



Creating the Web- and Participatory Versions of Your Repast Simulation with MASS/PET

Repast Integration Manual and Tutorial

Created by Vilmos Kozma

For AITIA International and ELTE-IKKK



2007 October

Contents

1	Introduction	4
1.1	Agent-Based Modeling	4
1.2	MASS	4
1.3	Purpose	4
1.4	Benefits of integration	5
1.5	Prerequisites	5
2	The implementation of the integration	6
2.1	Relevant functional differences	6
2.2	The resulting system	6
2.3	Incremental integration	7
2.4	Integration levels	7
2.4.1	The default level	8
2.4.2	Visualizations	8
2.4.2.1	Visualization a of Repast model	8
2.4.2.2	Custom visualizations	8
2.4.3	The agent provider level	8
2.4.4	The agent configuration level	9
2.4.5	The participatory level	10
2.4.6	The removable level	11
2.5	Limitations and rules	11
2.5.1	Nullary constructor for Repast model	11
2.5.2	Security manager	11
2.5.3	I/O operations	11
2.5.4	Direct invocation of scheduled methods	11
3	Tutorial: Integrating the Heatbugs example model	12
3.1	Introduction of the Heatbugs model	12
3.2	Software requirements	13
3.3	Preparations	13
3.4	Reaching the default level	13
3.5	Installing the model into PET	14
3.5.1	Building a pet	14
3.5.1.1	Installing Apache Ant	14
3.5.1.2	Creating the pet archive	14
3.5.1.3	Custom project structure	14
3.5.2	Running the simulation	14
3.5.3	Uploading the pet archive into PET	14
3.5.4	Creating a new model	15
3.6	Extracting visualizations	18
3.7	Extracting the agents: The AgentProvider level	19
3.7.1	Defining the agents in the descriptor	19
3.7.2	Implementing the AgentProvider interface	20
3.7.3	Trying the model-family at AgentProvider level	20

3.8	Configuring Agents	21
3.9	Controlling agents	23
3.10	Making the bugs removable	26
4	Conclusions	28
5	Appendix: The model-family descriptor xml	29
5.1	The structure of the descriptor	29
5.1.1	The model-visualization element	29
5.1.2	The visualization-group element	29
5.1.3	The agent element	30
5.1.3.1	The field element	31
5.1.3.2	The field-defaults element	31
5.1.3.3	The user-interface-mappings element	32

1 Introduction

1.1 Agent-Based Modeling

Agent-based modeling is a branch of computer simulation. It models the individual, together with its imperfections (e.g., limited cognitive or computational abilities), its idiosyncrasies and personal interactions. The approach builds the model from 'the bottom-up', focusing mostly on micro rules and seeking to understand the emergence of macro behavior. Participatory simulation - a branch of agent-based simulation - is a methodology building on the synergy of human actors and artificial agents, excelling in the training and decision-making support areas. In participatory simulations some agents are controlled by users, while others are software governed.

1.2 MASS

The Multi-Agent Simulation Suite (MASS) is a software package intended to enable modelers to utilize the tools of agent-based simulation in various fields, without having to develop heavy programming skills.

MASS consists of four applications built around a simulation core. The simulation suite has its own core called the Multi-Agent Core (MAC), but it is also able to run on the popular Repast core. Being multi-core enables modelers to verify that results are core-independent, thus we plan to further develop this option. The Functional Agent-Based Language for Simulation (FABLES) is a programming language and its integrated modelling environment specially designed for creating agent-based simulations. The Model Exploration Module (MEME) is a tool that enables orchestrating experiments, managing results and has support for their analysis. The Participatory Extension (PET) is an optional web-based environment for multi-agent and participatory simulations. The fourth element of MASS, the Visualization Package does not translate into a standalone application. It consists of the various implementations of charts and visualizations used in all the other software.

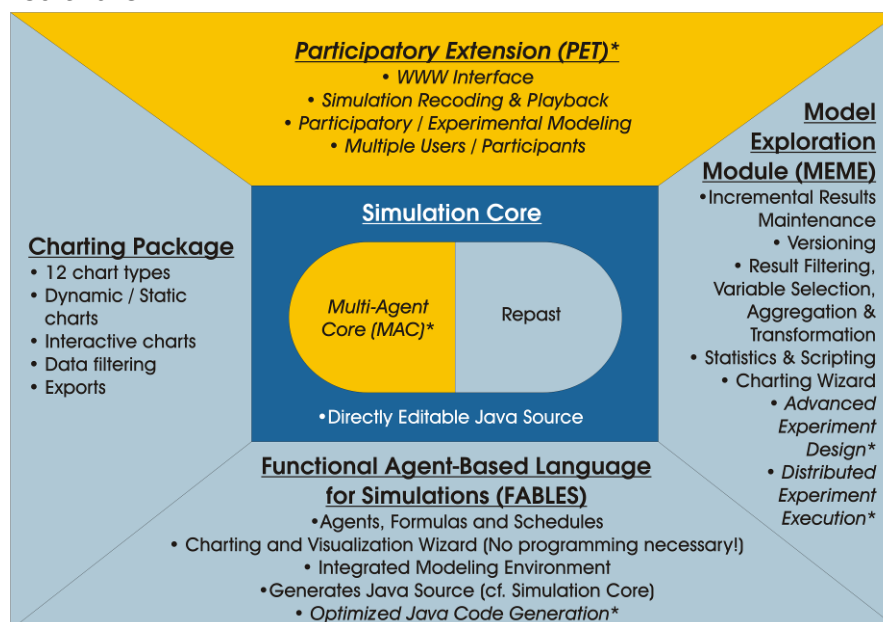


Figure 1 - Multi-Agent Simulation Suite

1.3 Purpose

The purpose of this document is to explain the process of integration of an existing Repast model to the Participatory Extension (PET). PET is a part of the Multi-Agent Simulation Suite (MASS), and it is a web based multi-user simulation environment,

capable of participatory simulation. It is based either on its native core, the Multi-Agent Core (MAC) or on Repast.

The paper goes through the steps of taking an existing Repast simulation and integrating it to PET. It describes the rules and related concepts, highlights the possible sources of bugs and details the incremental approach of integration. In addition it contains a tutorial, which is intended to ease the work of the modeler in the integration process.

1.4 Benefits of integration

A Repast model integrated to PET inherits all the functionality of PET which allows model writers to use their existing models in new dimensions. The following list sums up the potential advantages:

- The simulation runs in a Web environment therefore it can be viewed and controlled by multiple users simultaneously.
- Agents can be controlled by the user via the web. This means that the controller user is responsible for the actions instead of the agent.
- The simulation can be replayed after execution (including the actions taken by user-controlled agents).
- Visualizations of the Repast model are available in a web browser
- Agents can be preconfigured. This means setting their individual properties and adding them to the model, before the simulation is started.
- In the meanwhile, the model remains functional in the normal Repast environment as well.

1.5 Prerequisites

The reader of this document is expected to have the following abilities:

- To write and to understand Java code at a basic level,
- Basic knowledge of writing a Repast model,
- Administrator level knowledge of PET is an advantage (see the relevant MASS/PET documentation for the details).

2 The implementation of the integration

Both MAC and Repast are simulation toolkits, which help the user to write agent-based simulations through implementing the required interfaces and inheriting from the classes marked by the documentations. Both systems share some common concepts related to agent-based simulation and have some overlap as expected since they are all built around the same concepts. However there are many differences in how the different concepts have been taken into practice since the two toolkits were developed independently from each other. This enforced the process of integration development (the development process in which PET was enhanced with the ability to run a Repast simulation) to be well deliberate. Despite the fact that PET is Web-based while Repast is a desktop application the resulted system is an easy to use simulation development environment.

2.1 Relevant functional differences

- PET is a web application following the standards of JEE technology while Repast is a desktop application using Swing.
- In PET the user can control an agent moreover many users can control many agents simultaneously (at this point we can state that PET is participatory) while in Repast the single user is unable to feed input during the simulation is running.
- In Repast there are only global visualizations while in PET the visualization is basically bound to the agent and therefore it is capable to show the "world" from the agent's point of view.
- In Repast the agents are created and configured by the model while in PET they are added and configured by the user. The configuration process of a PET model includes the following steps:
 - Setting up the simulation properties
 - Adding agents to the model
 - Setting the properties of agents

It is important to mention that there is a naming conflict between the two systems concerning the model-family, model and simulation concepts.

In PET the concept of the written simulation code, configuration files and other necessary files are called model-family. The instantiation of a model-family creates a model which is a configurable representation of the simulation. By instantiating a configured model a simulation is created which finally can be run.

