# ABMland -
# A generic framework for collaborative agent-based model development

Status of documentation: August 2011

This document describes the ABMland framework as of August 2011 (version 1.0.2).
It can be downloaded under http://www.ufz.de/abmland.

Technical questions? → abmland[at]ufz.de

Authors: Daniel Kahlenberg, daniel.kahlenberg[at]ufz.de;
Nina Schwarz, nina.schwarz[at]ufz.de

Helmholtz Centre for Environmental Research – UFZ
Department Computational Landscape Ecology
Permoserstrasse 15, 04318 Leipzig, Germany

This work was partly funded by the PLUREL Integrated Project (Peri-urban Land Use Relationships) of the European Commission, Directorate-General for Research, under the 6th Framework Programme (project reference: 36921).

# Contents

# 1. Introduction

This documentation presents a framework for collaboratively developing agent-based models within a spatially explicit and joint environment. The proposed design concept covers three main issues: common scheduling, explicit data exchange, and full functionality even if agents are only partially implemented.

The ABMland framework is conceptualised for urban land use change. However, it can be adapted to a variety of modelling topics. Compared to existing tools, ABMland allows for collaborative implementation of agent-based models and parallel model development while simplifying the coding process. This concept and the implementation of the framework thus support the structured development of complex agent-based model in a collaborative environment.

The ABMland framework is implemented in Java building upon Repast Simphony (Repast S in the remainder of the document) and other libraries in form of regular jar files. It builds upon JPF (Java Plugin Framework) plugins and can easily be used with prepared Maven launch files for code handling in the developer environment Eclipse.

# 2. Conceptual approach

In the current version, six ABMs are included to represent the most important interactions regarding urban land use change: residents, businesses, planners, developers, infrastructure providers, and lobbyists.

## *2.1 Interactions*

The following figure depicts the interactions between agents as implemented in the current ABMland framework.

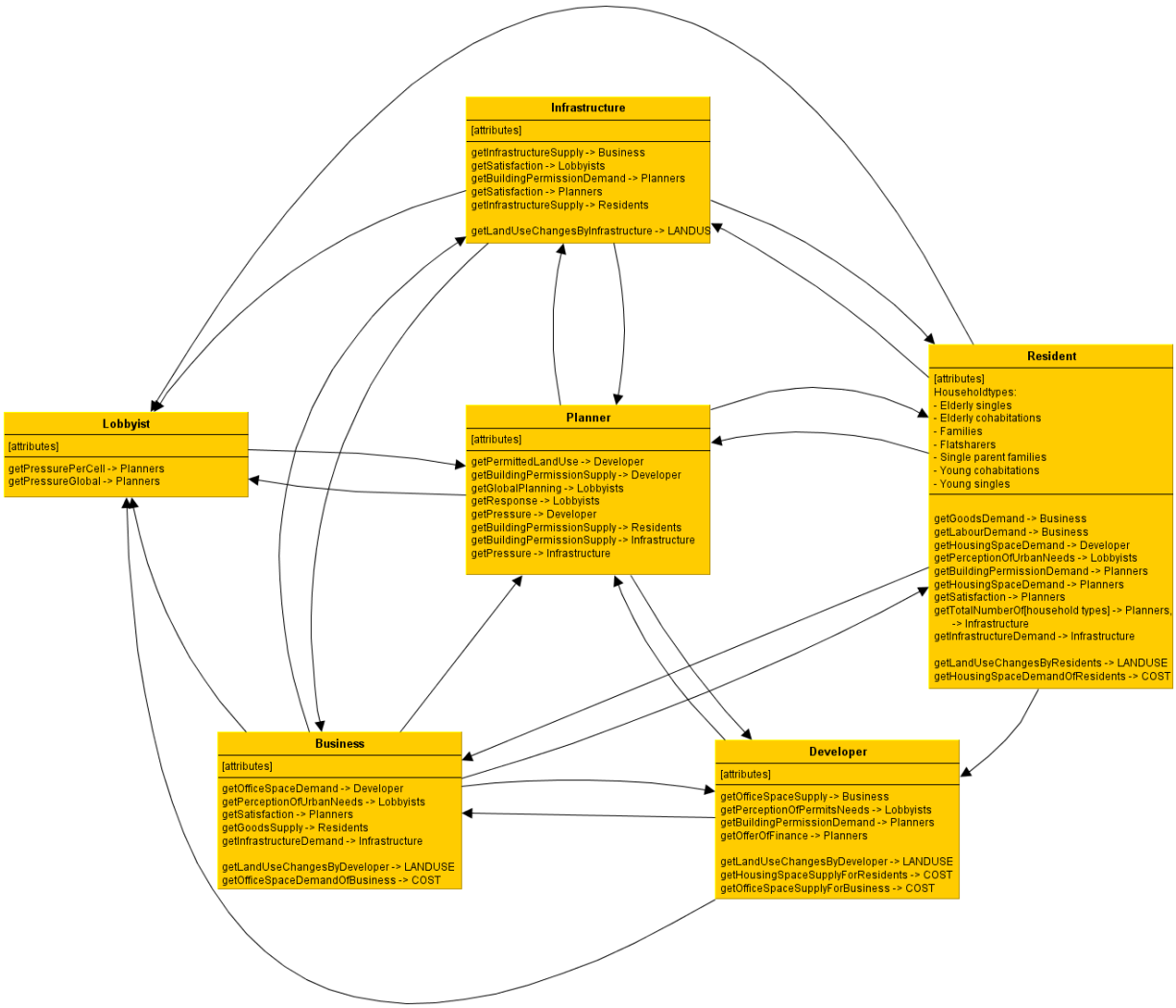


*Figure 1: Interactions of agents' sub-models in ABMland*

These interactions are summarised in the following Table. "Exporter" refers to the exporting model, "Importer" to the importing one. "Interface" is the name of the data exchange between the models. "Internal type" lists the set-up of ABMland-specific data types as well as Java or Repast S data types and structures. "Public type" provides the name of the data type which is used to summarise the internal type. See section on data types for details.

*Table 1: Implemented interactions of agents in ABMland. Note: AbstractAllCells<XXX> = Repast S value layer with XXX as value.*

| **Exporter** | **Importer** | **Interface** | **Internal type** | **Public type** |
|---|---|---|---|---|
| COST | Business | getLandPriceIndex | AbstractAllCells<IPriceIndex> | ILandPriceIndexAllCells |
| LANDUSE | Business | getCommercialSpaceAvailability | AbstractAllCells<ISpace> | IFloorSpaceAllCells |
| LANDUSE | Business | getLandUse | AbstractAllCells<CLCover> | ICLCoverAllCells |
| Planner | Business | getPermissionForProposals | Map<Map<GridPoint, CLCover>, Boolean> | IPermissionForProposalTable |
| Resident | Business | getTotalHouseholdNumber | Map<GridPoint, Map<Householdtypes, IhouseholdNumber>> | IHouseholdNumberPerTypeAllCells |
| LANDUSE | COST | getCommercialSpaceAvailability | AbstractAllCells<ISpace> | IFloorSpaceAllCells |
| LANDUSE | COST | getVacantAccomodation | AbstractAllCells <IHouseholdNumber> | IHouseholdNumberAllCells |
| Business | Developer | getCommercialSpaceDemand | AbstractAllCells<ISpace> | IFloorSpaceAllCells |
| COST | Developer | getLandPriceIndex | AbstractAllCells<IPriceIndex> | ILandPriceIndexAllCells |
| LANDUSE | Developer | getLandUse | AbstractAllCells<CLCover> | ICLCoverAllCells |
| Planner | Developer | getGlobalPressure | Map<T, Ipressure>, data T = Issuetypes \| CLCover | IPressureForXTable |
| Planner | Developer | getPermissionForProposals | Map<Map<GridPoint, CLCover>, Boolean> | IPermissionForProposalTable |
| Planner | Developer | getPermittedLandUse | Map<GridPoint, CLCover[]> | ICLCoversAllCells (PLURAL)!!! |
| Planner | Developer | getSpatialPressure | Map<GridPoint, Map<T, Ipressure>>, data T = CLCover | IPressureForCLCoverAllCells |
| Resident | Developer | getSatisfaction | Map<GridPoint, Map<E, Map<Is<uetypes, Isatisfaction>>>, data E = Businesstypes \| Householdtypes | ISatisfactionPerIssueOfAgentAllCells |
| Resident | Developer | getTotalHouseholdNumber | Map<GridPoint, Map<Householdtypes, IhouseholdNumber>> | IHouseholdNumberPerTypeAllCells |
| LANDUSE | Infra-structure | getLandUse | AbstractAllCells<CLCover> | ICLCoverAllCells |
| Planner | Infra-structure | getPermissionForProposals | Map<Map<GridPoint, CLCover>, Boolean> | IPermissionForProposalTable |
| Planner | Infra-structure | getSpatialPressure | Map<GridPoint, Map<T, Ipressure>>, data T = CLCover | IPressureForCLCoverAllCells |
| Resident | Infra-structure | getTotalHouseholdNumber | Map<GridPoint, Map<Householdtypes, IhouseholdNumber>> | IHouseholdNumberPerTypeAllCells |
| Business | LANDUSE | getCommercialSpaceOccupation | AbstractAllCells<ISpace> | IFloorSpaceAllCells |
| Business | LANDUSE | getLandUseChanges | Map<GridPoint, CLCover> | ICLCoverSomeCells |
| Developer | LANDUSE | getLandUseChanges | Map<GridPoint, CLCover> | ICLCoverSomeCells |
| Infra-structure | LANDUSE | getAvailabilityOfMainRoads | AbstractAllCells<Boolean> | IBooleanAllCells |
| Infra-structure | LANDUSE | getAvailabilityOfPublicTransport | AbstractAllCells<Boolean> | IBooleanAllCells |
| Infra-structure | LANDUSE | getLandUseChanges | Map<GridPoint, CLCover> | ICLCoverSomeCells |
| Infra-structure | LANDUSE | getSchoolSupply | AbstractAllCells<Boolean> | IBooleanAllCells |
| Resident | LANDUSE | getTotalHouseholdNumber | Map<GridPoint, Map<Householdtypes, IhouseholdNumber>> | IHouseholdNumberPerTypeAllCells |
| LANDUSE | Lobbyist | getLandUse | AbstractAllCells<CLCover> | ICLCoverAllCells |
| Planner | Lobbyist | getGlobalPressure | Map<T, Ipressure>, data T = Issuetypes \| CLCover | IPressureForXTable |
| Planner | Lobbyist | getPendingProposals | Set<Map<GridPoint, CLCover>> | IProposalsSet |
| Planner | Lobbyist | getSpatialPressure | Map<GridPoint, Map<T, Ipressure>>, data T = CLCover | IPressureForCLCoverAllCells |
| Resident | Lobbyist | getSatisfaction | Map<GridPoint, Map<E, Map<Is<uetypes, Isatisfaction>>>, data E = Businesstypes \| Householdtypes | ISatisfactionPerIssueOfAgentAllCells |
| Resident | Lobbyist | getTotalHouseholdNumber | Map<GridPoint, Map<Householdtypes, IhouseholdNumber>> | IHouseholdNumberPerTypeAllCells |
| Business | Planner | getCommercialSpaceDemand | AbstractAllCells<ISpace> | IFloorSpaceAllCells |
| Business | Planner | getProposalsNeedingPermission | Set<Map<GridPoint, CLCover>> | IProposalsSet |

