determined, the transaction mechanism creates a loan object in both the borrower and lender's balance sheet (in the liability and asset side, respectively). The characteristics of the loan (maturity, amortization scheme, interest rate) re determined either by the demander or the supplier, depending on the case. If the borrower already has a deposit account with the lender, the corresponding amount is added in that deposit account. Otherwise, a deposit account is created and added in both balance sheets (again in the asset and liability side, respectively).

3.5.3 Reallocation mechanisms

Reallocation mechanisms (DepositMechanism): this mechanism deals with the re-allocation of deposits from one bank to another, under the constraint of a certain amount of money to be held as cash. In this specific example, the mechanism controls whether the agent demanding a deposit account already has a deposit in the bank. If such a deposit account does not exist then it creates it. The money held in all the other deposit accounts are then transferred to the deposit account. The mechanism deals with the clearing process as well.

3.5.4 Labour mechanisms

There are two labour mechanisms implemented in JMAB: an unconstrained version (UnconstrainedLaborMechanism) and a constrained one (LaborMechanism). They differ in that in the constrained version, the mechanisms checks that the employer has enough liquidity to pay for the worker to be hired. If this is the case, the worker is hired and wages are paid. The unconstrained labour mechanism does not check for liquidity and the worker is hired directly. No wages are paid in this case (and the modeller should make sure that wages are paid later on in the simulation).

3.6 Agents

The Agent folder contains all the classes and interfaces related to the implementation of agents. The contents of this folder can be divided into two broad categories: the implementation of the simplest form of agents and the definition of roles that agent might have to fill. The first category contains the interface and abstract classes that will be at the basis of any specific implementation (i.e. what is shared by all agents may they be governments, households or corporations). The second category consists in interfaces that define the methods that have to be implemented by agents in order to be able to fill a specific role (credit demander, good supplier or tax payer for example).

3.6.1 Simple agent

Each agent is of a class extending the SimpleAbstractAgent class. This mother class contains a set of variables common to all agents: the StockMatrix (see hereunder and the StockMatrix folder description), a map containing strategies (see the Strategies folder description), a map containing lagged values, and a map of expectations (see the Expectation folder description). These three maps contain objects specific to each agent’s type (for example, banks may have strategies for interest settings, and credit supply, passed values about non performing loans and profits and expectations about deposits while households have strategies to select consumption good suppliers and wage setting, passed values about consumption prices and employment and expectations about consumption prices).

The stock matrix is an object containing references to all the assets (real or financial) and liabilities (only financial) that each agent owns. It is an abstraction of the concept of balance sheet, extended to include assets that are usually not accounted for, such as consumable goods. The stock matrix consists of two lists (assets and liabilities) of lists of Items (see the StockMatrix folder description). Each Item contains a reference to its asset holder and its liability holder. For real goods (such as consumption or capital goods) the liability holder refers to the producer. This specific implementation de facto creates a large network of stock matrices connected to each other via the Items and their references to asset and liability holders. The stock matrix and its items will be detailed in the StockMatrix folder description.

JMAB also provides three abstract classes related to the generic roles of Households, Firms and Banks. These classes contain methods that are likely to be used for any household, firm or bank, such as employer/employee methods or clearing methods for banks.

3.6.2 Roles

The specific agents will have numerous roles in a model. For example a household might be a good demander and a deposit holder, and a tax payer. All these different roles require specific strategies (for example a good supplier selection strategy, a bank selection strategy and a tax paying strategy, see the Strategies folder description). In order to remain fully flexible with future specific implementations of strategies and agents, we have decided to create interfaces that specify the name of the methods that have to be implemented. In this way, it is possible to use the predefined strategies used in JMAB with specific implementation of agents. All these interfaces extend the MacroAgent interface that describes all the publicly available methods that any JMAB agent has to implement. Most of the MacroAgent methods are implemented by the SimpleAbstractAgent class. The specific interfaces are the following:

- BaselIIIAgent: role of agents that respect Basel III requirements.
- BondDemander: role of agents that buy bonds.
- BondSupplier: role of agents that emit bonds.
- CreditDemander: role of agents that demand credit (i.e. borrower).
- CreditSupplier: role of agents that supply credit (i.e. lender).
- DepositDemander: role of agents that hold deposit accounts.
- DepositSupplier: (*extends LiabilitySupplier*) role of agents that supply deposit accounts.
- FinanceAgent: role of agents that have specific financing strategies to finance investment.
- GoodDemander: role of agents that buy goods.
- GoodSupplier: role of agents that sell goods.
- IncomeTaxPayer: (*extension of TaxPayer*) role of taxpayer agents that have to pay taxes on income.
- InterestRateSetterWithTargets: role of agents that sets interest rates, depending on some targets.
- InvestmentAgent: role of agents that can invest in capital goods.
- LaborDemander: role of agents that can hire workers (i.e. employers).
- LaborSupplier: role of agents that can be employed (i.e. employees).
- LiabilitySupplier: role of agents that supply banking accounts (i.e. reserves, deposits, saving accounts). This interface specifies the clearing mechanisms that have to be implemented.
- PriceSetterWithTargets: role of agents that sets prices of goods, depending on some targets.
- ProfitsTaxPayer: (*extension of TaxPayer*) role of taxpayer agents that have to pay taxes on profits.
- TaxPayer: role of all agents that have to pay taxes on a specific account.

- WageSetterWithTargets: role of a worker that changes its wage depending on some targets.

3.7 StockMatrix

The StockMatrix folder contains all the classes and interfaces related to the management of the Stock Matrix. As already stated, the stock matrix is an abstraction of the concept of balance sheet, extended to include assets that are usually not accounted for, such as consumable goods. This specific implementation de facto creates a large network of stock matrices connected to each other via the Items they contain, and the references to asset and liability holders.

The stock matrix consists of two lists (assets and liabilities) of lists of objects. All the objects contained in the stock matrix implement the interface Item and extend the abstract class AbstractItem. The Item and AsbtractItem refer to characteristics common to all the objects contained in a stock matrix: a value, a quantity, an age, a reference to its position in the stock matrix (i.e. in which list of the assets or liabilities it is contained), and references to its asset holder and its liability holder. The items can be divided in two broad categories: real goods (consumption or capital) and financial assets.

3.7.1 Real goods

Three classes implement real goods: AbstractGood containing methods common to both consumption and capital goods and the specific implementation of ConsumptionGood and CapitalGood. For real goods the liability holder refers to the producer but the good is not recorded in the liability side of the producer (i.e. real goods are not financial assets and are thus only an asset for someone). The methods common to all goods refer to the ageing of goods, the producer reference and the pricing of the good (unit cost of production and price). Consumption goods differ from capital good in that their only attribute is their duration (i.e. the length it is held in the stock matrix before being completely consumed and removed from the stock matrix). Capital goods on the other hand have a productivity, a capital-labor ratio, an amortization period and a duration period, and an obsolete status. The difference between amortization and duration lies in the fact that the first one is an accounting concept while the second one refers to the real life cycle of the good. If the good has come to the end of its life cycle or is considered as obsolete, it is removed from the stock matrix.

3.7.2 Financial Assets

The financial assets implemented in JMAB are the following. On top of the AbstractItem characteristics, they contain:

- Cash: no specific characteristics
- Deposit: interest rate
- Bonds: interest rate, price and maturity
- Equity: price
- Security: price
- Loan: interest rate, initial amount, maturity, amortization type (constant rate, constant capital payment, only interest payment).

3.8 Expectations

The Expectations folder contains classes related to expectation and memories (or passed values).

3.8.1 Passed Values

Passed values are implemented by an interface (PassedValues) and a class (TreeMapPassedValues). The concept of passed value is simple: it consists in a structure storing passed values and allowing to saved them and access them, based on the period in which they were collected. The only specific implementation uses a TreeMap structure.