In Repast the concept of model-family does not exist. However based on the previous definition we can say that a Repast model-family is the code and all the accompanying files without the values of properties of agents and model defined in the code which should usually be the `setup()` method of the model class. The model is the whole code and accompanying files, while the simulation is the instance of the model.

2.2 The resulting system

The resulting system benefits from both PET and Repast although in general (from the aspect of users) the "Repast side" was enhanced to reach the functional level of PET. However there are specific parts in a Repast-based PET simulation which does not exist in a native PET simulation. A fully integrated Repast model can do almost everything that a PET model can while it still benefits from the large variety of Repast classes since it remains a Repast model. Under certain circumstances the same model can even run with PET and in native Repast mode at the same time.

2.3 Incremental integration

We designed the integration process to be as simple as possible by distinguishing separate phases in the process of integration. These phases are called integration levels and thus the process of integration is incremental. The multiple levels show how the Repast model fits into PET. It is important to mention that with minimal effort (writing of a ten lines long simple structured xml file) a good result is achieved which is the default level. The working of some PET functionality is not possible without the explicit assistance of the model writer but in common cases this additional work is not so much and thanks for the good design it is simple to do. To sum up: the more work is done the more functionality of PET is available.

2.4 Integration levels

Some of the integration levels are built on the top of others while some are independent from others. Note that it is not required to implement all the functionality. Figure 1 below shows a dependency tree of the integration levels.

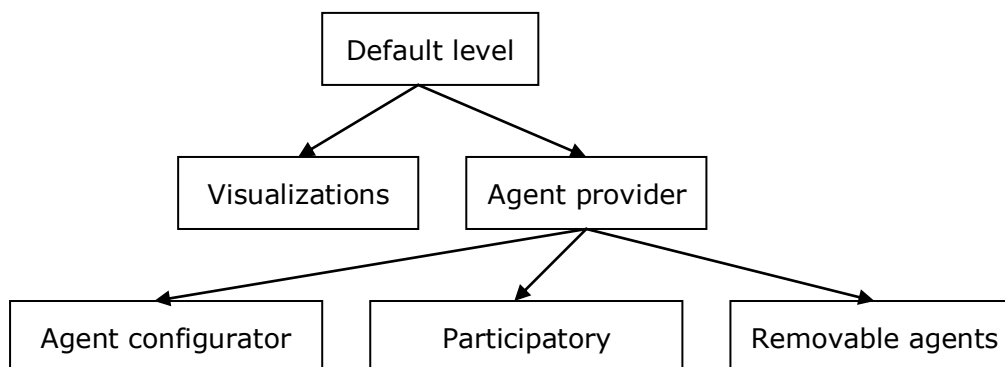


Figure 1 Dependency tree of integration levels

The following list explains the functionality that a level adds to a model:

- **Default level:** the user can control the simulation which includes starting, stopping, pausing and **replaying** it. Replaying the simulation is an important functionality which is added automatically. This allows repeating the simulation from start to end. After replay the simulation can be continued.
The simulation class is represented as a special agent named "Repast model agent". The init parameters of the Repast model are introduced as they were the parameters of this special agent.
- **Visualizations:** The visualizations of the simulation (`DisplaySurface` and `OpenGraph` instances) are visible on the web through an applet. Extracting the visualizations is automatic provided that certain conditions met. These conditions are discussed later.
- **Agent provider:** At this level the agents of the simulation are known to PET, so their properties can be set interactively by users over the web.
- **Agent configurator:** The configuring user can add agents to the Repast simulation through the web interface before the simulation starts. These agents can be configured by giving an initial value to its properties and setting the visibility and initialized state for each property.
- **Participatory level:** At this integration level the agents can be controlled by the user via the web! This means that in each round or in some of the rounds the controller user is responsible to take the actions instead of the agent.

- **Removable agents:** In PET (with certain settings) an agent might be removed from the simulation if the user who controls it does not issue command in time. PET is unable to remove an agent from a Repast simulation without the assistance of the Repast model. Therefore to support this behavior the Repast agent has to be extended with the [Removable](#) interface.

2.4.1 The default level

To reach the default level the developer needs to write a descriptor file which provides necessary information to PET on interpreting the Repast model. This is a strictly structured xml file, which is required by PET to recognize the installed Repast model. After the installation of the model a new model-family appears in PET that is ready to use. The installation process is discussed in detail under the chapter 3.5. You can read more about the descriptor file in the chapter 5.

2.4.2 Visualizations

2.4.2.1 Visualization a of Repast model

Visualizations coming from the Repast model are extracted automatically provided they are referenced in the descriptor file. PET identifies visualizations in the Repast model by assigning an index to them. This index must be referenced in the descriptor. It is important to understand the process how visualizations are extracted. PET uses two ways to do this:

- Through the [ai.aitia.mass.base.repast.abilities.DisplayProvider](#) interface:

```
public interface DisplayProvider
{
    public List getDisplays();
}
```

The only method of this interface [getDisplays\(\)](#) returns a list which contains all the displays that the model writer want to expose to PET. The elements of the list must be instances of [DisplaySurface](#) or [OpenGraph](#) (or descendants) classes. The reference index of a display will be its index in the returned list.

- If the interface is not implemented things become automatic. PET searches for [DisplaySurface](#) and [OpenGraph](#) (and descendants) objects in the simulation event listener and media producer lists of the [SimModelImpl](#) class. The reference index of the displays is in registration order. The list of [SimEventListeners](#) is iterated first and if a display exists in both lists then the reference index is set to its first occurrence. If the Repast model does not inherit (directly or indirectly) from the class [SimModelImpl](#) then the first approach is the only available solution.

2.4.2.2 Custom visualizations

It is possible to create custom visualizations for each agent type. This allows the model writer to create a visualization which shows the world from the point of view of an agent. Let's see the following example: the agents are captives, the world is a labyrinth and the goal is to escape from the labyrinth. In this case if the whole world is shown the task of a captive would be too simple. Instead we should visualize only the parts of the world which can be seen by the controlled captives.

To create such a visualization the model writer has to add a so called "detector method" to the agent class, write the rendering method and map these in the descriptor file.

We will not deal with this topic in detail here because it does not differ from the general PET practice that is discussed in the PET model writing manual and Tutorial documentation.

2.4.3 The agent provider level

With the exception of models deriving from [SimpleModel](#), a Repast simulation stores the agent objects wherever the model's creator wishes. Thus if we want PET to know about

the agents of the simulation, the Repast model must implement the `ai.aitia.mass.base.repast.abilities.AgentProvider` interface. This interface has only one method that is intended to provide a list that contains all agent objects that the model writer wants to expose to PET. This method might be called by PET any time after the Repast model's `begin()` method had been called. It is not expected to always return the same list object but that is recommended for performance considerations. The discussed interface is the following:

```
public interface AgentProvider
{
    public List getAgents();
}
```

2.4.4 The agent configuration level

A configurable model allows the user to add agents to the Repast simulation through the web interface before the simulation starts. These agents can be configured by giving an initial value to its properties. The `ai.aitia.mass.base.repast.abilities.AgentConfigurator` interface makes a Repast model configurable. It has two methods. The interface has two methods. The first one is:

```
public Object addAgent(String agentClass);
```

This method should create a new agent of the type defined by the parameter `agentClass` and install the new agent into the Repast model. It should also return the created agent or an instance of the class `ai.aitia.mass.base.repast.abilities.AddAgentError`. Here is the source of this class:

```
public class AddAgentError
{
    public enum AddAgentErrorType {NO_MORE_OF_THIS_TYPE, A_DIFFERENT_TYPE_FIRST,
AGENT_TYPE_NOT_CONFIGURABLE, NOT_ADDED, UNKNOWN_ERROR}

    private String value = "";
    private AddAgentErrorType type;

    public AddAgentError(AddAgentErrorType type)
    {
        this.type = type;
    }

    public AddAgentError(AddAgentErrorType type, String value)
    {
        this.type = type;
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public AddAgentErrorType getType() {
        return type;
    }

    public void setType(AddAgentErrorType type) {
        this.type = type;
    }
}
```

Sometimes the add operation fails or it is impossible to perform the addition in the current state of the simulation. In this case the `addAgent()` method should return an `AddAgentError` object, which indicates the cause of the failure in the `type` property. The `value` property holds a more detailed description of the error. The contents of `type` and `value` properties are displayed to the user who is configuring the model. The `type` property values have the following meanings:

- `NO_MORE_OF_THIS_TYPE`: The number of agents of the given type has reached the maximum limit.
- `A_DIFFERENT_TYPE_FIRST`: It is necessary to add an agent of different type first.
- `AGENT_TYPE_NOT_CONFIGURABLE`: This type of agent can not be added by PET.
- `NOT_ADDED`: The agent is not added for some model-specific reason.
- `UNKNOWN_ERROR`: An unknown error occurred

The second method of the `AgentConfigurator` interface is:

```
public boolean createDummyConfiguration () ;
```

This method should put the simulation into a state where calling the `begin()` method is safe. As the name "dummy" indicates this will be a dummy configuration and the simulation object will never be started. This method is necessary because PET creates trial simulation objects to obtain information on integration level, registered displays and default values of agent properties.

