

# Dynamic Information Architecture System (DIAS): Developer's Guide

---

Decision and Information Sciences Division  
Argonne National Laboratory



ARGONNE IS OPERATED BY THE UNIVERSITY OF CHICAGO FOR THE U.S. DEPARTMENT OF ENERGY OFFICE OF SCIENCE



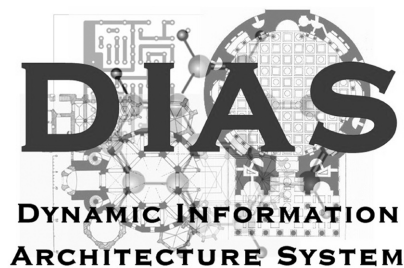
# Dynamic Information Architecture System (DIAS): Developer's Guide

---

by  
Kathy Lee Simunich

Modeling, Simulation and Visualization Group  
Decision and Information Sciences Division  
Argonne National Laboratory

February 2005



**About Argonne National Laboratory**

Argonne is operated by The University of Chicago for the U.S. Department of Energy Office of Science, under contract W-31-109-Eng-38. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see [www.anl.gov](http://www.anl.gov).

**Availability of This Report**

This report is available, at no cost, at <http://www.osti.gov/bridge>. It is also available on paper to U.S. Department of Energy and its contractors, for a processing fee, from:

U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831-0062  
phone (865) 576-8401  
fax (865) 576-5728  
[reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)

**Disclaimer**

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor The University of Chicago, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, Argonne National Laboratory, or The University of Chicago.

# Contents

Abstract .....	1
1 Introduction .....	1
1.2 DIAS Simulation Architecture .....	4
1.3 How to Use This Manual .....	7
2 DIAS Coding Fundamentals .....	7
2.1 Creating New Objects .....	8
2.1.1 Main DIAS Framework Classes .....	9
2.1.2 Creating Entity Subclasses .....	13
2.1.3 Creating Process Subclasses .....	13
2.1.4 Creating Model Subclasses .....	14
2.1.5 Creating ModelState Subclasses (optional) .....	14
2.1.6 Creating the ApplicationData Subclass .....	15
2.1.7 Creating the ContextBuilder Subclass .....	15
2.1.8 Creating the SimulationParameters Subclass .....	15
2.1.9 Creating the SimulationServerImpl Subclass .....	16
2.1.10 Creating the Simulation Client .....	16
2.2 Coding Simulation Statics .....	16
2.2.1 Entity Subclasses .....	16
2.2.2 Process Subclasses .....	17
2.2.3 The SimulationServerImpl Subclass .....	19
2.2.4 Context Builder .....	19
2.3 Coding Simulation Dynamics .....	22
2.3.1 Internal Models .....	22
2.3.2 External Models .....	25
3 The Inner Workings of the DIAS Framework .....	35
3.1 Context Execution .....	35
3.2 How to Create Entities in a Frame .....	35
3.3 The Simulation Manager and Scheduling Events .....	37
3.4 Event Priorities and Simulation Utilities .....	38
3.5 How Processes Execute .....	38
3.6 Start Simulation Processing .....	39
3.7 Simulation Execution .....	43
3.8 Running the Client .....	45
3.9 Controlling Execution with Simulation Manager Window .....	48
3.10 Distributed Execution Flags .....	50
4 References .....	52

## Contents (Cont.)

Appendix A: Farm Tax DIAS Example.....	53
Appendix B: DIAS Exceptions.....	61

## Figures

1 Schematic of the DIAS Architecture.....	2
2 Fully Distributed DIAS Execution Architecture.....	6
3 DIAS Execution within a Single JVM.....	6
4 Entity - Aspect - Process - Model Classes .....	8
5 Example Atmosphere Entity with Possible SDS Parameter Representation .....	12
6 Flow Chart Showing Context Build Process.....	20
7 Step Templates Defining a Course of Action Model .....	30
8 Schematic of the WithdrawCashFromATM COA Model.....	32
9 Context Execution Steps .....	36
10 Flow Chart for the Attempt to Execute a Process .....	40
11 Flow Chart for Process Execution.....	41
12 Flow Chart for the Simulation Initialization Process.....	42
13 Flow Chart of Simulation Execution Processing .....	44
14 How a Client Initiates a Simulation Run.....	46
15 Initialize and Run Logic Without a Client GUI.....	47
16 Simulation Manager Window .....	48
17 Logic for Execution of the Simulation via the Simulation Manager Window.....	49
18 Logic During Client Release of the Simulation Server.....	50