3.8.2 Expectations

Methods that any type of expectation has to implement are defined in the Expectation interface. These consist in: (i) updating the passed values that could be used to compute a new expectation, (ii) computing the expectation value, and (iii) obtaining the computed expectation. JMAB has different implementations of expectations:

- AnchoringAndAdjustmentExpectation: Anchoring and Adjustment Expectation, see Assenza et al. (2013)
- AugmentedAdaptiveExpectation:
- SimpleAdaptiveExpectation: $X^e = \varphi X^e_{-1} - (1-\varphi)X_{-1}$
- SimpleWeigthedExpectation: $X^e_t = \sum_{i=1}^{n} w_i X_{t-i}$
- TrendFollowingExpectation: Trend following expectation, see Assenza et al. (2013)

3.9 Strategies

Because agents in JMAB can have more than one strategy, we had to extend the strategy structure of JMAB, this is the purpose of the MacroStrategy interface and the MacroMultipleStrategy class. MacroStrategy extends the JABM strategy interface (net.sourceforge.strategy.Strategy) and defines the method to access a single strategy. MacroMultipleStrategy extend the JABM abstract strategy class (net.sourceforge.jabm.strategy.AbstractStrategy) and contains a structure (HashMap) allowing to access a specific strategy with a key (integer). It implements the MacroStrategy interface. All the interfaces extend the SingleStrategy interface.

3.9.1 Specific strategies

These are the specific strategies with interface and classes. All the classes extend the JABM abstract strategy class (net.sourceforge.jabm.strategy.AbstractStrategy) and implement the specific interfaces. These specific strategies are often related to the specific roles that agents might have to implement (see the Agent folder description).

- BankruptcyStrategy: Bankruptcy strategy.
- BondDemandStrategy: Bond demand strategy.
- ConsumptionStrategy: Consumption Strategy.
    - ConsumptionFixedPropensitiesOOIW: Consumption level depends on expected income and past wealth.

- ConsumptionFixedPropensitiesOOIWWithPersistency Consumption level is the weighted average between past consumption and a function of expected income and past wealth.
- DividendsStrategy: Dividend distribution strategy.
  - FixedShareOfProfitsToPopulationAsShareOfWealthDividends: dividends (a fixed share of profits) are distributed to households based on the agent's net worth over total households net worth.
- FinanceStrategy: Strategy computing the credit demand for a corporation (firms and banks).
  - BaselIIIReserveRequirements: credit demand for banks depends on the Capital Adequacy Ratio
  - BaselIIIReserveRequirementsBLR: credit demand for banks depends on the liquidity ratio
  - DynamicTradeOffFinance: credit demand depends on the targeted leverage of the firm (endogenous, depends on return rate on investment and reference variable), its actual level of debt and the expected level of credit needed to finance investment.
  - DynamicTradeOffFinance2_0: credit demand depends on the targeted leverage of the firm (endogenous, depends on profit rate on sales), its actual level of debt and the expected level of credit needed to finance investment.
  - PeckingOrderTheory: credit demand is such that first all internal source of finance for investment are used before turning to external credit.
- InterestRateStrategy: Interest rate setting strategy
  - AdaptiveInterestRate: interest rate is increased or decreased by a random amount depending whether a reference indicator is above or below its target
- InvestmentStrategy: Investment strategy (desired growth of production capacity)
  - InvestmentCapacityOperatingCashFlow: desired growth rate is a function of operating cash flow and capacity utilisation.
  - InvestmentCapacityProfits: desired growth rate is a function of profit rate and capacity utilisation.
  - InvestmentProfitsDebtBurden: desired growth rate is a function of profit rate and debt burden (interest payments over profits).
- PricingStrategy:Pricing strategy
  - AdaptiveMarkUpOnAC: price is a mark-up where mark-up is endogenously updated by a random value, depending on whether a target has been met (for example the sales to inventories ratio).
  - AdaptivePriceOnAC: price increase or decrease from a random value depending on whether a target has been met (for example the sales to inventories ratio).
  - RandomPrice: price randomly extracted from a distribution (see Distribution folder description).
- ProductionStrategy: Strategy determining the level of output
  - OutputOnDemand: output is produced on demand