2.4.5 The participatory level

This functionality was achieved by assuming that every agent has at least one method (let's name it scheduled method) that is called by the Repast schedule object directly or indirectly and this method can be replaced by alternate user-callable methods. By feeding the information on scheduled and alternate method mappings into PET and performing byte code instrumentation on the class files of agents it is possible to take the control over agent objects. The byte code instrumentation is based on the descriptor file and is done with a helper program shipped with PET. Taking the control means that instead of the scheduled method a user-callable method is invoked depending on the controller user's will. Additional opportunity is setting the properties of the agent directly.

Let's see the following example: in its turn, an agent should move in one of four directions which are north, east, south and west. The scheduled method is `move()` and there are four alternate methods: `moveNorth()`, `moveEast()`, `moveSouth()` and `moveWest()`. When the agent is not controlled, the `move()` method is invoked by the scheduler in which a decision is made upon the moving direction and the move action is taken. Contrarily, if the agent is controlled it is doing the will of its controller (the user via the web interface), who may order the agent to move north, east, south, or west by issuing orders. It is important to mention that orders are not executed immediately but are buffered and the execution is done when the original `move()` method is called by the scheduler. The buffer for user orders stores only the latest order so the previous order (if was one) is always overwritten. This is done automatically by PET with the help of byte code instrumentation. The `move()` method is called whenever the scheduler calls it. If the controller user is not giving input then the agent can perform a default step operation, do nothing, or wait for input by blocking the whole simulation. This behavior depends on the wait and timeout policy settings of the model. For more information on the policy settings consult the PET Admin Manual.

A simulation is participatory when at least one agent is controllable. An agent is controllable when

- At least one of its scheduled methods is mapped in the descriptor file or it has one or more writable properties. (This information is also extracted from the descriptor file)
- The value of the `controllable` property of the agent is true

To learn more about the descriptor file please read chapter 5.

When a user takes the control over or releases an agent an event is generated for both actions. If the Repast simulation wants to know about these events the agent must implement the `ai.aitia.mass.base.repast.abilities.Controllable` interface.

```
public interface Controllable
{
    public void onTake () ;
```

```
public void onRelease();  
}
```

This interface has two methods `onTake()` and `onRelease()`, which are invoked on the agent by PET to inform the Repast model about these events. This feature is optional, but it can be useful. The simple example for this case is when a display wants to indicate the controlled state of the agents in their appearance (e.g. marking them with different colors).

2.4.6 The removable level

An agent is removable during the running of a simulation if it implements the `ai.aitia.mass.base.repast.abilities.Removable` interface. This interface has only one `remove()` method which is called by PET. This interface allows PET to inform the Repast model to remove the agent. Here is the interface:

```
public interface Removable  
{  
    public void remove();  
}
```

2.5 Limitations and rules

The created system has some limitations and rules that must be obeyed during the integration process.

2.5.1 Nullary constructor for Repast model

The repast model must have a nullary constructor (a constructor that gets no parameter), otherwise it can not be instantiated.

2.5.2 Security manager

The JEE environment that runs the repast simulation might have an installed security manager. In this case certain user written code will fail during the run and the simulation will not work. If that happens either remove the security manager from the Application Server or remove/replace the inadequate code. However with the default MASS/PET installation this problem should never occur.

2.5.3 I/O operations

This issue might arise if your Repast simulation reads/write a file. I/O operations are not always functional because the simulation runs in a JEE web application on the server side. However PET itself will not disable these functions so the success of a disk operation will depend on the running environment. Potential sources of problems are:

- Installed security manager (with the default installation this should never happen)
- Improper file structure
- Lack of write permission from the Operating System.

2.5.4 Direct invocation of scheduled methods

In a Repast model agents typically have one or more methods which are called by the scheduler. Sometimes, however, the model (or other agents) calls this scheduled method directly as well. Calling these methods while they are mapped in the descriptor file, and the agent is controlled by a user, may result in unexpected behavior because the actions stored in the buffer are executed each time the method is called.

3 Tutorial: Integrating the Heatbugs example model

This chapter goes through the steps of the integration of an example model showing how the theory works in practice. The example model is the Heatbugs model which is one of the canonical demos of agent based simulations.

3.1 Introduction of the Heatbugs model

The original heat bugs simulation consists of heat bugs - simple agents that absorb and expel heat and a heatspace, which diffuses this heat into the area surrounding the bug. Heat bugs have an ideal temperature and will move about the space in attempt to achieve this ideal temperature. The simulation has a display (see Figure 2 The heatbugs display) on which the user can see how bugs are moving and the space as diffusing heat.

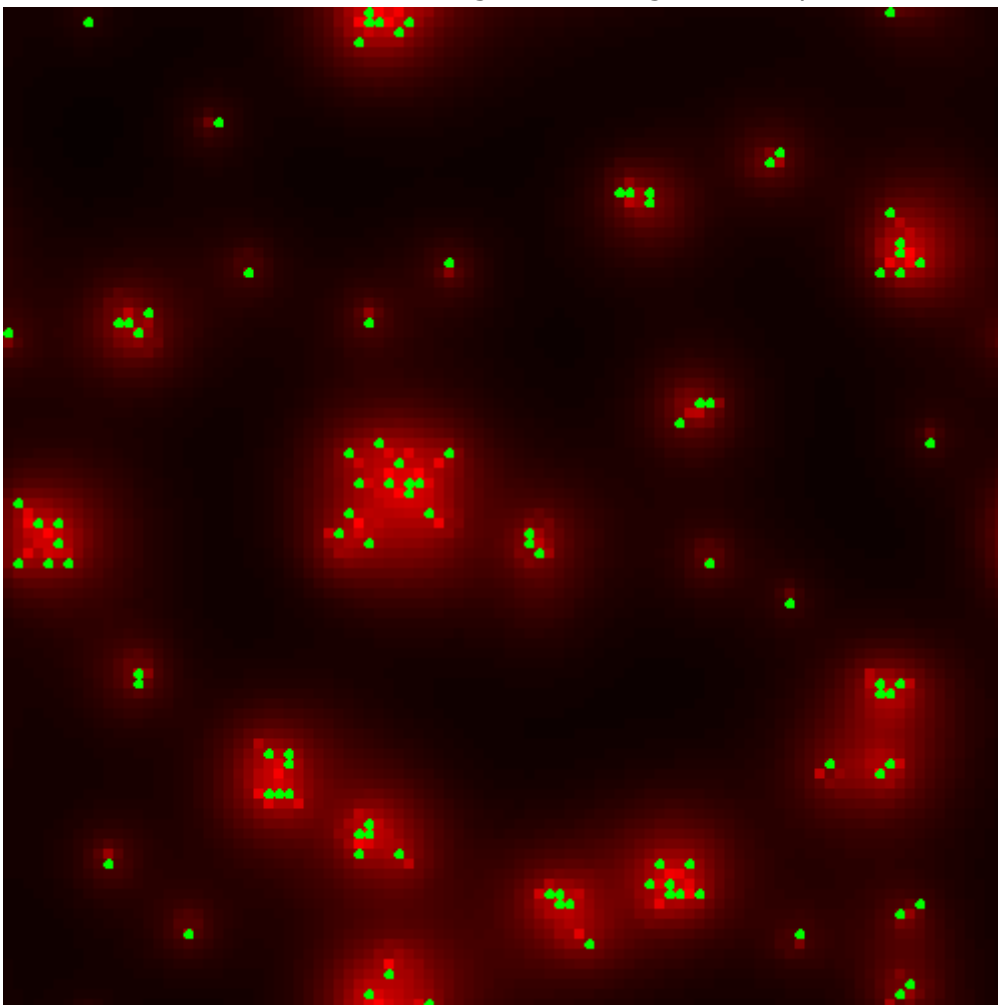


Figure 2 The heatbugs display

After the integration the Repast model benefits the following functionality:

- The model runs in a web application thus holding the advantages of a multi-user environment.
- Heat bugs become controllable. Users can take and loose the control over bugs. Controlled bugs move in the direction of the controller's choice and are displayed with a different color.
- Heat bugs can be added to the simulation and configured one by one or by group.