| Exporter | Importer | Interface | Internal type | Public type |
|---|---|---|---|---|
| Business | Planner | getSatisfaction | Map<GridPoint, Map<E, Map<Is<uetypes, Isatisfaction>>>, data E = Businesstypes \| Householdtypes | ISatisfactionPerIssueOfAgentAllCells |
| Developer | Planner | getProposalsNeedingPermission | Set<Map<GridPoint, CLCover>> | IProposalsSet |
| Infra-structure | Planner | getProposalsNeedingPermission | Set<Map<GridPoint, CLCover>> | IProposalsSet |
| LANDUSE | Planner | getLandUse | AbstractAllCells<CLCover> | ICLCoverAllCells |
| Lobbyist | Planner | getGlobalPressure | Map<T, Ipressure>, data T = Issuetypes \| CLCover | IPressureForXTable |
| Lobbyist | Planner | getSpatialPressure | Map<GridPoint, Map<T, Ipressure>>, data T = CLCover | IPressureForCLCoverAllCells |
| Lobbyist | Planner | getSupportForPlan | Map<T, Ipressure>, data T = Map<GridPoint, CLCover> | IPressureForXTable |
| Resident | Planner | getSatisfaction | Map<GridPoint, Map<E, Map<Is<uetypes, Isatisfaction>>>, data E = Businesstypes \| Householdtypes | ISatisfactionPerIssueOfAgentAllCells |
| Resident | Planner | getTotalHouseholdNumber | Map<GridPoint, Map<Householdtypes, IhouseholdNumber>> | IHouseholdNumberPerTypeAllCells |
| Business | Resident | getBusinessPresence | Map<GridPoint, Businesstypes[]> | IBusinesstypesAllCells |
| COST | Resident | getLandPriceIndex | AbstractAllCells<IPriceIndex> | ILandPriceIndexAllCells |
| LANDUSE | Resident | getAvailabilityOfMainRoads | AbstractAllCells<Boolean> | IBooleanAllCells |
| LANDUSE | Resident | getAvailabilityOfPublicTransport | AbstractAllCells<Boolean> | IBooleanAllCells |
| LANDUSE | Resident | getLandUse | AbstractAllCells<CLCover> | ICLCoverAllCells |
| LANDUSE | Resident | getSchoolSupply | AbstractAllCells<Boolean> | IBooleanAllCells |
| LANDUSE | Resident | getVacantAccomodation | AbstractAllCells <IHouseholdNumber> | IHouseholdNumberAllCells |

The framework forces the respective sub-models of ABMland to implement the methods. E.g. the Lobbyist sub-model has to provide the two methods getPressurePerCellForPlannersByLobbyists and getPressureGlobalForPlannersByLobbyists, otherwise the framework will produce error messages and will not be compiled. At the beginning of the implementation process, programmers working on a single sub-model can insert these methods automatically into their models [by using the Eclipse command "add unimplemented methods" in their respective model class]. This procedure (1) helps programmers with their work as the expected results are clearly stated and (2) is necessary for the interaction.

## *2.2 Agents & models*

The ABMland framework distinguishes between agents and models. For each agent, Java classes for the respective agent and the model are needed (e.g. InfrastructureAgent.java and InfrastructureModel.java).

The main purpose of the model is to aggregate values of individual agents to export them to other models in ABMland, e.g. the infrastructure model aggregates values of different infrastructure agents. Therefore, other agents do not need exact information on how many infrastructure agents actually are implemented, but they simply rely on the infrastructure model to provide the information needed. Of course it is possible to ask the model to provide information for single agents, so no information is lost using the model – agent concept.

The main purpose of the agent is to represent persons, groups, or institutions and their decision processes. It is possible to implement different agent classes representing different types (therefore having different decision algorithms) or to simply implement one agent class with several agents that only differ in their attributes.

## *2.3 Time & space*

The actual framework of ABMland is independent of a concrete definition of space and time. However, programmers will have to specify a certain configuration for each different simulation run. Furthermore, a definition of space is necessary to make sure that interactions between agents as well as land uses have proper units and that agents are located correctly in space. The definition of space is realised using data for initialisation of ABMland.

With the current version, ABMland works on extended CORINE land cover classes provided by the European Environment Agency (http://www.eea.europa.eu/publications/COR0-landcover) as land uses. However, other land use classes can easily be included. The spatial and temporal resolution needs to be defined during initialisation. Discrete time steps (called "tick" in Repast S) are used.

# 3. The Java framework

The ABMland framework consists of three main parts: (1) ABMland core functions, (2) ABMland wrapper and (3) ABMland models. ABMland core and ABMland wrapper build the actual framework of ABMland.

(1) ABMland core encompasses all programming code which is necessary to couple several Repast S models in general. The core functions are provided by four different projects:

- common-abmland
  Commonly used framework classes which are specific for ABMland and the current simulation, but do not depend on Repast S.
- abmlandcore
  Commonly used framework classes which are very generic and can probably be used for another spatially explicit and coupled ABM.
- common-repasts
  Commonly used framework classes which (1) should not be changed and (2) directly depend on Repast S.
- "repasts-legacy"
  External Repast S libraries.

The code of the packages common-abmland and abmlandcore is packed into .jar-files and it is bundled with common-repasts. Common-repasts is assembled as a plugin for the framework that Repast S uses.

(2) ABMland wrapper mainly fires up the Repast S GUI scenario view with some configuration data files, where runtime parameters such as the number of integrated models, which models are integrated, can be changed. Accordingly, individual programmers need to adapt the code.

(3) ABMland models (individual sub-models) are stored in respective folders and jar-archives. The package abmland.mainctrl in common-repasts encompasses the Controllers for storing and updating information on land use and costs.

## *3.1 ABMland core*

The core of ABMland consists of various packages which include all Java classes necessary to provide the framework for including all agent-based models in ABMland. These classes are independent of the concrete implementations and could be used for any other coupled agent-based model as well.