## Figures (Cont.)

A.1 Farm Tax Execution Mode.....	54
A.2 Event Sequence for Farm Tax Demonstration Program .....	58

## Tables

1 Distributed Execution Flag Settings.....	51
A.1 Farm Tax E-A-P-M Design.....	55
A.2 Process/Model Input/Output Parameters.....	56
A.3 Event Data Map.....	57
A.4 Entity/Process Event Map .....	57



# Dynamic Information Architecture System (DIAS): Developer's Guide

Kathy Lee Simunich

## Abstract

The Dynamic Information Architecture System (DIAS) is an object-oriented framework designed specifically to provide a structure for complex modeling and simulation problems. By adherence to two key tenets of the object programming paradigm, encapsulation and inheritance, DIAS (1) provides a clean, modular design because objects manage their own attributes and dynamic behaviors and (2) promotes code re-use and extensibility because object subclasses can "inherit" attributes and behaviors from parent object classes. This developer's manual provides an overview of the DIAS conceptual structure, basic simulation guidance, and step-by-step instructions for selected simulation functions.

## 1 Introduction

A major challenge in modeling and simulation is the need to account for the complex, dynamic, and multi-scale nature of "real-world" systems for the purpose of experimentation, problem solving, and decision support. In response to this challenge, the Decision and Information Sciences Division (DIS) of Argonne National Laboratory (ANL) has developed the Dynamic Information Architecture System (DIAS), an object-oriented framework designed specifically to provide tools for complex modeling and simulation problems.

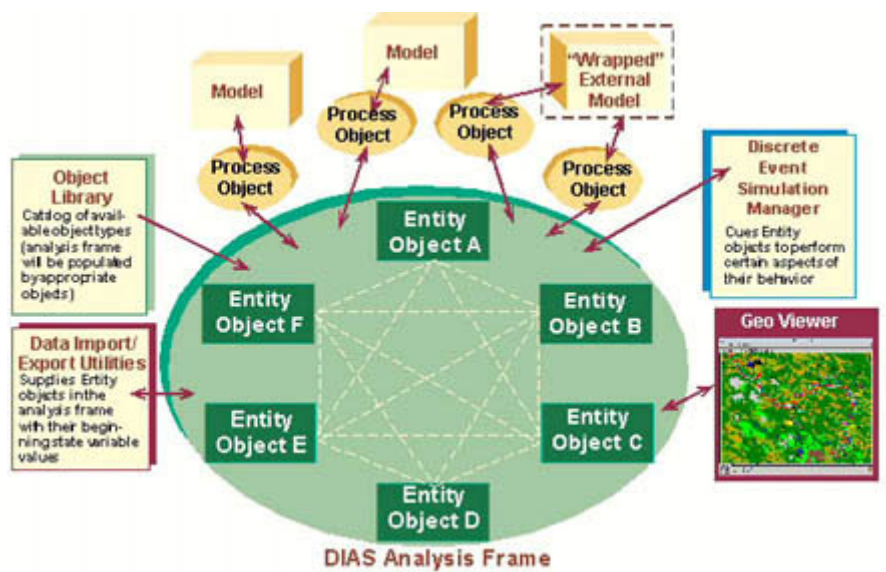
The flexible DIAS software infrastructure offers the ability to:

- Address a complex problem by allowing many disparate multidisciplinary simulation models and other applications to work together harmoniously within a common framework;
- Integrate existing ("legacy") models without extensive reworking, thus capitalizing upon previous investments in models and applications;
- Provide an integrated architecture that allows for the dynamics of real-world systems — in DIAS, thousands of objects can interact via dozens to hundreds of concurrent processes;
- Implement new DIAS models, including models of complex, cooperative behaviors of agents (e.g., persons and organizations);

- Encourage the development of object libraries that contain a large number of reusable objects to represent a wide variety of real-world elements and therefore reduce the long-term cost of redeveloping objects and technologies;
- Support software applications that can operate at multiple spatial and temporal scales;
- Incorporate new data, concepts, and technologies that will bring the best available knowledge, science, and technology to bear on decision-making processes; and
- Operate in a distributed environment where applications can be linked across multiple machines via computer networks.