- The simulation can be replayed which includes the steps of controller users.

3.2 Software requirements

- Installed JDK 5.0 or later
- Installed and working PET application

3.3 Preparations

- Choose a specific location for your works. This is the place where you will put all the files that is required to create and build a PET-ready repast model. During this tutorial you will edit some of these files.
- Locate the original source files of the repast Heatbugs simulation and copy them (including package directories and required jar files) into your working location. The location of sources is `<PET installation dir>/ repast_integration/ tutorial/ original`. Copy the whole structure into your working directory.

Now you should have the directory structure shown by Figure 3.

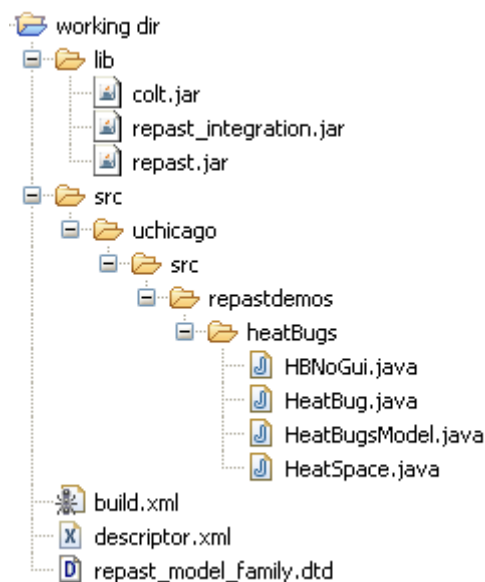


Figure 3 The tutorial's directory structure

3.4 Reaching the default level

Open the descriptor (descriptor.xml) with an xml editor (a simple text editor is appropriate too) and change the content to the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model-family SYSTEM "repast_model_family.dtd">
<model-family>
  <family-name>Test Repast HeatBugs</family-name>
  <model-class>uchicago.src.repastdemos.heatBugs.HeatBugsModel</model-class>
  <image>../images/heatbugs.jpg</image>
  <description>Repast based heatBugs simulation</description>
  <model-visualizations/>
  <agents/>
</model-family>

```

By doing this we have reached the default level. Now you can install and run the resulted model-family.

3.5 Installing the model into PET

The installation process has three steps:

- Building a pet file using the ant build script located in the root of your working directory.
- Uploading the created pet file into PET.

3.5.1 Building a pet

To make the process of building a pet file easier we created an Ant build script which you can use to create your PET model which can be uploaded into PET. Thus you will need to have an installed Ant on your computer.

3.5.1.1 Installing Apache Ant

- Download the latest binary distribution from Ant's home (<http://ant.apache.org>). (At the time of writing this tutorial the latest version is 1.7.1.)
- Unzip the downloaded file to an arbitrary location on your hard drive.
- Extend the Path variable of your Operating System to include the `bin` directory of the unzipped content. Now, in a command line tool type in the `ant -version` command. You should see a message similar to this: "Apache Ant version 1.7.1 compiled on June 27 2008".

To learn more about Apache Ant visit its home page.

3.5.1.2 Creating the pet archive

Now you can build a pet archive from the sources in your working directory by simply issuing the `ant` command from your working directory. This will create a file named `integration_tutorial.pet` in the `<working dir>/build` directory.

3.5.1.3 Custom project structure

The build script finds its files relatively to the working directory. The used directory names are hard-coded in the `build.xml` file so if you want a different project structure, feel free to modify the value of the top four ant property in `build.xml`. By doing this you can customize the script to work with your own model.

3.5.2 Running the simulation

Open a new browser window and go to the admin login page of PET.

- If you installed PET from the windows installation file then click on the Admin Login menu item under PET folder in the Start Menu.
- In any other case you can type the IP address of the PET server followed by `/pet/admin.do`. (For example: <http://127.0.0.1/pet/admin.do>) in a browser.

Login with administrator account.

3.5.3 Uploading the pet archive into PET

After login you are on the page of PET models' list. Locate and click the button labeled as "**Import Repast based model family**" at the top of the model list. See Figure 4.

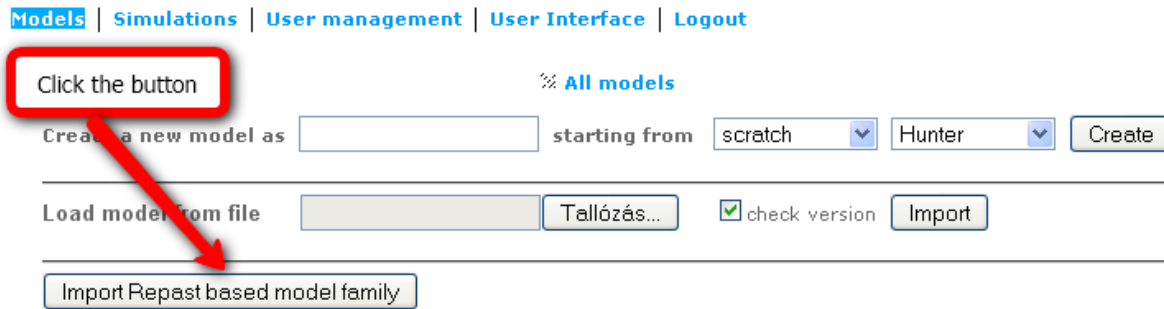


Figure 4 Importing a Repast based model family

On the upload page you can select the pet archive and upload it into PET as shown on Figure 5.

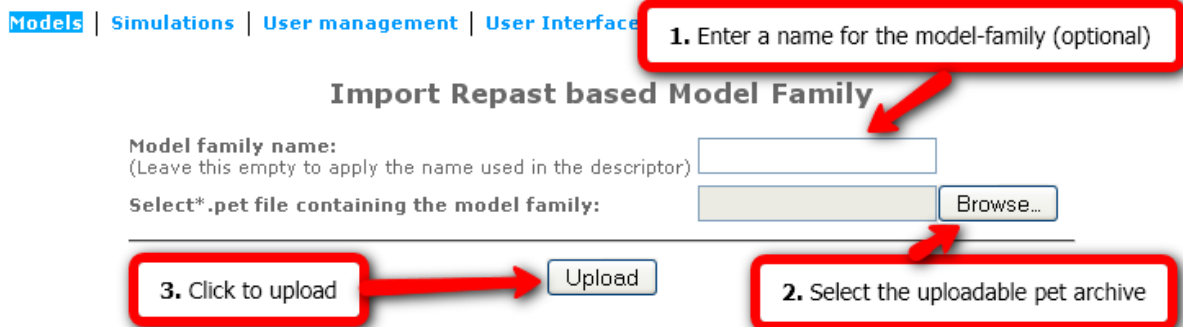


Figure 5 Uploading a repast based model family

The field "**Model family name**" is optional since this information can be extracted from the model. However filling in that field will override the setting in the model.

3.5.4 Creating a new model

After login in the **Create new model** section of page, type in a new name for this model and select "starting from **scratch**". In the next dropdown list your newly installed model-family should be enlisted. If not then check the log file of PET application. You can infer the location of log file by opening the file: [WEB-INF/config/log4j.properties](#) and check the value assigned to the property: `log4j.appender.F.File`. If everything is ok then select the name of the newly created model-family. Click on the **Create** button. See Figure 6 below.