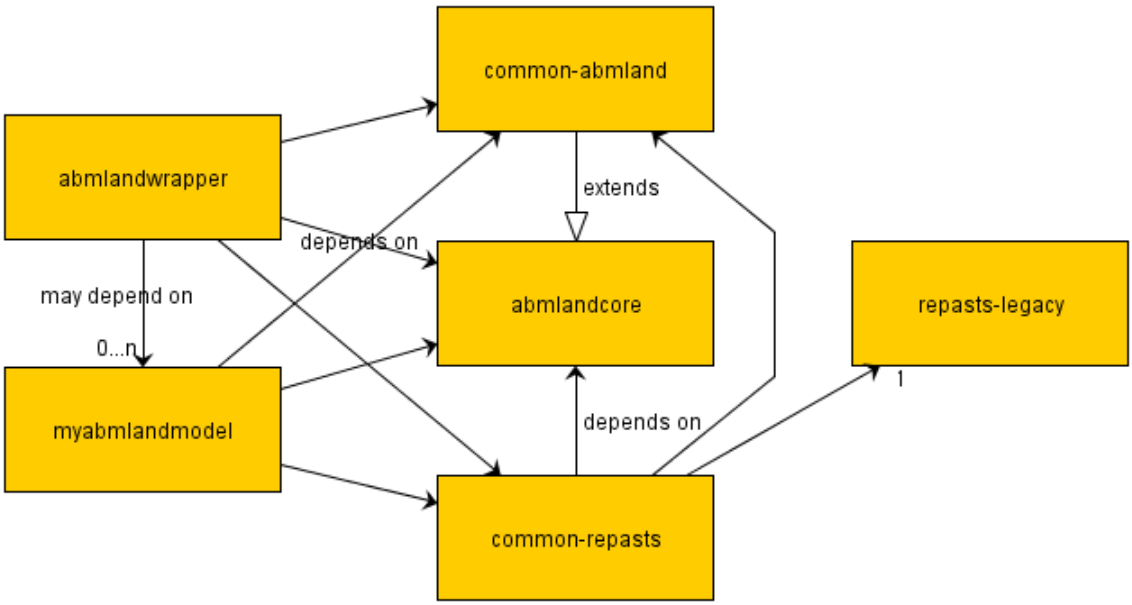


*Figure 2: Overview of ABMland core*

### 3.1.1 Relationship of agents and models

The general principle of ABMland is to enforce a certain, efficient chain of communication between agents: An agent of a certain implementation (e.g. infrastructure provider) asks its model for certain information, e.g. provided by residents. The model contacts a server which in turn asks the residents model, which asks its resident agents. This chain of communication has two major advantages: (1) The agent needing the information does not need to know the specific "name" or implementation of the other agents, and it does not even need to know if these agents are in fact part of the simulation or not. This feature is necessary if one wants to test a single model and not all models of ABMland. Therefore, one does not have to have Java code when testing a single model versus all models of ABMland. (2) If an agent needs aggregated information for several agents (e.g. several household types) it does not need to ask all resident agents and sum up the aggregated value on its own, but this agent lets the respective partner aggregate the values.

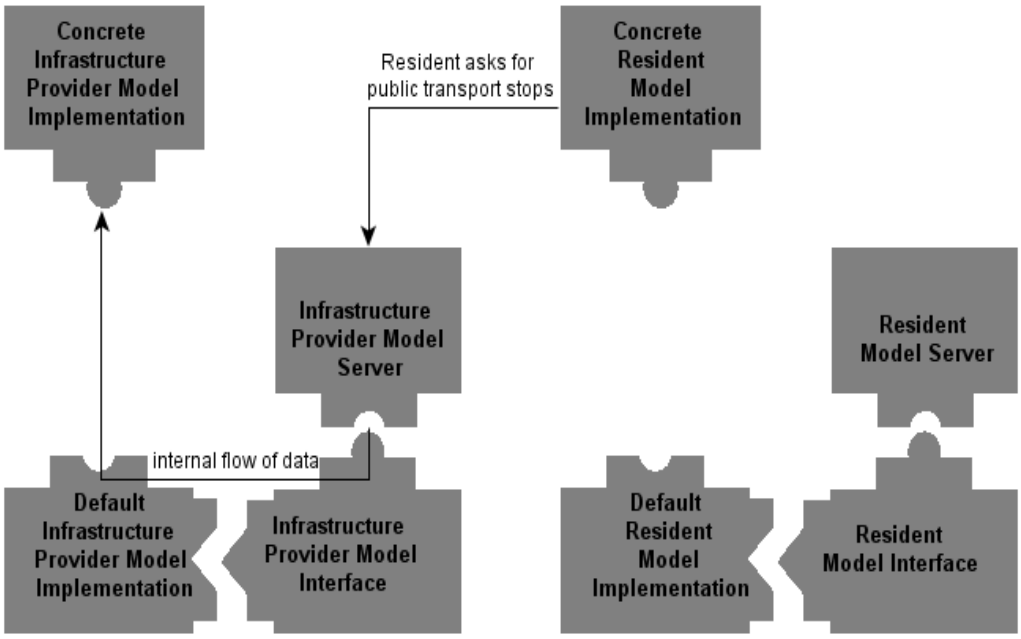


*Figure 3: Concept of interacting servers with residents and infrastructure as example*

### 3.1.2 Dummy models

This feature of ABMland is realized using so-called default implementations. At the moment, they are active if no specific model is implemented. Each implemented model extends a default implementation providing dummy values in case no concrete implementation for a specific model is available for a coupled simulation. This default implementation extends an interface that lists all the methods the model uses to export data. Servers forward the data to a coupled model. The server gets the information via the interface and either receives the data from the concrete model implementation or from the default implementation, if there is no concrete implementation available.

### 3.1.3 Detailed package description

The files are sorted into these main packages:

- abmland.common
- abmland.data
- abmland.mainctrl
- abmland.mainctrl.config
- abmland.models (interfaces)
- abmland.models.abstract_impl
- abmland.models.bindings
- abmland.models.config
- abmland.models.default_impl
- abmland.models.$AGENTNAME
- abmland.models.serve
- abmland.types (interfaces)
- abmland.types.impl

#### 3.1.3.a abmland.common

Contains classes mainly for handling data (generating, pre- and post-processing), bridging different interfaces (i. e. Repast S class usage) or data descriptions that do not fit directly.

#### 3.1.3.b abmland.data

Every quantifiable, enumeratable data fits here, when no furthermore requirements are given. Examples: "We have the following LU types..." → CLCover; "We distinguish 14 types of households..." → Householdtypes; "Agents' method-executions (processes) can be scheduled by these discrete priorities..." → Priorities.

#### 3.1.3.c abmland.mainctrl

Basically contains the same as abmland.models... packages, but more global controllers in their respect, as those special models access data from all the local agent controllers (models in our terms).

#### 3.1.3.d abmland.mainctrl.config

Mainly configuration data handler classes for the global controllers and agents (LanduseConf etc.).

#### 3.1.3.e abmland.types

Contains the data typically shared between the models, but is not captured by the Java-builtin type Map or similar. In this folder you will find the specifications of that data only, interfaces in computer science terms.

#### 3.1.3.f abmland.types.impl

Sometimes it is needed to make instances of the shared data. For this task use the implementation classes of the above described interfaces, but always try to assign the created objects to the most generic interface having those object's properties completely specified. So you will have to track only one point (where the creation occurs), in case the implementation changes. The implementation classes are located in that package.

#### 3.1.3.g abmland.models

This package contains the models to use in your API-client code, which means e.g.: use IbusinessModel to return something from a Business implementation.

#### 3.1.3.h abmland.models.abstract_impl

This package contains the abstract classes providing implementations not every implementation class of an arbitrary interface provides itself. Usually an abstract class leaves out those concretisations a client of the same interface should provide itself, like connection properties to a certain database or similar. All method concretisations are overwritable, if needed. The abstract implementations by default query a stochastic generator class for data.

#### 3.1.3.i abmland.models.default_impl

This package contains default implementation templates for the different model cases. They are also used to demonstrate the usage of the server classes below, but query the real default implementations of other models (in case that an export is not implemented,

determined by the interface, I. e. IResidentModel). Example: If you create a class ResidentModel it should extend the DefaultResidentModel out of this package. This means that it implicitly implements IResidentModel, which other model implementations know about.

Also they already show a basic usage pattern of the API:

```
private static IDeveloperModel developermodelsrv = new FDeveloperModelServer(
new DeveloperModelDefaultImpl());
```

What you see here is that you instantiate a special object the server (`FDeveloperModelServer`) for your implementation (which is `DeveloperModelDefaultImpl` here) and this all is connectable because all objects have the same base type (`IdeveloperModel`), which for your use case, completely describes the properties all these ojects have in common. You could i.e. plug a DeveloperMySpecialImpl in here, regarded it had the type IdeveloperModel. The FdeveloperModelServer is responsible for wrapping the other API clients calls to this object, i. e.:

```
public final ISpaceMap getOfficeSpaceSupplyForBusiness() {
 return developermodelsrv.getOfficeSpaceSupplyForBusiness();
}
```

This calls getOfficeSpaceSupplyForBusiness() on the FdeveloperModelServer which could internally call an arbitrary method, i. e. getOfficeSpaceSupplyForBusiness() (the equal names are coincidence) on the object having the type IdeveloperModel.