## 1.1 DIAS Objects

The modeling domain of DIAS is flexible and is determined by available objects and processes within DIAS and by the suite of models and other data processing applications that are available to the developer. The object-oriented infrastructure classes of DIAS are the key to the flexibility and dynamics of the system. Figure 1 provides an overview of the DIAS architecture.



**Figure 1 Schematic of the DIAS Architecture**

The infrastructure of DIAS consists of the following components:

**Analysis Frame:** This DIAS object represents the area of interest in a simulation, holds all the “playing pieces,” and defines the purpose of a simulation. The analysis frame defines the geographic or conceptual bounds of the simulation.

**Entity Object:** The entity objects represent the real-world components (e.g. building, person, atmosphere, etc.) that interact in the simulation. Each entity object has a number of parameter and aspect objects associated with it. The parameter object contains the static properties of the entity object, and the aspect object describes the behavior of the entity object and how it interacts with other objects.

**Object Library:** The object library contains a permanent, shared collection of entity objects of various types, to be drawn upon during simulations. The library is continually growing, as new applications add objects and developers modify existing objects.

**Data Import/Export Utilities:** These data-ingestion utilities have been developed to supply the object variables (parameters and aspects) from a variety of external data sources.

**Model Wrapper:** The model wrapper provides the mechanism for linking models and applications to DIAS. The model wrapper consists of two components, the model object and the model controller. The model object has references to all necessary process objects, and the model controller provides the explicit linkage to external model/application source codes and data structures external to the DIAS framework.

**Process Object:** The process object provides the means of addressing a particular entity object behavior. The process object is the one object in DIAS that holds information about the entity objects *and* the specific requirements of the corresponding model. It is also responsible for passing data to and from the model.

**Simulation Manager:** The discrete event simulation manager processes events in a time-ordered queue. Events are created principally by entity behavior and user interactions.

**JeoViewer:** The JeoViewer is an object-oriented geographic information system (GIS) module that provides displays of object positions and states. It provides the capability to query, view, and manipulate objects within an analysis frame; it can be used to enter and update data and as a debugging tool during model development.

**Context Object:** The context object is responsible for specifically laying out the context of the simulation. It links entity object behavior (aspects) with associated process objects (and specifies dependencies between them) and contains the temporal extent of the simulation.

**Spatial Data Set:** DIAS spatial data set (SDS) objects contain the complete geometric specification for a one-dimensional, two-dimensional, or three-dimensional spatial partitioning. A SDS is not a DIAS entity object, but is used as a entity parameter that can extend the entity object's attribute specifications to include spatial dependencies. The SDS provides the developer with a cost-effective method of translating and/or partitioning data as required by the different modules in a DIAS simulation.

**DIAS Models/Applications:** Existing models and applications written in virtually any software language can be used in DIAS via a formal registration process. New models can also be directly written within DIAS.

DIAS embraces and extends key software engineering tenets of the object paradigm: encapsulation and inheritance. Encapsulation promotes clean, modular design because objects manage their own attributes and dynamic behaviors. Inheritance promotes code re-use and extensibility because object subclasses can "inherit" attributes and behaviors from parent object classes.

DIAS extends the object paradigm by abstraction of the object's dynamic behaviors, separating the "what" from the "how." DIAS object class definitions contain an abstract description of the various aspects of the object's behavior (*what*), but no implementation details (*how*). Separate models and applications carry the implementation of object behaviors. These models are linked to appropriate domain objects as needed for specific simulation contexts.

In DIAS, models communicate only with domain (entity) objects, never directly with each other. From a software perspective, this makes it easy to add models, swap alternative models in and out without re-coding, and scale up to increasingly complex simulations.

## 1.2 DIAS Simulation Architecture

The Dynamic Information Architecture System (DIAS) is an object-oriented simulation framework to allow the seamless integration of models. Models may be written within the framework (in the Java language), or they may be external models, written in another language. The model controller is used to "wrap" an external model to be executed as a remote procedure call by the simulation in progress. External models that make good candidates for wrapping in this manner are ones that have a separable GUI and do not require user interaction during a run. Any number of external models may be used within a DIAS simulation, where the DIAS framework is used as the integrating architecture for executing the external as well as any number of internal models as a coherent overall simulation.