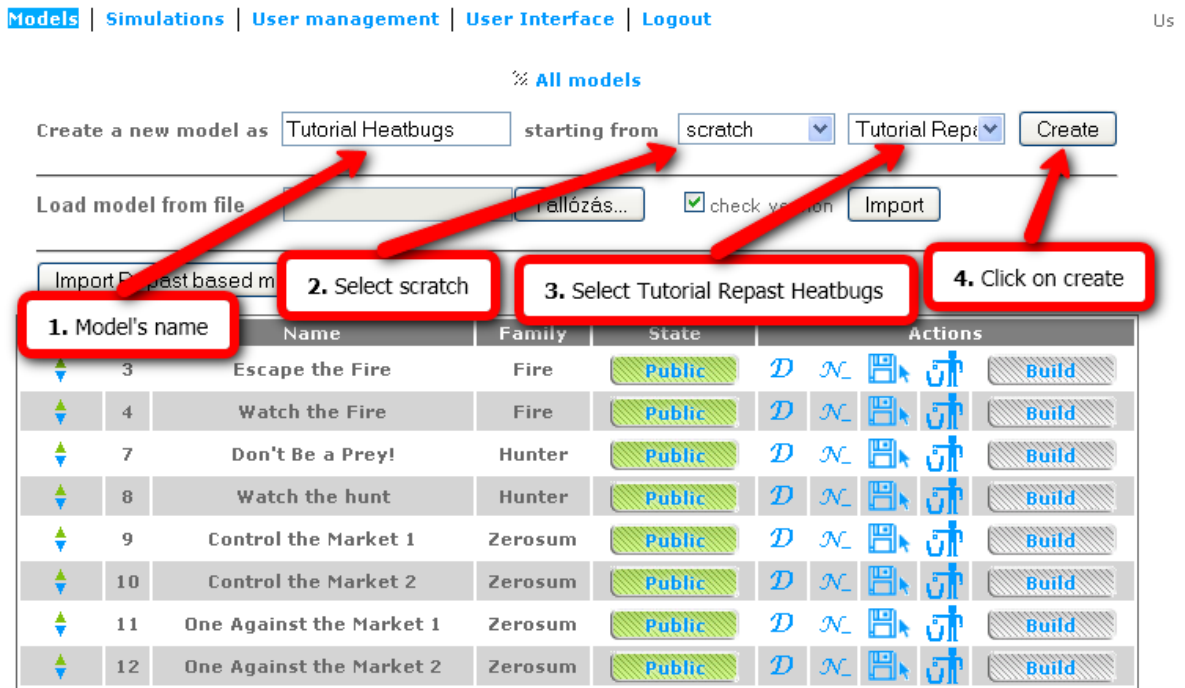


Figure 6 creating a model

After the model is created click on the *Private* button to make it public and then click on its name to configure it. Making a model public is necessary if you want to see this model enlisted when you login as normal user. See Figure 7 below.

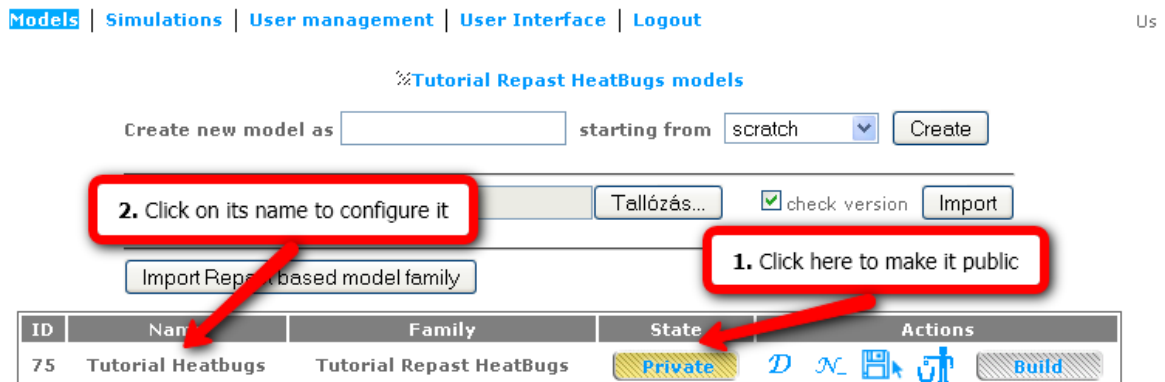


Figure 7 Making the model public

Configuration of a model at the default level consists of the following steps:

- Configuring the parameters of the Repast model.
- Configuring the simulation properties.




If you configured the model then click on the build simulation link to create a simulation. See Figure 8 below:

[Models](#) | [Simulations](#) | [User management](#) | [User Interface](#) | [Logout](#)

↳ [Edit model \(Tutorial Heatbugs\)](#) | [List agents](#) | [Groups](#) | [Description](#) | [Build simulation](#)

⌘ [Edit model Tutorial Heatbugs \(18\) of type Tutorial Repast](#)

⌘ [Agents](#)

Type	Class	Number	Actions
Repast model	ai.aitia.mass.base.repast.RepastModelAgent	1	  

⌘ [Simulation properties](#)

Property	Class	Value
Pause at	java.lang.Integer	-1
Random seed	java.lang.Long	1195478452750
Release Policy		Continue
Timeout		300
Timeout policy	java.lang.Integer	Call on timeout
Type of Event Logger DAO	java.lang.String	Hibernate
Visualization	java.lang.String	None
Wait policy	java.lang.Integer	Timeout

Figure 8 Configuring a model and building a simulation

The configuration of the Repast model parameters is shown by Figure 9.

[Models](#) | [Simulations](#) | [User management](#) | [User Interface](#) | [Logout](#)

↳ [Edit model \(Tutorial Heatbugs\)](#) | [List agents](#) | [Groups](#) | [Edit agent \(repastModelAgent\)](#)

⌘ [Edit agent #1302 of type repastModelAgent](#)

Property	Class	Value	Visibility	Initialized
maxIdealTemp	java.lang.Double	1.0	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false
maxOutputHeat	java.lang.Double	0.99	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false
minIdealTemp	java.lang.Integer	31000	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false
minOutputHeat	java.lang.Integer	10000	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false
numBugs	java.lang.Integer	17000	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false
worldXSize	java.lang.Integer	3000	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false
worldYSize	java.lang.Integer	100	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false

Figure 9 Setting the model parameters

By clicking on the Build simulation in the model editing page a new simulation can be created. Now click on it and see how the simulation runs. Now, with the control buttons you can start, stop and replay the simulation.

Mode: Simulation Simulation status: Ready

Name: Tutorial Repast HeatBugs 1 You are the owner of this simulation.

Type: Tutorial Repast HeatBugs

Created at: Mon Nov 19 13:30:57 CET 2007

Tick count: 0

You can Start/stop and replay the simulation with the control buttons

Property	Class	Value
Current tick	java.lang.Long	0
Identifier	java.lang.Long	13
Last tick	java.lang.Long	0
Pause at	java.lang.Integer	-1
Random seed	java.lang.Long	1195475875640
Release Policy	java.lang.Integer	Continue
Simulation status	java.lang.Integer	Ready
Timeout	java.lang.Long	300
Timeout policy	java.lang.Integer	Call on timeout
Type of Event Logger DAO	java.lang.String	Hibernate
Visualization	java.lang.String	None
Wait policy	java.lang.Integer	Timeout

Figure 10 Starting the simulation

Now start it and see how the step-counter starts incrementing. At this point we don't see too much because Repast visualizations are not extracted.

3.6 Extracting visualizations

In the descriptor file change the following line:

```
<visualizations/>
```

to:

```
<model-visualizations>
  <model-visualization index="0" display-name="Default display"
    refresh-rate="100"/>
</model-visualizations>
```

With this section we declare that we want to use the model's zero indexed visualization. For more information about the indexing mechanism see chapter: 2.4.2.1. By referencing a display of a Repast model in the descriptor the display will be available in PET for selection as the model's visualization.

You can reinstall the model-family now by following the instructions of the chapter 3.5. See how the Heatbugs display appears on the simulation page. Remember that by default no visualization is selected for the model so you have to do it on the Edit model page. See Figure 11 and Figure 12.