This way you can use an abstract class for your model, by letting it extend that and then in your source code use those abstract classes properties explicitly with the specifier super, i. e. (see AbstractBusinessModel):

```
super.getOfficeSpaceSupplyForBusiness()
```

to delegate the responsibility to request the correct developer model server to the abstract class (for implicit usage leave off your implementation and the prefix "`super`.").

#### 3.1.3.j abmland.models.config

Configuration handler and reader implementation classes also for individual agents.

#### 3.1.3.k abmland.models.serve

This package contains servers (delegate implementations) for the different model types. Use them when instantiating a certain model type. Their constructor argument is your concrete implementation of a certain model type. Only be sure to use the server with the same most common base type your implementation has.

#### 3.1.3.l abmland.models.$AGENTNAME

Contain your implementation of the agents canon.

#### 3.1.3.m abmland.models.bindings

Contain the runtime classes for actually running a simulation with Repast S – basically the MainContextBuilder.

### 3.1.4 Scheduling

The sub-models of ABMland need to be scheduled in order to ensure a correct flow of data during the simulation. The concept for scheduling the models uses the scheduled methods annotation of Repast S. This annotation @ScheduledMethod provides Repast S with the information on when to trigger a certain method.

For instance, the scheduling for the compute method of one specific model should look like this:

```
@ScheduledMethod (start = 2, interval = 1.0, priority = 998)
@Override
public void compute() {
   // compute
}
```

Accordingly, compute() starts in the second tick (in the first one, only initialisation takes place), goes on in all following steps, and has a priority of 998.

| **1st step** | **priority** | **Model & method** |
|---|---|---|
| 1 | ScheduleParameters.FIRST_PRIORITY | controllers: init() |
| 1 | Double.MAX_VALUE - 1 | sub-agent models: init() |
| 2 | ScheduleParameters.FIRST_PRIORITY | controllers: getData() |
| 2 | 1000 | controllers: compute() |
| 2 | 999 | sub-agent models: getData() |
| 2 | 998 | schedule controller: shallCompute() * |

*Table 2: Listing of the starting points and priorities of all scheduled methods.*

(*) The abmland.mainctrl.ScheduleController also has a @ScheduledMethod-annotated method shallCompute(), triggered according to Table 5. The sub-agent models have @Watch-annoted methods computeXYZ() where XYZ stands for the level of their ability to decide on things. Those @Watch-annotations are configured such that each of the annotated methods is triggered when each of both shallCompute triggers a state variable change in the schedule controller and the agent class with the @Watch-annotation has the correct level of competence to make a decision, checking of which also is configured directly within the annotation data.

So one could say this state variable is watched for togglement. If a toggle occurs and one of the methods watching for that togglement belongs to the agent with sufficient decision competence, computation inside the model may begin/is triggered.

### 3.1.5 ABMland-specific data types

A main feature of ABMland is the usage of specific data types. In order to minimise errors in communication between different agents, data types are specified for each communication interface. Thus, the models are able to show errors, if a corrupt value is being exported, e.g. a negative number of households per cell. Consequently, agents do not exchange pure Float, Integer or String values, but specific data types. Definitions of all data types are listed in the following table.

| Data type | Type | unit | min | max |
|---|---|---|---|---|
| ISupplyPerDemand | float | supply/demand | 0 | 1,000,000 |
| CLCover | enumeration of extended CORINE land cover classes | land cover type | not applicable | not applicable |
| ISpace | integer | square meter | 0 | 10,000 |
| IPressure | float | - | -1 | 1 |
| ISatisfaction | float | - | 0 | 1 |
| ILabourDemand | integer | no of jobs | 0 | 10,000,000 |
| IHouseholdNumber | integer | no of households | 0 | 100,000 |
| IMoney | integer | Euro | 0 | 10,000,000 |
| IPlan | enumeration of extended CORINE land cover classes | land cover type | not applicable | not applicable |

*Table 3: Definitions of all data types.*

All these types have an optional constructor parameter (which is set to false by default), signalling if the type is instantiated outside the case study area, requiring the flag to be set to true in that case otherwise to false. By this technique we are able to distinct between different meanings of Not-a-Number values, handling them as failure for in-case-study locations, for all other locations handling them in similar manner as outside locations in ESRI Raster files are encoded.

The following table depicts the extended land cover classes based upon CORINE land cover.

| Code | Name | Relationship to CORINE |
|---|---|---|
| 1901 | Town centre | CLC 111 /112 / 141 / 142 |
| 1902 | 19th century tenement | CLC 111 /112 / 141 / 142 |
| 1903 | Prefab multi-storey | CLC 111 /112 / 141 / 142 |
| 1904 | Multi-storey housing | CLC 111 /112 / 141 / 142 |
| 1905 | Single house, villa | CLC 111 /112 / 141 / 142 |
| 1906 | Park | CLC 111 /112 / 141 / 142 |
| 1907 | Urban forest | CLC 111 /112 / 141 / 142 |
| 1908 | Sports field | CLC 111 /112 / 141 / 142 |
| 1909 | Allotment | CLC 111 /112 / 141 / 142 |
| 1910 | Hospital | CLC 111 /112 / 141 / 142 |
| 1911 | School | CLC 111 /112 / 141 / 142 |
| 1912 | Other municipal buildings | CLC 111 /112 / 141 / 142 |
| 121 | Industrial or commercial units | CORINE |
| 122 | Road and rail networks and associated land | CORINE |
| 123 | Port areas | CORINE |
| 124 | Airports | CORINE |

| | | |
|---|---|---|
| 131 | Mineral extraction sites | CORINE |
| 132 | Dump sites | CORINE |
| 133 | Construction sites | CORINE |

*Table 4: Depiction of the extended land cover classes which are used in ABMland. For all non-artificial surfaces of CORINE land cover, the CLC codes are used. Note: CLC 111 Continuous urban fabric / CLC 112 Discontinuous urban fabric / CLC 141 Green urban areas / CLC 142 Sports and leisure facilities.*

## *3.2 ABMland wrapper*

ABMland wrapper consists of the class MainContextBuilder. It provides the master context participation role (see repast.simphony.essentials.RETest.setUp() (TM)). It may also directly instantiate the sub-contexts (e. g. model contexts) for a simulation if no model.score is used.

There is also TestMain_2 (in the test class path scope package abmland.frontends.cli) - for start an overall Repast S simulation run without the GUI.

Furthermore, different versions of the following configuration data files tuple are necessary:

- .rs folder
- .launch folder

The files in these are necessary to start models using the ABMland framework in differently configured modes, which we call Repast S -scenario. Thus, it depends on the concrete implementation of the ABMland framework and is not provided with the framework itself. However, the toy model implementation also encompasses a wrapper to build and run the simulation. See the toy model documentation for more information.

#### 3.2.0.a .rs folder

This consists of the files model.score, extended_params.xml, scenario.xml and further xml-files to configure the GUI and model output in Repast S.

#### 3.2.0.b .launch folder

This is the file directly configuring the Java invocation from inside Eclipse to fire up the Repast S GUI with our model classes.

## *3.3 Implementing a single model and respective agents*

In Repast S, two basic ways to implement agents are possible, namely using Java classes or using Groovy modelling. Groovy agents can be generated using a graphical agent editor provided by Repast S. But Groovy agents are hard to integrate into a more complex GIS model, so Repast S developers recommend using Java classes to build GIS agents. Therefore, the ABMland framework relies on Java classes as well. In the ABMland framework empty templates of all necessary classes are provided.

For a single agent, four classes are provided:

1. [Agent-name]Agent.java
   If you would like to create a new empty agent, create a new Java class and add

"implements I[Agent-name]" after the class name. Afterwards, hit the key "Ctrl" in parallel with the key "1" (applies for the recommended development tool Eclipse), click on "add unimplemented methods" to create all necessary method stubs.

2. [Agent-name]AgentContext.java
   In this class, the context needed for Repast S is created. In this class, paths to and names of shapefiles for initialising the agents are specified. At least for resident currently there is a default implementation of the context already in common-repasts which you may use. For your own agents implementations you needed to configure your agents' class name in the file config/params/ResidentConf.xml.
3. [Agent-name]Model.java
   The model is responsible for communicating with other sub-models. Therefore, all export data need to be filled (see empty method stubs). No methods for importing data are provided, because they are already implemented in the respective abstract model class provided by the framework. That means that you do not have to write your own importing methods and therefore, you do not need to know where the import values actually come from. But you can use the import values nevertheless. Example: The Business model needs to know if it has a building permission. The method "getBuildingPermissionForBusiness()", the Planning model provides against the other models, can be used by the Business model inside one of its methods, to ask the Planning model, therefore providing the Business agents with the resulting data.
   If you would like to create a new empty model, create a new Java class and add "extends Abstract[Agent-name]ModelImpl", afterwards add unimplemented methods.
4. [Agent-name]ModelContext.java
   In this class, the context needed for Repast S is created. In this class, the model belonging to the agent type, is added to the context. For resident there is a default implementation of the context already in common-repasts as an example.