    - TargetExpectedInventoriesOutputStrategy: output is equal to expected sales plus the quantity (could be negative) needed to restore the level of inventories (equal to a share of expected sales).
- RandDInvestment: Investment in R&D Strategy
    - RandDInvestmentSharePastSales: Investment in R&D is equal to a share of past sales.
- RandDOutcome: Strategy determining the outcome of an investment in R&D
    - RandDOutcomeBetaDistribution: Productivity gain is $prod_{gain} = 0.2 * gain * P$ where $gain \sim Beta(\alpha, \beta)$, $P \sim Binomial(1 - e^{-\rho * nb_{res}})$, α, β, ρ are parameters and nb_{res} is the number of researchers hired by the innovative firm.
- RealCapitalDemandStrategy: Strategy that computes the real demand of capital goods, based on the supplier.
    - RealCapitalDemandSimple: real demand is equal to the quantity of goods necessary to attain the desired level of potential output.
    - RealCapitalDemandWithPayBackPeriod: real demand is equal to the quantity of goods necessary to obtain the desired level of potential output. All the existing capital goods for which the unit costs of production over the life of the capital good (lifetime unit cost) is higher than the lifetime unit cost with the new capital good will be considered as obsolete and discarded, hence increasing the real demand for capital goods.
- SelectDepositSupplierStrategy: Deposit supplier selection strategy
    - MostPayingDeposit and MostPayingDepositWithSwitching: select deposit supplier with highest interest rate, with or without probability of switching based on interest rate differential.
- SelectLenderStrategy: Lender selection strategy
    - CheapestLender and CheapestLenderWithSwitching: select lender with lowest interest rate, with or without probability of switching based on interest rate differential.
- SelectSellerStrategy: Seller selection strategy
    - BestQualityPriceCapitalSupplier and BestQualityPriceCapitalSupplierWithSwitching: for capital goods: selects the good offering the best lifetime unit cost, with or without probability of switching based on lifetime unit cost differential.
    - CheapestGoodSupplier and CheapestGoodSupplierWithSwitching: for any good: selects the cheapest good, with or without probability of switching based on price differential.
- SelectWorkerStrategy: Worker selection strategy
    - SelectCheapestWorkerStrategy: Cheapest worker within the subset (note that class can also send a wage-ordered list)
    - SelectRandomWorkerStrategy: Random worker from the subset (note that class can also send a randomised list)
- SpecificCreditSupplyStrategy: Specific (to the demander) credit supply strategy
    - ExpectedReturnCreditSupply: credit supply is such that the expected return of the loan (based on credit operational cash flow and banks risk aversion) is above zero.

        - DefaultProbilityComputer: interface describing the method to be implemented to compute the probability of default of a specific borrower.
        - DeterministicLogisticLeverageDefaultProbabilityComputer: probability of default is $P = 1 - e^{\alpha*(lev-\beta)}$ where α, β are parameters and *lev* is the leverage of the borrower.
    - MaxExposureStrategy: credit supply is such that the demander's total credit with the supplier is below a certain share of the supplier's loan portfolio.
    - MaxLossCreditStrategy: credit supply is such that in the case of default, the loss (see hereunder for the specific computations) is below a certain share of the bank's net wealth.
        - ShareOfColateralLossComputer: interface describing the method to be implemented to compute the loss function in the case of non performing loans.
        - FixedShareOfColateralLossComputer: the loss is a constant share of the outstanding amount of credit.
- SupplyCreditStrategy: Aggregate credit supply strategy.
    - InfiniteSupplyCreditStrategy: infinite supply.
    - SupplyCreditAdaptiveCARTarget: credit supply depends on past credit and on whether the actual capital adequacy ratio (CAR) is above or below the targeted one (change over time depending on non-performing loan ratios).
    - SupplyCreditBaselIII: credit supply depends on the difference between actual CAR and targeted one.
    - SupplyCreditBaselIIIEndogenousTarget: credit supply depends on the difference between actual CAR and targeted one (change over time depending on non-performing loan ratios).
- SwitchingStrategy: Strategy determining whether the agent switches by comparing two double values.
    - AbstractSwitchingStrategy: Abstract implementation of switching strategy, depends on the probability distribution
    - SwitchingStrategySimple: probability: $P = 1 - e^{\lambda^{\frac{p_1 - p_2}{p_1}}}$ where λ is a parameter and p_1 and p_2 are the two values to compare.
- TaxPayerStrategy: Strategy of tax payers
    - IncomeWealthTaxStrategy: tax paid depends on income and wealth
    - ProfitsWealthTaxStrategy: tax paid depends on profits and wealth
- UpdateInventoriesProductivityStrategy: Strategy used to update the productivity level of an inventory stock.
    - UpdateInventoriesProductivityWithCost: the update costs a share of the stock (in real terms), which is lost.
- WageStrategy: Strategy used to determine the wages of an worker.
    - AdaptiveWageStrategy: wage depends on macro and micro targets, wage increase by a random amount if micro and macro thresholds are respected, wage decrease by a random amount if micro threshold is violated.