```

        <displayName>Unhappiness</displayName>
        <shortDescription>Unhappiness</shortDescription>
    </field>
    <field access="rw" name="x">
        <displayName>x coordinate</displayName>
        <shortDescription>x coordinate</shortDescription>
    </field>
    <field access="rw" name="y">
        <displayName>y coordinate</displayName>
        <shortDescription>y coordinate</shortDescription>
    </field>
    <field access="r" name="idealTemp">
        <displayName>Ideal temperature</displayName>
        <shortDescription>Ideal temperature</shortDescription>
    </field>
    <field access="r" name="randomMoveProb">
        <displayName>randomMoveProb</displayName>
        <shortDescription>randomMoveProb</shortDescription>
    </field>
</fields>
<field-defaults>
    <agent-field-default name="visualisation_Pet" visible="true"
        initialized="false" value="0" />
</field-defaults>
<user-interface-mappings />
<detectors />
<image>../images/bug.jpg</image>
<visualizations>
    <model-visualization-for-agent index="0"
        display-name="Repast bugs" />
</visualizations>
<description>Heat bug</description>
</agent>
</agents>

```

In the above xml segment we define the implementing class of the agent, its fields and in addition we make the model's zero indexed visualization available to each bug and set the default value of the visualization property for each bug to the assigned visualization.

3.7.2 Implementing the AgentProvider interface

Extend the class `HeatBugsModel` with the interface `AgentProvider` and add the `getAgents()` method to the class:

```

public class HeatBugsModel extends SimModelImpl implements AgentProvider
{
    ...

    public List getAgents() {
        return new ArrayList(heatBugList);
    }
}

```

In the above `getAgents()` method we create a list which contains all the agents of the model. Instead of just returning the existing list `heatBugList` it is advised to create a new list because PET modifies it for technical reasons. Again, you can reinstall the model-family by following the instructions of the chapter 3.5.

3.7.3 Trying the model-family at AgentProvider level

At agent provider level, at the edit model page you should see all agent types that were extracted from the Repast model. See Figure 13.

Models | Simulations | User management | User Interface | Logout U:

↳ Edit model (Tutorial Heatbugs) | List agents | Groups | Description | Build simulation

⌘ Edit model Tutorial Heatbugs (18) of type Tutorial Repast HeatBugs

⌘ Agents

Type	Class	Number	Actions
Repast model	ai.aitia.mass.base.repast.RepastModelAgent	1	
Heat bug	uchicago.src.repastdemos.heatBugs.HeatBug	?	

A red box highlights "Heat bug agent" with an arrow pointing to the "Heat bug" row in the table.

Figure 13 Heat bug is extracted

In addition, after building a simulation and selecting the **Agents** submenu you can see the list of heat bugs coming from the Repast model. See Figure 14.

Models | Simulations | User management | User Interface | Logout User: admin

↳ Simulation Control>Tutorial Repast HeatBugs 1 | Agents | Agent properties

⌘ Simulation Control (Tutorial Repast HeatBugs 1)

Mode: Simulation Simulation status: Ready
You are the owner of this simulation.

Name: Tutorial Repast HeatBugs 1
Type: Tutorial Repast HeatBugs
Created at: Mon Nov 19 16:24:36 CET 2007
Tick count: 0

⌘ Default display Block(s)

A red box highlights the "Agents" menu item with the text "Click here to see the list of heat bugs".

Figure 14 Selecting the **Agents** submenu

3.8 Configuring Agents

Configuring agents means adding them to a model by hand and then setting the properties of added agents. To enable this function the model class needs to implement the `AgentConfigurator` interface. This class inherits directly from `AgentProvider` adding two new methods to our model: `addAgent(String agentClass)` and `createDummyConfiguration()`.

```
public Object addAgent(String agentClass) {
    if (HeatBug.class.getName().equals(agentClass)) {
        return createHeatBug();
    } else {
        return new AddAgentError(AddAgentErrorType.UNKNOWN_ERROR,
            "Unknown agent type.");
    }
}

public void createDummyConfiguration() {
}
```

The above code implements the necessary methods according to specification described in earlier chapters. For practical reasons we extracted the heat bug creation code into a new method: `createHeatBug()`.

```
private HeatBug createHeatBug() {
    int idealTemp = Uniform.staticNextIntFromTo(minIdealTemp, maxIdealTemp);
    int outputHeat = Uniform.staticNextIntFromTo(minOutputHeat, maxOutputHeat);
    int x, y;

    do {
        x = Uniform.staticNextIntFromTo(0, space.getSizeX() - 1);
        y = Uniform.staticNextIntFromTo(0, space.getSizeY() - 1);
    }
```

```

} while (world.getObjectAt(x, y) != null);
HeatBug bug = new HeatBug(space, world, x, y, idealTemp, outputHeat,
                           randomMoveProbability);
heatBugList.add(bug);
return bug;
}

```

The `buildModel()` method also changed:

```

private void buildModel() {
    for (int i = 0; i < numBugs; i++) {
        createHeatBug();
    }
}

```

As it is seen to remain simple the creation of bugs is done with our new method. Another change here is that the creation of `space` and `world` object has moved to the end of the `setup()` method. This is necessary because the `addAgent()` method is called before `buildModel()` is called and a created agent requires both the `space` and `world` objects in its constructor. Copy this two lines to the end of `setup()`:

```

space = new HeatSpace(diffusionConstant, evapRate, worldXSize, worldYSize);
world = new Object2DTorus(space.getSizeX(), space.getSizeY());

```

Now reinstall the model-family by following the instructions of the chapter 3.5. and see what changes on the web-interface.

First note that on the **Edit model** page the Add heatbugs icon changed from gray to blue indicating that adding heat bugs to the model is now possible. See Figure 15.

[Models](#) | [Simulations](#) | [User management](#) | [User Interface](#) | [Logout](#)

↳ [Edit model \(Tutorial Heatbugs\)](#) | [List agents](#) | [Groups](#) | [Description](#) | [Build simulation](#)

⌘ [Edit model Tutorial Heatbugs \(18\) of type Tutorial Repast](#)

⌘ [Agents](#)

Type	Class	Number	Actions
Repast model	ai.aitia.mass.base.repast.RepastModelAgent	1	
Heat bug	uchicago.src.repastdemos.heatBugs.HeatBug	0 + ?	

Click the icon to add heat bugs to the model

Figure 15 Adding new heat bugs to the model

The process of adding agents is showed by Figure 16. As it is visible we add ten heatbugs to the model. The value of `controllable` property is set to true so in the created simulation ten heat bugs will be controllable. The coordinates of heatbugs are set to 50, 50 so they will be placed in the center of the display when the simulation starts. The ideal temperature is 20000.

Models | Simulations | User management | User Interface | Logout

↳ Edit model (Tutorial Heatbugs) | List agents | Groups | New agent (Type) | Add new agent - heatBug

1. Set the values of properties

2. Set the visibility and initialized state of properties

Property	Class	Value	Visibility	Initialized
Controllable	java.lang.Boolean	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Detectors visible	java.lang.Boolean	<input type="radio"/> true <input checked="" type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Ideal temperature	java.lang.Integer	20000	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
randomMoveProb	java.lang.Float	0.0	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Unhappiness	java.lang.Double	0.0	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
User step size	java.lang.Integer	Small	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visible	java.lang.Boolean	<input checked="" type="radio"/> true <input type="radio"/> false	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
Visualization	java.lang.String	Repast b	<input type="radio"/> true <input checked="" type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
x coordinate	java.lang.Integer	50	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false
y coordinate	java.lang.Integer	50	<input checked="" type="radio"/> true <input type="radio"/> false	<input checked="" type="radio"/> true <input type="radio"/> false

3. Set the number of bugs to add

Add 10 piece(s)...

4. Click the Add button

Figure 16 Adding agents to a model

3.9 Controlling agents

In order to control agents in PET we need to add some code to the agent class and the descriptor. First append the following `<field>` element in the descriptor file to the end of `<fields>` section:

```
<fields>
...
...
<field access="rw" name="userStepSize">
  <displayName>User step size</displayName>
  <shortDescription>Describes how large a step is
    that a user can take with one action
  </shortDescription>
  <constants>
    <constant name="Small" value="1"/>
    <constant name="Medium" value="2"/>
    <constant name="Larger" value="3"/>
    <constant name="Even more large" value="4"/>
    <constant name="Largest" value="5"/>
  </constants>
</field>
</fields>
```

The above xml fragment defines a new property to the heat bug class with constant values. With the help of this property when we control an agent we can define the size of a step. Although this is an optional extension of the model it is good for demonstrating how constant values can be defined to a property. Add this property with its getter/setter methods to the `HeatBug` agent class:

```
private int userStepSize;

public int getUserStepSize() {
  return userStepSize;
}

public void setUserStepSize(int userIncrement) {
  this.userStepSize = userIncrement;
}
```

Now we add user callable methods to `HeatBug.java` and create the corresponding mapping in the descriptor. The method called by the scheduler is `step()`. We add four alternate methods which can be called by the user: `left()`, `right()`, `up()`, `down()`.