### 3.3.1 Building the model

The model class is responsible for summarizing the information provided by all agents of this model to export the information to other agents in ABMland. Therefore, it is forced to implement the methods which are needed by other agents' models in ABMland to get information. In Eclipse, right-click on the name of the class (with the marked errors), and choose "add unimplemented methods" to include them into your code. These methods need to return the values you want to share with other agents in ABMland, so they are the most important feature of the model.

Furthermore, the model needs to implement the scheduling.

### 3.3.2 Building agents

In the agent class, the decision process of the individuals / groups / organisations is represented. To avoid having a very large agent class including all aspects of the decision making process, you can create new Java classes to outsource code.

# 4. Installation

The aim of this section is to describe all necessary steps before using the ABMland framework. NOTES:

1. All paths, files and so on relate to a tested example configuration with Windows XP(TM) and need to be adapted to the local installation by the user. This also relates to version numbers, vendors, …

2. The ABMland framework itself cannot be run independently. Instead, one uses it to implement specific agent-based models. See the toy model provided on www.ufz.de/abmland for an example!

## *4.1 Installation of Eclipse, Repast S and other resources*

### 4.1.1 Installation

Assuming you installed Eclipse – http://wiki.eclipse.org/Older_Versions_Of_Eclipse, choose the "Eclipse Classic 3.5.2" link for the download – with the following set of plugins (you will have to choose things matching your OS architecture though):

| Name | Version | Id |
|---|---|---|
| Eclipse SDK | 3.5.2.M20100211-1343 | org.eclipse.sdk.ide |
| GroovyFeature | 1.5.7.20081120_2330 | org.codehaus.groovy.eclipse.feature.feature.group |
| Maven Integration for Eclipse (Required) | 0.10.2.20100623-1649 | org.maven.ide.eclipse.feature.feature.group |
| repast.simphony_feature | 01.02.00 | repast.simphony_feature.feature.group |

*Table 5: Installation packages.*

For the selection of used update sites we used the following bookmarks file (import in Eclipse: "Help >> Install New Software... >> Clicking on 'Available Software Sites' link >> Import..."):

```
<?xml version="1.0" encoding="UTF-8"?>

<bookmarks>

  <site url="http://download.eclipse.org/releases/galileo" selected="true"
name="Galileo"/>

  <site url="http://m2eclipse.sonatype.org/sites/m2e" selected="true"
name="m2eclipse"/>

  <site url="http://mirror.anl.gov/pub/repastsimphony/site.xml" selected="true"
name="Repast Simphony"/>

  <site url="http://download.eclipse.org/eclipse/updates/3.5" selected="true" name="The
Eclipse Project Updates"/>
```

</bookmarks>

For further update site information see
"http://repast.sourceforge.net/docs/development.html" (Simphony, remove the check mark from "Group items by category" when installing),
"http://www.polarion.com/products/svn/subversive/download.php" (Subversion client) and
"http://m2eclipse.sonatype.org/installing-m2eclipse.html" (Maven integration). Some features not mentioned in the above list should be installed automatically by Eclipse package management mechanism. You need a recent version of Java (http://java.sun.com/javase/downloads/widget/jdk6.jsp) - best match is version 1.6, we have 1.6.0_21.

### 4.1.2 Adaptations

In preparation of the further steps needed, open the sample.eclipse.ini file in the project-setup working copy to see what your eclipse.ini, located where you installed Eclipse – i. e. "/java/eclipse" or "c:\program files\eclipse galileo\eclipse" should contain. The needed entries in the eclipse.ini are somewhat similar to:

```
-vm
h:/jdk1.6.0/bin/javaw.exe

…

-vmargs
-Denv.M2_RUNTIME=C:\Program Files\apache-maven-2.2.1
```

Important here is that the javaw executable from the JDK is used, not the one from the JRE (JDK is the development environment, JRE the runtime environment bundle). Make sure that you use and do not duplicate a potentially existing "-vmargs" entry here. Important - at least we recognized issues when running Mirosoft Windows Vista - are the upper case variants of the installation paths (used here: the JDK one and the Maven one) as they usually appear in a cmd.exe application window, when using "cd C:\Program Files\apache-maven-2.2.1" therein.

## *4.2 Installation of the ABMland framework*

ABMland creates plugins to be run within Repast S. For running a simulation, common-repast s, wrapper and all sub-models to be included have to be plugins. In the following, the sequence of setting up a simulation is described.

### 4.2.1 Creating a plugins folder

A folder is needed to contain all plugins created from ABMland. Thus, create a folder for your private plugins, such as E:/test/myplugins/plugins.

### 4.2.2 Include common-repasts as plugin

1. Move the folder common-repasts-1.0.2 into your private plugins folder.

2. Download the following missing libraries as jar-files into the lib-folder of the common-repasts plugin:

| **Library** | **Location** |
|---|---|
| org.functionaljava:fj:jar:2.12 | http://functionaljava.googlecode.com/svn/maven/org/functionaljava/fj/2.12/fj-2.12.jar |

| | |
|---|---|
| com.google.collections:google-collections:jar:1.0 | http://ftp.cica.es/mirrors/maven2/com/google/collections/google-collections/1.0/google-collections-1.0.jar |
| java-esri-ascii:javaRasters:jar:0.0.1.2 | http://java-esri-ascii.googlecode.com/svn/maven2/java-esri-ascii/javaRasters/0.0.1.2/javaRasters-0.0.1.2.jar |
| junit:junit:jar:4.7 | http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.jar |
| net.sourceforge.jexcelapi:jxl:jar:2.6 | http://repo1.maven.org/maven2/net/sourceforge/jexcelapi/jxl/2.6/jxl-2.6.jar |
| piccolo:piccolo:jar:1.0.3 | http://repo1.maven.org/maven2/piccolo/piccolo/1.0.3/piccolo-1.0.3.jar |
| org.simpleframework:simple-xml:jar:2.3.1 | http://repo1.maven.org/maven2/org/simpleframework/simple-xml/2.3.1/simple-xml-2.3.1.jar |

3. Rename the jar-files to match the names given here (thus: without version numbering):

    1. fj.jar
    2. google-collections.jar
    3. javaRasters.jar
    4. junit.jar
    5. jxl.jar
    6. piccolo.jar
    7. simple-xml.jar

### 4.2.3 Include wrapper as plugin and as a project

If you want to use the framework for implemented agent-based models, you need to bundle all models to run them within Repast S. In the toy model, we provide such an example for bundling called "wrapper". See the documentation of the toy model for installation.

### 4.2.4 Download maven helper file

To ease up your work in Eclipse, we prepared one launch configuration file. They are usually stored on your machine in a folder pointing to "${workspace_loc}/.metadata/.plugins/org.eclipse.debug.core/.launches", where "${workspace_loc}" points to the location, where your workspace used by Eclipse is created.

The launch file is located in the framework-install.zip and needs to be copied to your .launches folder. The new launcher configuration should automatically appear below the "External tools" knob. In case you changed something directly in the launch file with the editor, you might have to restart Eclipse first to have Eclipse seen your changes.

The working copy is made easily. If your workspace is still unused, you first have to create the .launch-folder. For this, once click the Run → "External Tools Configurations..." button and in the appearing dialog, double-click on the "Program" tree entry. Now your .launches folder should be there. Open workspace folder and therein navigate to ".metadata/.plugins/org.eclipse.debug.core/.launches". Copy the .launch configuration files vice versa from the working copy into the latter opened ".metadata/.../.launches" folder. Restart Eclipse.

#### 4.2.4.a [m2eclipse] clean compile + has Simphony(TM) deps

If you changed something in your project, which has dependencies somehow on Repast S – e. g. you're using "repast.*" imports in your sources... or you import something which has Repast S dependencies itself - and now wanted to check it, use that launch file to compile with the dependencies set up for Maven. Assumes you once set up the "$repast" variable in the launch dialog ("Main" tab, "Parameter Name, Value" table), but its current setting should suffice in most cases.

## *4.3 Adapt system paths*

### 4.3.1 Include common-repasts into configuration

#### 4.3.1.a Adapt manifest.mf

Modify the META-INF/MANIFEST.MF file in the repast.simphony.**gui** plugins folder (e. g. on our system in: E:\Programme\Eclipse Galileo\eclipse\plugins\repast.simphony.gui_1.2.0).