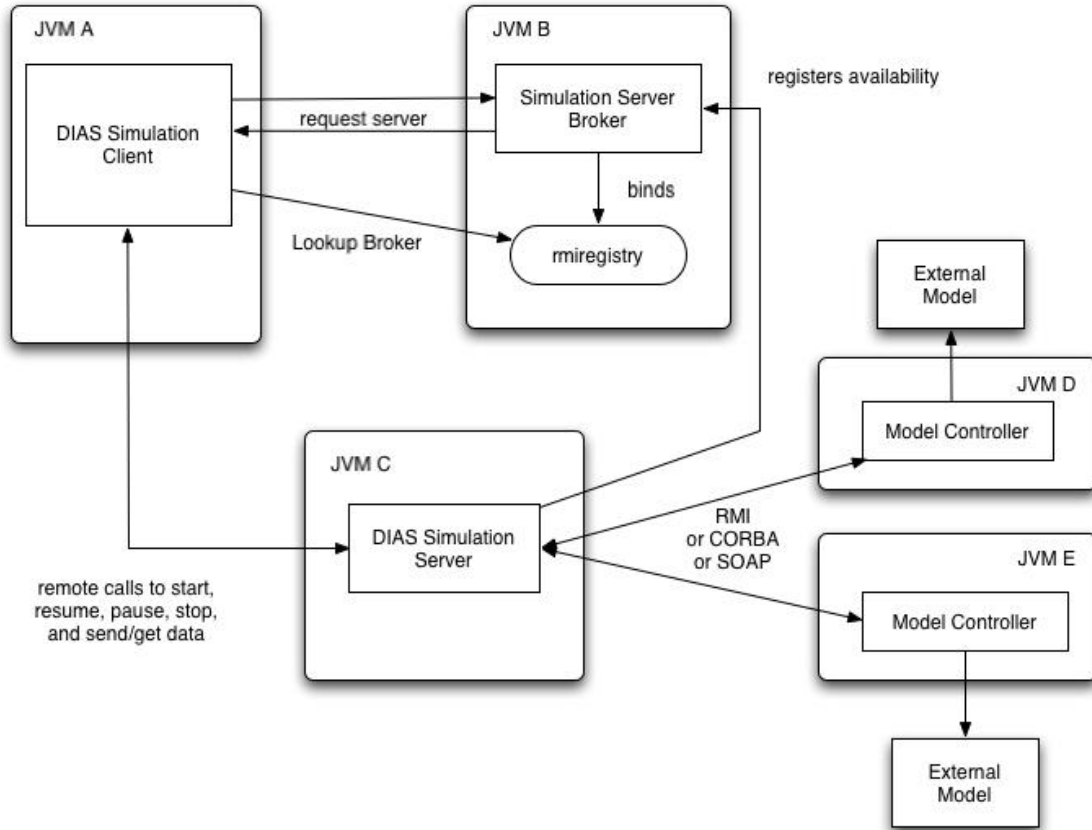
The main components of a DIAS simulation are the:

- Simulation Client – used to set up a simulation scenario, control the execution of the simulation (start, stop, pause, resume), and to optionally interface with the user.
- Simulation Server – executes a scenario using a discrete event manager to advance simulation time. Responsible for executing any internal or external models. Execution may be started, stopped, paused, and resumed.
- Server Broker – a singleton instance, used as an intermediary for pairing up clients to servers for execution.
- Model Controller – used as a bridge from a DIAS simulation to an external model (optional, only needed if incorporating external models).