```
public void left() {
    x = x-userStepSize < 0 ? world.getSizeX() -
        Math.abs(x-userStepSize) : x-userStepSize;
}

public void right() {
    x = x+userStepSize > world.getSizeX() ?
        x+userStepSize-world.getSizeX() : x+userStepSize;
}

public void up() {
    y = y-userStepSize < 0 ? world.getSizeY() -
        Math.abs(y-userStepSize) : y-userStepSize;
}

public void down() {
    y = y+userStepSize > world.getSizeY() ? y+
        userStepSize-world.getSizeY() : y+userStepSize;
}
```

The mapping in the descriptor is the following:

```
<user-interface-mappings>
  <scheduledMethod name="step" visible="true" displayName="Move"
    shortDescription="Move the heatbug">
    <timeoutMethod name="step"/>
    <humanCalledMethod name="left" displayName="Left"
      shortDescription="Move left"/>
    <humanCalledMethod name="right" displayName="Right"
      shortDescription="Move right"/>
    <humanCalledMethod name="up" displayName="Up"
      shortDescription="Move up"/>
    <humanCalledMethod name="down" displayName="Down"
      shortDescription="Move down"/>
  </scheduledMethod>
</user-interface-mappings>
```

Replace the row `<user-interface-mappings/>` with the above section. Now you can reinstall the model by following the instructions of the chapter 3.5.

As an optional task the model integrator may want to implement the `Controllable` interface which makes it possible to receive an event upon the value changes of the `controlled` property. In our example we use this information to change the color of the controlled agents. Extend the code of `HeatBug` class with the following:

```
private Color color = Color.GREEN;

public void onTake() {
    color = Color.WHITE;
}

public void onRelease() {
    color = Color.GREEN;
}
```

To imply the selected color change the body of `draw(SimGraphics g)` function in `HeatBug` to the following:
















```
public void draw(SimGraphics g) {
    g.drawFastRoundRect(color);
}
```

As a result the controlled agents appear on the screen with white color.

Now test the code by reinstalling the model family. You can do it by following the instructions of the chapter 3.5.

If you clicked on the **Agents** link on the **Simulation Control** page you can see the list of agents participating in this simulation. Now you can take an agent by simply clicking on a control agent icon. See Figure 17 below.

🔍 Agent list

Id	Type	Added by	Controlled	Actions
1302	 Repast model	User	false	 
1353	 Heat bug			 
1354	 Heat bug			 
1355	 Heat bug	User	false	 
1356	 Heat bug	User	false	 

Click one of the icons to take an agent

Figure 17 taking an agent

After taking an agent you can control it with the commands of the **Function group** box. You can easily distinguish the controlled heat bug from the others because it is displayed with white. After starting the simulation try clicking the commands in the **Function group** box and see how your heatbug moves. See Figure 18 below.

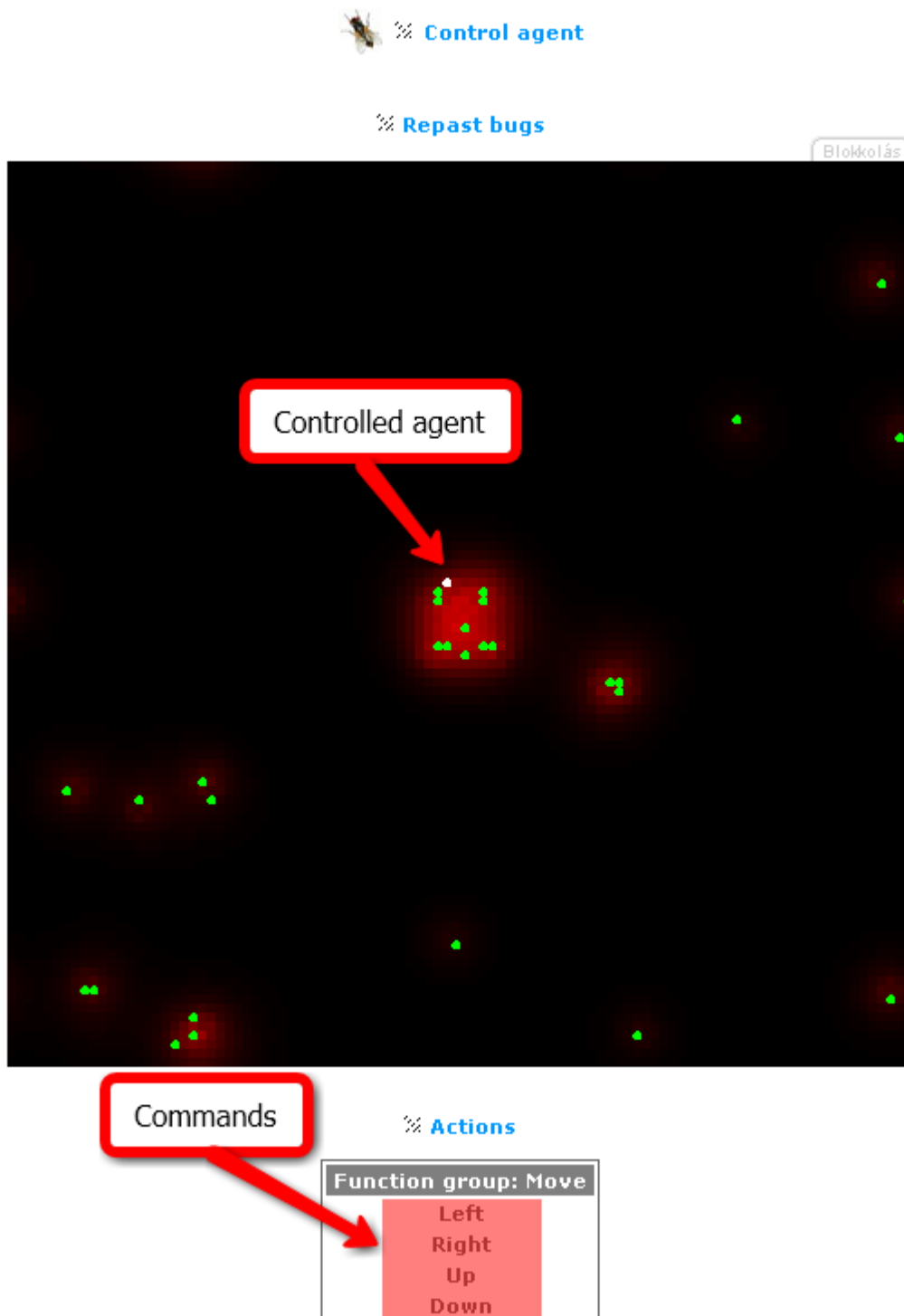


Figure 18 Controlling an agent

3.10 Making the bugs removable

In PET (with certain settings) an agent might be removed from the simulation if the user who controls it is not giving order in time. PET is unable to remove an agent from a Repast simulation without the assistance of the Repast model. Therefore to support this behavior the Repast agent has to be extended with the `Removable` interface. Extend the class `HeatBug` with the following:

```
public void remove () {
```

```
bugList.remove(this);
}
```

This interface is attached to the `HeatBug` class although the list of agents is stored in the model class. Therefore we must keep a reference of this list in each agent to enable them to remove themselves from the list. See the code below:

```
private ArrayList bugList;

public void setBugList(ArrayList bugList) {
    this.bugList = bugList;
}
```

We must invoke the `setBugList()` method for each bug after its creation so modify the `createHeatBug()` method in the model class as follows:

```
private HeatBug createHeatBug() {
    int idealTemp = Uniform.staticNextIntFromTo(minIdealTemp, maxIdealTemp);
    int outputHeat = Uniform.staticNextIntFromTo(minOutputHeat, maxOutputHeat);
    int x, y;