Add the following text before the line "Bundle-Vendor: %providerName":

Require-Bundle:
repast.simphony.core,repast.simphony.score,repast.simphony.score.runtime,saf.core.ui,repast.simphony.runtime,libs.piccolo;visibility:=reexport,common-repasts;visibility:=reexport

#### 4.3.1.b Adapt plugin_jpf.xml

Modify the plugin_jpf.xml file in the folder repast.simphony.**gui** plugins (e. g. on our system in: E:\Programme\Eclipse Galileo\eclipse\plugins \repast.simphony.gui_1.2.0).

After </attributes>, include the following:

```
<requires>

  <import plugin-id="saf.core.runtime"/>

  <import plugin-id="saf.core.ui"/>

  <import plugin-id="repast.simphony.essentials"/>

  <import plugin-id="repast.simphony.groovy"/>

  <import plugin-id="repast.simphony.score"/>

  <import plugin-id="repast.simphony.score.runtime"/>

  <import plugin-id="libs.piccolo"/>

  <import plugin-id="common-repasts"/>

</requires>
```

### 4.3.2 Inform Repast S about location of plugin folder

"repast.simphony.gui_1.2.0" (in our example case at E:\Programme\Eclipse Galileo\eclipse\plugins) is the folder for the repast.simphony.gui plugin. Only the plugins known there are seen by the Repast S class loader when the Repast S GUI is started. We use a folder separate from the Eclipse installation, so it is required to edit several files.

#### 4.3.2.a Adapt boots.properties

The boot.properties file in the folder repast.simphony.**runtime**_1.2.0 (you'll find it inside your Eclipse installations plugins folder) needs to be changed to include the plugin folder. For our example folder E:/test/myplugins the entry looks like this:

```
pluginFolders = ../,E:/test/myplugins/plugins
```

#### 4.3.2.b Adapt Repast S launch files

The launch file in the wrapper needs to be adapted. See the documentation of the toy model for the configuration.

#### 4.3.2.c Adapt Repast S xml files

In some Repast S-related xml files, absolute paths are given. This refers to all output that is stored to the disk (thus: NOT the displays and graphs, but the loggers). This is configured in the wrapper part of the simulation. Check the toy model documentation on this.

## *4.4 Optional: Update plugins*

If you have made changes to either wrapper (e.g. changing the scenario settings) or to one of the agents, you have to update the plugins in your private plugins folder. Maven is used to handle creating plugins and putting them into the necessary folders.

### 4.4.1 Export wrapper as plugin

1. Use the Run Configurations menu to generate the classes by invoking the `[m2eclipse] clean compile + has Simphony(TM) deps` goal once to generate the classes in the appropriate workspace folder prepared for copying.

2. In the folder META-INF of the project, open the manifest.mf with the Eclipse manifest editor.

3. On the "Overview" page, press the „export"-button.

4. On the options page, un-select „package plug-ins".

5. On the destinations page, give the directory as one folder above your private plugins folder.

Now, an updated plugin sould be built in your private plugins folder. Check that resources are also copied into the plugin (e. g. the .rs folder was copied to wrapper/target/classes).

### 4.4.2 Export an agent as plugin

Mark the agent project and choose the `[m2eclipse] clean compile + has Simphony(TM) deps` configuration to run as explained for wrapper.

After exporting it you should see the new plugin (a new folder e. g. "abmlandplanner_0.0.1") in the private plugins folder.

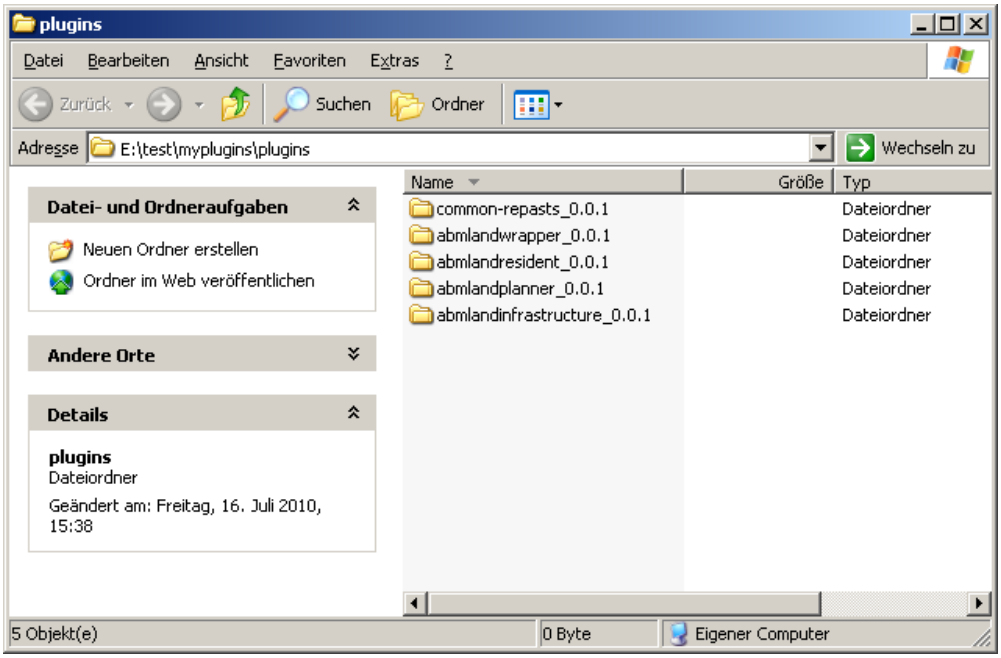


*Figure 4: Plugins folder with common repasts, abmlandwrapper and sub-models*

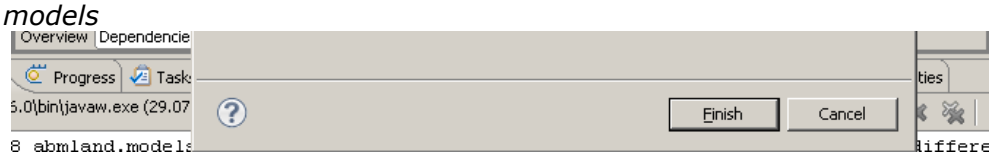


*Figure 5: Exporting sub-models as plugins*

# 5. Running a simulation

The Repast S GUI provides you with obvious transport buttons to initialise, completely run or successively step through, pause and reset a simulation round. Repast S uses xml files for the configuration of a simulation.

Each time code was changed (e.g. for an agent), the respective classes have to be created and included in a new plugin. Thus, the steps described above need to be done before a simulation can be run.

In the toy model, launch files and scenario configurations are already provided and are documented in the toy model documentation. In the following, a general description of files to be used for configuring Repast S is given.

## 5.1 Repast S launchers

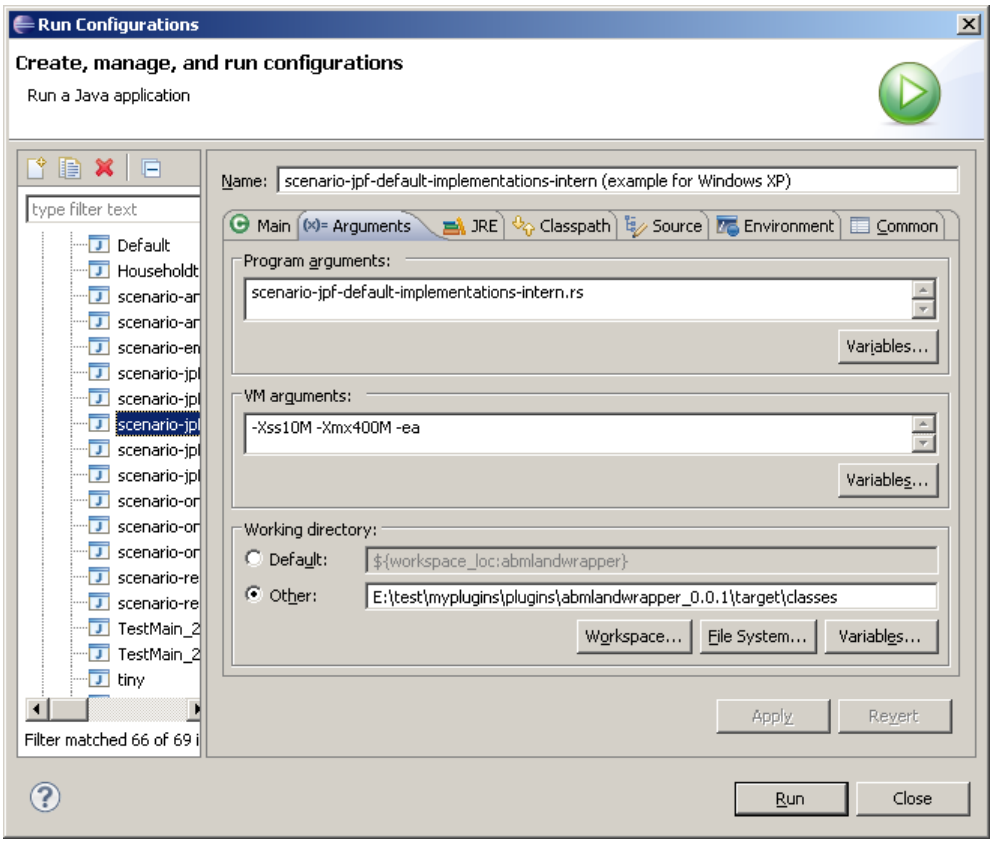


*Figure 6: Run configurations in Eclipse.*

In the folder launchers of the wrapper project, xml files with launch configurations for Repast S are stored. Brief comprehension, check that:

- "Main" tab: Project name is [project name]; Main class = repast.simphony.runtime.RepastMain
- "Arguments" tab: Program arguments are set to i. e.: "${workspace_loc:[project name]}/[model.score's folder]"
- "JRE" tab: Leave on defaults
- "Classpath": Leave on defaults
- "Source" tab: Leave on defaults
- "Environment" tab: -
- "Common" tab: Leave on defaults

## *5.2 Repast model.score*

If you want to include / exclude agent sub-models in a simulation, the model.score file needs to be changed. If a sub-model is not included in the model.score, all export data to be provided by this sub-model comes from its default implementation.