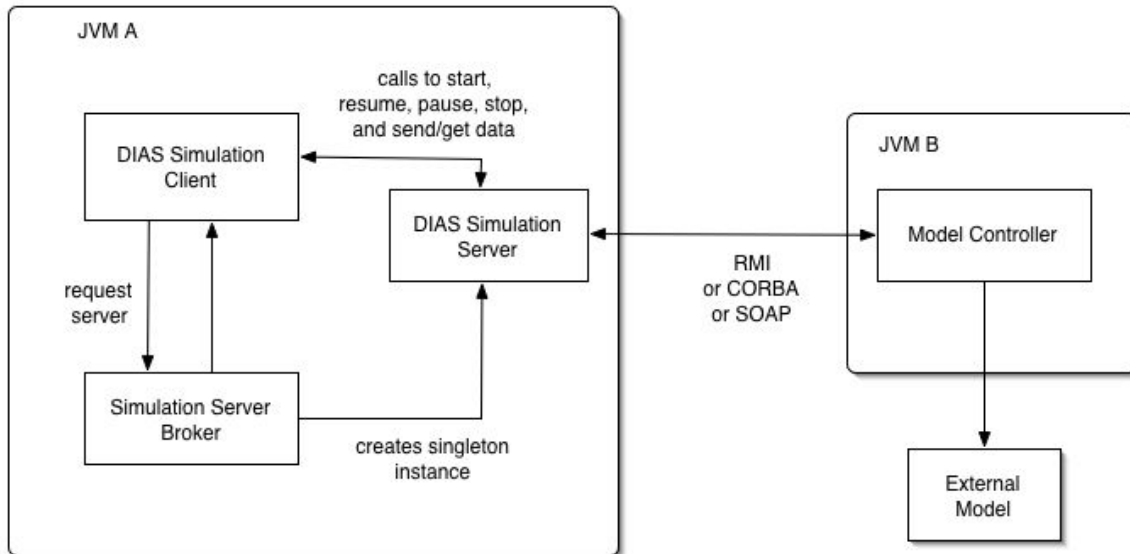
The analysis framework is set up as a client/server application. External models may run on the same Java virtual machine (JVM) as the simulation or they may run on another machine connected over the network. Either way, DIAS communicates with the external models using Java's remote method invocation (RMI), which is the default method, although the Common Object Request Broker Architecture (CORBA) or Simple Object Access Protocol (SOAP) may be used as well.

Figure 2 shows a fully distributed DIAS simulation setup. The single instance of the server broker is started up in its own JVM. JVM processes can run on a single machine or separate machines connected by a network. The server broker binds itself into an RMI registry and waits. Any number of simulation servers may be started, each one in its own JVM. Each simulation server will register itself with the broker, export itself to an RMI registry, and then wait. Any number of clients may be started, each in its own JVM as well. The client will look up the broker in the registry and request a simulation server. The broker will select a server from its list and pass the reference on to the client, where the client and server will communicate directly for setup and execution of a simulation. The broker resumes listening for more client requests. It will take each request in order and match it up with an available server. If no server is available, the requests are queued until one becomes available. A simulation server may call external models as part of the simulation. The external models' model controller needs to be started within its own JVM in which it will wait for calls from the server to execute the external model.

Figure 3 shows the same components, with most running within the same JVM. The basic processing is the same: the client requests a server from the broker, which returns the singleton server to the client. When the broker, client, and server components are running in the same JVM, RMI communication is bypassed and method calls are made directly between the objects. If there are external models to be run, they would still be executed by remote model controllers, and called by the DIAS simulation server component (a single model controller call is illustrated). See Section 3.9 (p. 50) for code examples on how to set up the execution mode of a DIAS simulation.



**Figure 2 Fully Distributed DIAS Execution Architecture**



**Figure 3 DIAS Execution within a Single JVM**

### 1.3 How to Use This Manual

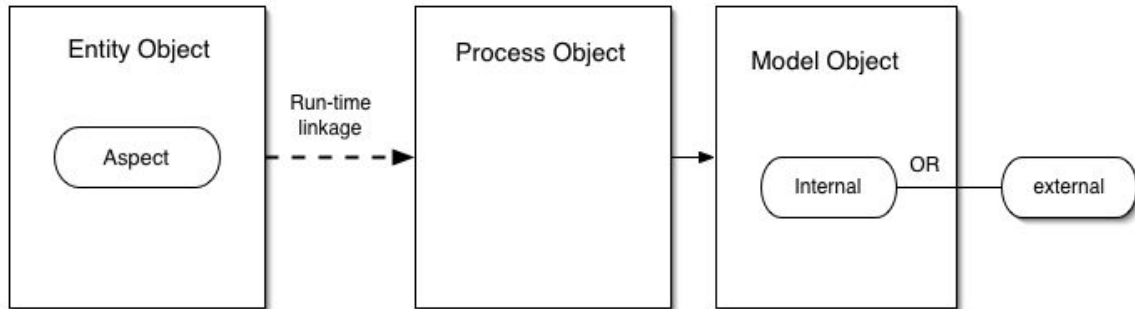
This document is written for the developer who will be coding within DIAS. A more general overview of the DIAS concepts is available that presents the elements in a non-programmer view (Simunich 2002). Distributed with this document is a fully functional demonstration program called the Farm Tax simulation. The code examples in this document come from the demonstration program. You may find it helpful to have the example code nearby as a reference while reading this manual. Where appropriate, the fully qualified class and method names are given, to facilitate ease of lookup. The demonstration code is both documented within the code and also in the javadoc format. There is a readme file and run scripts included. Appendix A contains additional discussion of the example.