    do {
        x = Uniform.staticNextIntFromTo(0, space.getSizeX() - 1);
        y = Uniform.staticNextIntFromTo(0, space.getSizeY() - 1);
    } while (world.getObjectAt(x, y) != null);
    HeatBug bug = new HeatBug(space, world, x, y, idealTemp, outputHeat,
        randomMoveProbability);
    bug.setBugList(heatBugList);
    heatBugList.add(bug);
    return bug;
}
```

At this point our Repast model is fully integrated into PET. Now test the code by reinstalling the model family. You can do it by following the instructions of the chapter 3.5. You can test removing of agents by setting the models **Wait policy**, **Timeout policy**, **Release policy** properties as showed by Figure 19 on the "Edit model" page.

⌘ Simulation properties

Property	Class	Value
Pause at		-1
Random seed		1195654856423
Release Policy	java.lang.String	Remove agent
Timeout	java.lang.Integer	300
Timeout policy	java.lang.Integer	Release agent
Type of Event Logger DAO	java.lang.String	Hibernate
Visualization	java.lang.String	Default display
Wait policy	java.lang.Integer	Timeout

Update properties

Figure 19 Setting the policy properties

By setting the simulation properties as the Figure 19 describes, the controlled agent is removed from the running simulation if the controller user is not giving input for the agent for a period of time defined by the **Timeout** simulation property.

4 Conclusions

PET is an agent based modeling framework providing infrastructure for modelers to develop participatory simulations and run them in web-based environment. As simulation engine, PET uses its native core, the Multi-Agent Core or use Repast to allow the integration and running of existing Repast models with minimal effort. A PET integrated Repast simulation is empowered with the nature of a web application and inherits the functionality of PET which makes the Repast model participatory.

5 Appendix: The model-family descriptor xml

With the help of a properly installed descriptor file the PET system is able to recognize and collect information on model-families that are installed in the system. Each model-family has its own descriptor file. The general requirements of the descriptor are listed below:

- The file name must be `descriptor.xml`.
- The content of the file must follow the rules defined in the `<PET installation directory>/repast_integration/tutorial/sources/repast_model_family.dtd` file.

Now let's examine the structure which is defined by the above mentioned XML Document Type Definition file.

5.1 The structure of the descriptor

The name of root element is `model-family`. Its child elements are:

- `family-name`: the name of the model-family
- `model-class`: the class name of the Repast model
- `image`: path to the image that is assigned to the Repast model. The path is relative from the `resource` directory of pet archive.
- `description`: the description of the model
- `model-visualizations`: contains a list of `model-visualization` and `visualization-group` elements. The two element type is deeply discussed in later chapters.
- `agents`: contains a list of the `agent` element. The `agent` element is discussed deeply in chapter 5.1.3.

5.1.1 The model-visualization element

Each `model-visualization` element refers to a display of the Repast model. The element is empty but has five attributes:

- `index`: The index of the referenced display. The index of a display is defined by either the order of display elements in the list returned by the function `getDisplay()` of the `DisplayProvider` interface or the registration order of the simulation event listener list and media producer list in the `SimModelImpl` class. For more information read chapter 2.4.2.1.
- `display-name`: The display name of the visualization. If empty then the display name is retained from the display object. If the name of display object is empty then the name is derived from the reference index.
- `width`: The width of the visualization applet. If omitted then the width of the applet is set to the original width of the display.
- `height`: The height of the visualization applet. If omitted then the height of the applet is set to the original height of the display.
- `refresh-rate`: Defines the refresh rate of the applet in milliseconds.

Once a display of the model is referenced with its index it becomes available and the user can select it in PET at the model's properties page.

5.1.2 The visualization-group element

A `visualization-group` element defines a group of visualizations that should be displayed in one applet. This makes it possible to view multiply visualizations simultaneously or

almost simultaneously separated by tabs. A visualization group is considered as one visualization. Its attributes are:

- **id**: the identifier of the visualization. Must be unique.
- **refresh-rate**: The refresh rate of the visualization in milliseconds.
- **layout**: defines how to layout the members of the group. Is three possible values are:
 - **grid**: visualization are put side by side from left to right. In case the width of the resulted image exceeds the value of **max-width** attribute, the exceeding image part (a visualization) is put in the next row. This way, all the visualizations that are member of the given group are visible at the same time.
 - **tabs**: Each visualizations of the group are displayed on a different tab. Using the **tabs** value, users might see only one visualization at a time.
 - **both**: mixes the above mentioned two. The visualizations are put in a tab, but the image content of the last tab is equivalent with the content of a **grid** set group.
- **max-width**: it plays a role only when the layout attribute is set to **grid**. (Read the above)
- **display-name**: the text written above the visualization.

The **visualization-group** element has at least one or more child element named as **visualization-entry**. Each **visualization-entry** element defines a group member. Its content is empty and has four attributes:

- **index**: its meaning is the same as with **model-visualization** element.
- **name**: the display name of the visualization.
- **width**: the width of the visualization in pixels.
- **height**: the height of the visualization in pixels.

5.1.3 The agent element

Each **agent** element defines an agent type. The attributes of the element are:

- **class**: The name of class of the agent
- **name**: The reference name of this agent type

The child elements of the **agent** element are:

- **fields**: Contains a list of **field** elements. Each **field** element describes a property of the given agent type. The next chapter discusses the **field** element in detail.
- **field-defaults**: Enables to define default values for the properties of agents. For more information on this element read the chapter 5.1.3.2.
- **user-interface-mappings**: Describes which methods of this agent type are called by the repast scheduler and what are the methods that are appropriate as a user callable replacement of them. For more information read the chapter 5.1.3.3.
- **detectors**: Contains a list of **detector** elements. A **detector** element has empty body and a single **name** attribute. The name must be a function name of the agent. The function must have **not void** return value and must get no parameters.
- **image**: The path of image associated to this agent. The path is relative to the **WEB-INF** directory.
- **visualizations**: Enumerates the visualization definitions associated to this agent. It has two kind of child element:
 - **model-visualization-for-agent**: With the help of this element it is possible to associate a model display to an agent type. This element is empty but it has three attributes:
 - **index**: The reference index of a model display

- **display-name**: The display name of the visualization
 - **refresh-rate**: Defines the refresh rate of the applet in milliseconds
- **visualization**: Defines a custom visualization which is assigned to this agent type. For more information on this element consult the PET Tutorial documentation.
- **description**: The display name of the agent type

5.1.3.1 The field element

Each **field** element describes an agent property. It has two attributes:

- **name**: The name of the property
- **access**: Defines the permissions on this property. Possible values and their meanings are:
 - **r**: The property is read only
 - **rw**: The property has read/write access
 - **no**: The property is not visible for any agents of this type

The **field** element has the following child elements:

- **displayName**: The display name of this property
- **shortDescription**: A short description for this property
- **constants**: Contains a list of **constant** elements. A **constant** element defines a named value which is useful when we want to display human readable text instead of numbers or text tokens. A good example is when a property has 3 different states and the states are expressed with numbers. For example: 0=cold, 1=tepid, 2=hot. The element has two attributes:
 - **name**: The displayed text which should be human readable.
 - **value**: The assigned value.

5.1.3.2 The field-defaults element

This element is used for defining default value, visibility and initialized state for the properties of agents. In addition the user can define groups of agents that are created with the same property settings. The group defines the number of agents and the group settings override the defaults. The element has three kind of child element:

- **basic-agent-field-defaults**: This element is used for setting the default value of controllable and visible properties of the agents. It has only one occurrence. Its body is empty and has two attributes:
 - **controllable**: A Boolean value indicating whether the agent can be controlled
 - **visible**: A Boolean value indicating whether the agent is visible
- **agent-field-default**: This element is used for giving a default value to a custom property and to set up the visibility and initialized state of the property. It has four attributes:
 - **name**: The name of the referenced property
 - **visible**: A Boolean value indicating whether the property is visible by default
 - **initialized**: A Boolean value indicating whether the property is initialized by default. If a property is not initialized then it is possible to set its values in the Agent Properties page either one by one or in group. To know more about this function read the corresponding part of the PET Admin guide.
 - **value**: The default value for this property of this agent type
- **set**: The previous two elements (**basic-agent-field-defaults** and **agent-field-default**) defined a default value for all the agents of the given type. This element

does the same thing except that the number of affected agent is limited with the `count` attribute. This element has two child elements:

- `basic-agent-fields`: Equivalent with `basic-agent-field-defaults` but its scope is defined with its parent's `count` attribute.
- `agent-field`: Equivalent with `agent-field-defaults` but its scope is defined with its parent's `count` attribute.

5.1.3.3 The user-interface-mappings element

This element contains a list of `scheduledMethod` elements. For understanding the concept of a scheduled method read chapter 2.4.5. A `scheduledMethod` element identifies a scheduled method by its `name` attribute. Each of these elements defines a function group which has at least one or more alternate methods and an optional timeout method. The `scheduledMethod` element has four attributes:

- `name`: The name of the scheduled method
- `visible`: A Boolean value indicating whether this scheduled method is displayed on the screen as a Function group
- `displayName`: A human readable name of the defined function group
- `shortDescription`: A short description for the defined function group

As described earlier it is possible to identify alternate methods and an optional timeout method for each function group. It is done with the child elements of the `scheduledMethod` element as follows:

- `timeoutMethod`: This element has a single `name` attribute which identifies the method that should be called on timeout events. To learn more about timeout events consult the PET Admin guide documentation.
- `humanCalledMethod`: This element identifies a replacement method for the scheduled method. This replacement method is only invoked by the controller user by issuing orders through the Web. It has three attributes:
 - `name`: The name of the replacement method
 - `displayName`: A human readable name of the replacement method. This value is displayed on the screen for the user.
 - `shortDescription`: a short description. Appears on the screen as a popup hint