Every sub-model to be included has to have basically two entries below the ABMlandModels node. We will now show the settings for one of those by the example of a default implementation of the Resident agent, but only the ModelContext side, because the AgentContext side is fully analogue to it. The entries are cascaded as follows, assuming the names of the implementing Java files (as is the same for the generated class files) are the same (if you use different class names instead, that is no problem, simply use the Show Advanced button described above and customise the "Class Name" field).

Sample view of file *.rs/model.score in the Repast S Model Editor, expanded:

```
ABMlandModels
    ResidentModelContext
        ResidentModel
```

Now the look-like description for the remaining levels, shown only for the ResidentModelContext, because this you will have to edit manually. Things you may want to change are "Package" and "Class Name", depending on your agent profile's setup, though you can leave the proposed defaults here for compatibility reasons (the things said on naming consistency applies too, especially for ResidentModelContext. Base Path is empty as it is derived from the root contexts one):

Sample Properties view of file *.rs/model.score, the ResidentModelContext element expanded:

```
IDs
    ...
    Label:: ResidentModelContext
    ...
Implementation
    Base Path::
    Bin Directory:: common-repasts_0.0.1\target\classes
    Class Name:: ResidentModelContext
    ...
    Mode:: LOAD
    Package:: abmland.models.default_impl
    ...
```

Everything you add below this node, will derive the parent node's settings, so you have nothing more to customize, after simply adding the ResidentModel and ResidentAgent respectively.

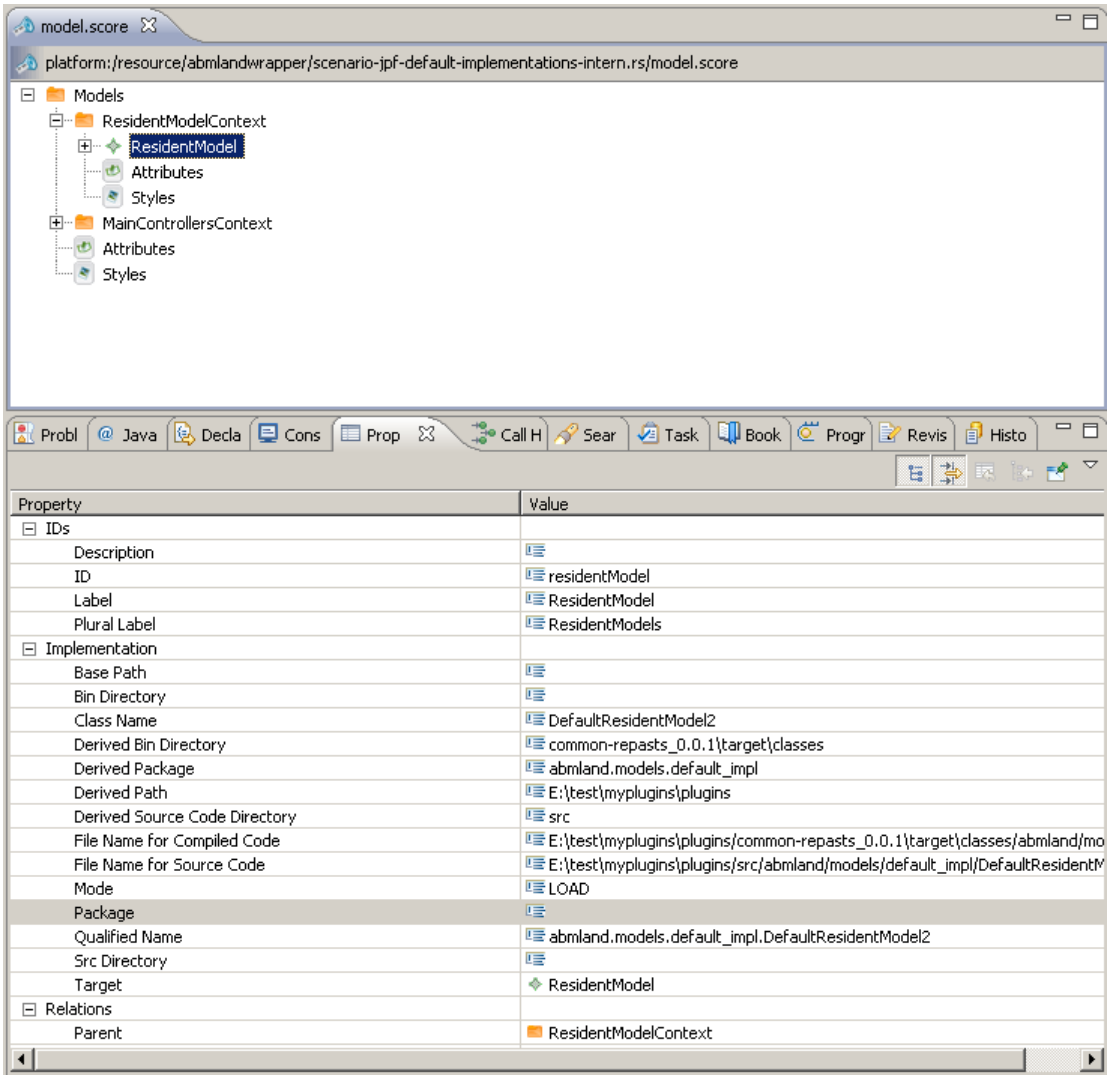


*Figure 7: Model score with ResidentModel.*

## *5.3 Repast S scenario settings*

The configuration of a single simulation run is stored in a xml file. The file scenario.xml is stored in a folder *.rs directly under the abmlandwrapper project folder. In terms of the Repast S implementation, the initial class of a model is referred to as a "Data Loader", so we can say the data loader of the model and its name (the "context" element in the file) are declared in scenario.xml, where the real name of the class realising the data loader, is referred to over an additional file i. e. "repast.simphony. dataLoader.engine.MainContextBuilder.xml", as you can see in the example below. This file only contains an element "string" with the fully qualified class name of the data loader realization, e. g.:

Sample "src/main/resources/common-scenario-files/repast.simphony.dataLoader. Engine. MainContextBuilder.xml" files contents:

<string>abmland.models.bindings.MainContextBuilder</string>

Furthermore, displays to be used for output during runtime can be included into this file (see below). A simple, running scenario.xml file would look like (assumed is that the files named "repast.simphony.action.Display[agentname]Agent.xml" exist):

Sample *.rs/scenario.xml" files contents (replace … with /abmlandwrapper/src/main/resources/common-scenario-files/):

```
<?xml version="1.0" encoding="UTF-8" ?>
<Scenario>
<repast.simphony.dataLoader.engine.ClassNameDataLoaderAction
context="ABMlandModels"
file="...repast.simphony.dataLoader.engine.MainContextBuilder.xml" />
<!-- Display for one agent follows here -->
<repast.simphony.action.display context="ResidentModelContext"

                file="...repast.simphony.action.DisplayResidentAgent.xml" />

</Scenario>
```

The string above in the xml files xpath "Scenario/repast.simphony.dataLoader.engine.ClassNameDataLoaderAction" – there the "context" attributes value – should be literally the same as the string you gave the label attribute in the root node of the model.score file. The same applies for the other "context" attributes values.

These xml files can easily be adapted: Hit "Run >> Run Configurations..." of Eclipse. The following list gives an overview where the corresponding ABMland-configuration files can be found that correspond to the Repast S files.

#### 5.3.0.a repast.simphony.action.Display*

Declared in xml configuration files included in *.rs/scenario.xml (i. e. src/main/resources/common-scenario-files/repast.simphony.action.Display*Agent.xml etc.)