This manual uses some simple typography conventions to ease readability.

- A **sans-serif font** is used for Java object classes and methods. Consistent with coding conventions, class names start with a capital letter and method names start with a lowercase letter. In both cases, subsequent words in the name are capitalized and concatenated (no spaces).
- A *sans-serif italic font* is used for parameter values (*true*, *false*, etc.).
- A `monospaced font` is used for code, which is always displayed on separate lines. This convention is also used for the DIAS exception messages in Appendix B.

## 2 DIAS Coding Fundamentals

The fundamental architectural feature of the DIAS framework is the entity – aspect – process – model (E-A-P-M) connection (see Figure 4). This abstraction of layering is the key to creating simulations that can truly provide “plug and play” implementation of behaviors (processes and models) to corresponding domain objects (entities and aspects). To design an application for the DIAS framework, you need to map your domain objects (e.g., an atmosphere object) to subclasses of the DIAS entity class, define their behaviors with DIAS aspects, and implement each behavior through DIAS Process connections to a corresponding DIAS model for execution. Set up in this way, models never call other models, but always just read and write attributes within the entities. This allows a new or different implementation of a behavior to be switched out to create different types of simulations using the same domain entities. The linkage to the specific process and model that will be used to execute the entity’s aspect for a particular run of a simulation is determined at runtime and is referred to as the simulation context.



**Figure 4 Entity - Aspect - Process - Model Classes**

The following section describes the DIAS framework coding guidelines for creating an application within DIAS. Appendix A describes a DIAS application created as a tutorial example and is used to show concrete code within this manual.

Implementation of an application consists of three basic steps when creating a DIAS simulation:

1. Creating New Objects – Editing existing classes, creating new domain objects, and subclassing the DIAS framework classes.
2. Coding Simulation Statics – Overriding methods in the subclasses that set up the objects to be used within the DIAS framework. This includes static initialization of all entity and process subclasses, as well as methods in the context builder.
3. Coding Simulation Dynamics – Overriding methods in the subclasses that execute during the simulation (context execution). Writing discrete-event handlers to progress the simulation through time, and scheduling events to keep execution flowing. Writing internal models and connecting external models through their model controllers is where the domain-specific part of the application exists.

## 2.1 Creating New Objects

In order to create a DIAS simulation, various framework objects need to be subclassed and/or interfaces implemented. These include any number of entity objects, process and model objects, optional `ModelState` objects and singleton instances of `ApplicationData`, `ContextBuilder`, `SimulationParameters`, `SimulationClient`, and `SimulationServer`. Any number of regular Java classes may be created to implement the entire domain of an application.

## 2.1.1 Main DIAS Framework Classes

### 2.1.1.1 ParameterizedObject

The `anl.dias.core.ParameterizedObject` is the abstract superclass for all classes that require parameters. Parameters are dynamic data holders and represent attributes of a class. Instead of explicitly declaring Java classes or primitives for a class, parameters may be used to hold attribute information for a class that extends the `ParameterizedObject` superclass. This allows the developer to generically define attributes and change their type representation. A parameter object is able to store data (any Java object or an `anl.dias.core.SpatialDataSet`), be input to multiple process objects, have historical keys associated with it (e.g., time-based values), and be annotated (e.g., string description relating data to a geographical region).

Each parameter holds an associated meta parameter, which holds the name and units of the parameter. This is a space-saving measure in that all instances of a given parameter have a reference to the same instance of meta parameter. The developer does not need to manipulate the meta parameter or parameter class itself, but can get and set their values through accessor methods within `ParameterizedObject`. Use the methods `ParameterizedObject.addMetaParameter` and `getValue` and `setValue` to get and set the parameter data.