3.10 Init

The Init folder contains the classes and interfaces that are used to initialize the objects composing a model, one they have been created via the SpringFramework. There are two interfaces one used to define the methods needed to initialise all the agents of the simulation (MacroAgentInitialiser) and one defining the methods needed to initialise the markets' networks (MarketAgentInitializer). The agent initialiser interface has to be implemented for each specific model, since it needs to know what agents are going to be created. Nonetheless JMAB contains an abstract class that deals with all the necessary methods but the initialise method and a SerializationInitialiser that allows to start a simulation from a special file containing all the characteristics of a model saved in the past. This allows stopping a model at a point, which is deemed interesting by the modeller, and re-starting it at a later stage[1]. In order to be able to use the serialisation, all classes in the model need to implement the following two methods: populateFromBytes and getBytes.

The network initialiser interface allows creating an initial network for all the markets that require one. This is typically the case when buyers or sellers are selected with a strategy allowing for a switching probability. In order for the first period to be simulated correctly, there need to be an original buyer or seller. The random market initialiser (RandomMarketInitialiser) allows to randomly allocate buyers and sellers.

3.11 Report

The Report folder contains all the classes and interfaces needed to manage and generate report files, i.e. output files containing the relevant simulated data. There are two main types of report: macro report and micro report. The MacroVariableComputer interface defines the methods used to compute single macro indicator reports (e.g. Nominal GDP, inflation or unemployment rate, etc.). The MicroMultipleVariablesComputer interface defines the methods used to compute multiple micro-indicators reports (real consumption for each household, interests paid for each firm, etc.). One special class lies outside these two categories the BalanceSheetNetworkComputer that will be treated in detail hereunder.

In general any report (may it be macro or micro) follows the following workflow (defined in JABM):

1. An event of a special type (either MacroSimEvent or MicroSimEvent) event is being filled (variable name, variable value and time) and sent by the MacroSimulation, using a computer of type either MacroVariableComputer or MicroMultipleVariablesComputer to compute the value of the indicator(s).

[1] Note that this method is highly dependent on a specific implementation of a model. Any change in the class or structure of the model will render the saved file incompatible with the new model. This method is still in beta version and needs to be tested correctly.

2. The event is “listened” by an instance of MacroSimEventReport or MicroSimEventReport and triggers the onEventPrototype method which triggers the compute of the attribute reportVariables.
3. This reportVariables attribute is an instance of the class CSVReportVariables. Within the compute method, the method compute of the attribute reportVariables is invoked.
4. This reportVariables attribute is an instance of the (JMAB) class MacroVariableReport or MicroMultipleVariablesReport. The compute method triggers the eventOccured method which updates its value from those contained in the event. Thus the reportVariables attribute of the reportVariables attribute of the MacroSimEventReport (or MicroSimEventReport) instance is being updated by the MacroSimEvent (or the MicroSimEvent).

This is totally generic and any variable or set of variables can thus be computed this way. The only specific class that has to be implemented is the computer. All the rest is being managed by JMAB or JABM. The configuration of the reports is done in the SpringFramework xml configuration file.

The reports existing in JMAB are not listed here (there are more than 50 of them). The user is invited to go and look at the code when the name is not self-explanatory.

3.12 Distribution

The distribution folder contains the classes defining specific distribution probabilities, to be used when random extractions are used. JMAB offers two classes on top of those offered by JABM: a folded normal distribution and an exponential distribution.

4 Benchark folder description

Antoine Godin 18/2/2016 21:24
Comment [3]: TODO

- Plus root for benchmark (and any other model)

5 References

Assenza, T., Heemeijer, P., Hommes, C., and Massaro, C. (2013). Individual expectations and aggregate macro behavior. *Tinbergen Institute Discussion Papers, 16*.

Caiani, Alessandro, et al. "Agent Based-Stock Flow Consistent Macroeconomics: Towards a Benchmark Model." *Available at SSRN* (2015).