#### 5.3.0.b repast.simphony.chart.engine.*

Declared in xml configuration files included in *.rs/scenario.xml (i. e. src/main/resources/common-scenario-files/repast.simphony.chart.engine.*Chart*Agent.xml etc.)

#### 5.3.0.c repast.simphony.data.engine.*

Declared in xml configuration files included in *.rs/scenario.xml (i. e. src/main/resources/common-scenario-files/repast.simphony.data.engine.DataSet*Agent.xml etc.)

#### 5.3.0.d repast.simphony.data.logging.outputter.engine.Outputter*

Declared in xml configuration files included in *.rs/scenario.xml (i. e. src/main/resources/common-scenario-files/repast.simphony.data.logging.outputter.engine.Outputter*Agent.xml etc.).

#### 5.3.0.e repast.simphony.dataLoader.engine.*

Declared in xml configuration files included in *.rs/scenario.xml (i. e. src/main/resources/common-scenario-files/repast.simphony.dataLoader.engine.*.xml etc.)

## *5.4 Displays*

Graphical outputs during runtime are controlled using the Repast S GUI (see e.g. tutorials on http://repast.sourceforge.net/docs/tutorial/SIM/index.html). It is possible to choose

attributes in order to show them on a map [Display]. Furthermore, graphs to summarise simulation results can be defined [Graph].

In the data structure provided, settings are stored in the "scenario.xml" (located in the "models.rs" folder of your "RepastS" project). Displays are defined in the respective xml files. These settings can be edited by changing the file or by changing settings in the "GUI" and saving them (disk symbol).

## *5.5 Using Repast S for exporting time series*

Repast S provides tools to save tabular text files for analysing time series. These text files have as many rows as time steps (ticks). If data are exported for agent types, the number of rows equals the number of agents multiplied with the number of ticks. The user can specify the columns. These colums can store one value per time step. This means e.g. values for all cells of a grid cannot easily be stored like that. This tool is rather appropriate for storing values of a single agent or the whole simulation area. See also the Repast S reference. The file scenario.xml controls this procedure.

To use the Repast S tools, one needs:

1. a data set

2. a data outputter

The scenario.xml file contains links to single xml files with the information regarding data sets and loggers. In general, the single xml files can be either changed using the xml files or using the Scenario tree in the Repast S GUI.

> In ABMland, it is strongly recommended to solely use the xml files themselves, because otherwise, the Repast S GUI will rename all xml files in the models.rs folder.

In the data set, the values which should be exported are specified. The exported values must be delivered by the respective model / agent class in a public method which provides a single value for a given agent/model in a certain time step. Repast S annotation can be used in order to facilitate the usage. In the data outputter, the name of the data set to be exported as well as the values are stated, together with the name of the output file and the value delimiters. The ResidentModel below has a method called mayCompute() whose values are exported in each time step:

```
public class ResidentModel extends AbstractResidentModelImpl implements
      DataListener {
   ...
   @Parameter(displayName = "May Compute", usageName = "mayCompute")
   public boolean mayCompute() {
      return mayCompute;
   }
   ...
}
```

The corresponding data set looks like this:

```
<repast.simphony.data.engine.DefaultDataGathererDescriptor>
  <name>DataSetResidentModel</name>
  <dataSetId class="string">ResidentModel</dataSetId>
  <scheduleParameters>
```

```
    <start>1.0</start>
    <interval>1.0</interval>
    <priority>-Infinity</priority>
    <duration>-1.0</duration>
    <frequency>REPEAT</frequency>
  </scheduleParameters>
  <dataMappingContainer
class="repast.simphony.data.logging.gather.DefaultDataMappingContainer">
    <nameMappingTable>
      <entry>
        <string>Tick</string>
        <repast.simphony.data.logging.gather.DefaultTimeDataMapping/>
      </entry>
      <entry>
        <string>mayCompute</string>
        <repast.simphony.data.logging.gather.MethodMapping>
          <method>
            <class>abmland.models.resident.ResidentModel</class>
            <name>mayCompute</name>
            <parameter-types/>
          </method>
        </repast.simphony.data.logging.gather.MethodMapping>
      </entry>
    </nameMappingTable>
    <mappingNameTable>
      <entry>
        <repast.simphony.data.logging.gather.MethodMapping
reference="../../../nameMappingTable/entry[2]/repast.simphony.data.logging.gather.Metho
dMapping"/>
        <string>mayCompute</string>
      </entry>
      <entry>
        <repast.simphony.data.logging.gather.DefaultTimeDataMapping
reference="../../../nameMappingTable/entry/repast.simphony.data.logging.gather.DefaultTi
meDataMapping"/>
        <string>Tick</string>
      </entry>
    </mappingNameTable>
  </dataMappingContainer>
  <aggregateContainer
class="repast.simphony.data.logging.gather.DefaultAggregateDataMappingContainer">
    <nameMappingTable/>
    <mappingNameTable/>
    <alternatedNameTable/>
    <nameAlternatedTable/>
  </aggregateContainer>
  <agentClass>abmland.models.resident.ResidentModel</agentClass>
</repast.simphony.data.engine.DefaultDataGathererDescriptor>
```

And the outputter like this:

```
<repast.simphony.data.logging.outputter.engine.DefaultFileOutputterDescriptor>
  <name>OutputterResidentModel</name>
  <outputterFactory
class="repast.simphony.data.logging.outputter.engine.DefaultFileOutputterDescriptor$1">
    <outer-class reference="../.."/>
  </outputterFactory>
```

```
  <dataSetHandler>
    <dataSets>
      <string>ResidentModel</string>
    </dataSets>
  </dataSetHandler>
  <streamHandler>
    <columns>
      <string>Tick</string>
      <string>mayCompute</string>
    </columns>
  </streamHandler>
  <isBatchAction>false</isBatchAction>
  <dataSetId class="string">ResidentModel</dataSetId>
  <fileName>output/ResidentModelOutput.txt</fileName>
  <insertTimeToFileName>true</insertTimeToFileName>
  <appendToFile>false</appendToFile>
  <writeHeader>true</writeHeader>
  <formatType class="repast.simphony.data.logging.outputter.LMDelimitedFormatter">
    <streamHandler>
      <columns/>
    </streamHandler>
  </formatType>
  <delimiter>;</delimiter>
</repast.simphony.data.logging.outputter.engine.DefaultFileOutputterDescriptor>
```

Exports for models and agents can be included easily by:

- copying and adapting the existing xml files for data set and outputter and
- including the names of these xml files into the scenario.xml.

## *5.6 Logging*

The log4j-framework is used to organise logging information of all simulation runs. In abmland-wrapper, two files mainly organise the logging-properties:

- MessageCenter.log4j.properties: organises logging levels and output into log files
- logging.properties: which models' output should be stored to log file on local disc?

Log-files are stored in [plugins-folder]\[wrapper-plugin]\target\classes\logs.

# 6. Data for initialisation

To use ABMland, some initialisation data are necessary. This refers to

- common initialisation as well as
- data for initialising agents.

ABMland uses the standard ESRI ascii file format for maps, with the header specifying the raster and values separated by spaces only. Data to initialise specific agents/models can be used in ABMland as well.

Data to initialise the overall framework are stored in the folder common-repasts/ src/main/resources/config. The minimum requirement for the framework to run is:

- Spatially explicit maps (folder .../maps/...)
  - A land use map
  - A map with land prices
  - A map with an initial distribution of households
- Parameter configurations (folder .../params/...)
  - land use: location and name of the land use map, attributes of the land cover classes specified for ABMland (LanduseConf.xml)
  - prices: location and name of the land prices map (MarketConf.xml)
  - maps in general: extent (MiscConf.xml)
  - households: location and name of the household map (ResidentConf.xml)

To facilitate the organisation of different initialisation files depending on different scenario configurations, the following scheme for searching for data was implemented: Both maps and parameter xml-files are organised in sub-folders following the pattern [case study]/ [start year]. Data are provided for an example grid with 30 by 30 cells and a start year of 2010. Thus, the map for initial land use is stored under common-repasts/src/main/resources/config/maps/GRID30BY30/2010/toylanduse30_30.asc. These settings are stored in the file extended_params.xml in the .rs-folder of abmland-wrapper.

Explanations of the data already provided can be found in the ABMland toy model documentation.

# 7. Adapting the framework

The ABMland tool can be adapted to (a) include other models and/or (b) data types and/or (c) different case studies. For (a), users have to

- modify the model.score file of their Repast S simulation project,
- for each new model implement a default implementation, a server and an interface,
- add two context classes (for model and agent).

To (b) include new data types, they need to be added with annotated boundaries into the respective package.

To (c) use another case study, users have to

- include the name of the case study in the file CaseStudy.java (common-abmland),
- add all necessary initialisation files using the pattern described above into common-repasts and the individual agent models,
- include the new case study in extended_params.xml in the .rs-folder of wrapper.