Parameter existence exceptions that can be thrown include:

- `NoSuchParameterException` – thrown when the requested parameter does not exist (i.e., the `ParameterizedObject` subclass does not specify it as a meta parameter) and
- `ParameterNotInstantiatedException` – thrown when the requested parameter has not been instantiated (i.e., the `ParameterizedObject` subclass specifies it as a meta parameter, but it has not been instantiated at the time of access).

Appendix B provides examples and explanations of other exceptions.

Parameters may be annotated using any Java object. An `anl.dias.core.Annotation` is basically used as a secondary key for referencing data within a parameter. If you use the `setValue` method that takes an annotation object, the data will be set for the parameter and annotation pair. You will then always have to use the `getValue` method that specifies the annotation in order to access it afterward. The method `ParameterizedObject.annotateParameter` can be used to set an annotation on an existing parameter. One example of an annotation is to reference data with a geographical region. The parameter is named as usual, and the annotation is a geographic region instance. In fact, the class `anl.dias.core.GeographicAnnotation` can be used for this type of annotation for any parameter that needs a name and `anl.spatial.geometry.Geometry` object.

The method `ParameterizedObject.getAnnotations` returns an `anl.util.system.ReadOnlyIterator` that iterates through the collection of annotations for a parameter.

Parameter annotation exceptions that can be thrown with regards to annotations include:

- `AnnotationAlreadyExistsException` – thrown when an attempt is made to duplicate an annotation that a parameter already has,
- `AnnotationNotFoundException` – thrown when an annotation is not found for a parameter,
- `AnnotationRequiredException` – thrown when an annotation is required but had not been specified, and
- `ParameterNotAnnotatedException` – thrown when an annotated parameter is expected, but the parameter is not annotated.

Parameters may also have a historical key associated with them. Historical keys can be any Java object, but the object needs to implement the `anl.dias.core.HistoricalKeySource` interface. The provided class `anl.dias.core.SimulationTimeKeySource` can be used to store data associated with the current simulation time (when `SimulationTimeKeySource.currentHistoricalKey` is called). This is a useful class to use if you have time-based data you want to keep track of under the same parameter name. The following code can be found in `anl.csiro.farntax.simulation.FarmTaxContextBuilder.createInitialEntities` in the demonstration example. It sets up a `simulationTimeKeySource` for tracking the annual rainfall throughout the simulation:

```
/** Time Key Source for the Simulation */
private SimulationTimeKeySource simTimeSource;
atmosphere.setHistoricalKeySource("annualRainfall",
simTimeSource);
```

The methods `ParameterizedObject.setHistoricalValue` and `getHistoricalValue` could be used to set or get parameter values associated with a historical key. These parameters may also have an annotation as well. With the `get/setHistoricalValue` methods, the historical key must be sent in as a method parameter each time. Alternatively, the method `setHistoricalKeySource` may be called once on a parameterized object to set the `HistoricalKeySource` for a particular parameter. After that, the normal `getValue/setValue` can be called to access the parameter value. The framework takes care of calling the `currentHistoricalKey` method on the `HistoricalKeySource` to access the current data for the requested parameter.

The method `ParameterizedObject.getHistoricalKeys` returns an `anl.util.system.ReadOnlyIterator` that iterates through the collection of `HistoricalKeySources` for a parameter.

### 2.1.1.2 SpatialDataSet (SDS)

The `anl.dias.core.SpatialDataSet` is the abstract superclass for all datasets that associate data structures with specific cells or points in two or three dimensions. The parameters within an SDS share the same topology/coordinate system specified within the SDS. Cell-based and point-based object classes are subclasses of the abstract SDS class. Depending on the type of data that are to be represented spatially, a cell or point two-dimensional or three-dimensional SDS is available:

- **Point2DCollection** – data at discrete point locations, for example, well boring data.
- **Point2D (or 3D) QuadGrid** – to represent continuous variables with values at discrete intervals and interpolated between grid points, for example, atmospheric temperature and ocean currents.
- **Cell2DCollection** – data associated with variably shaped polygon cells and valid anywhere within the cell, for example, census data.

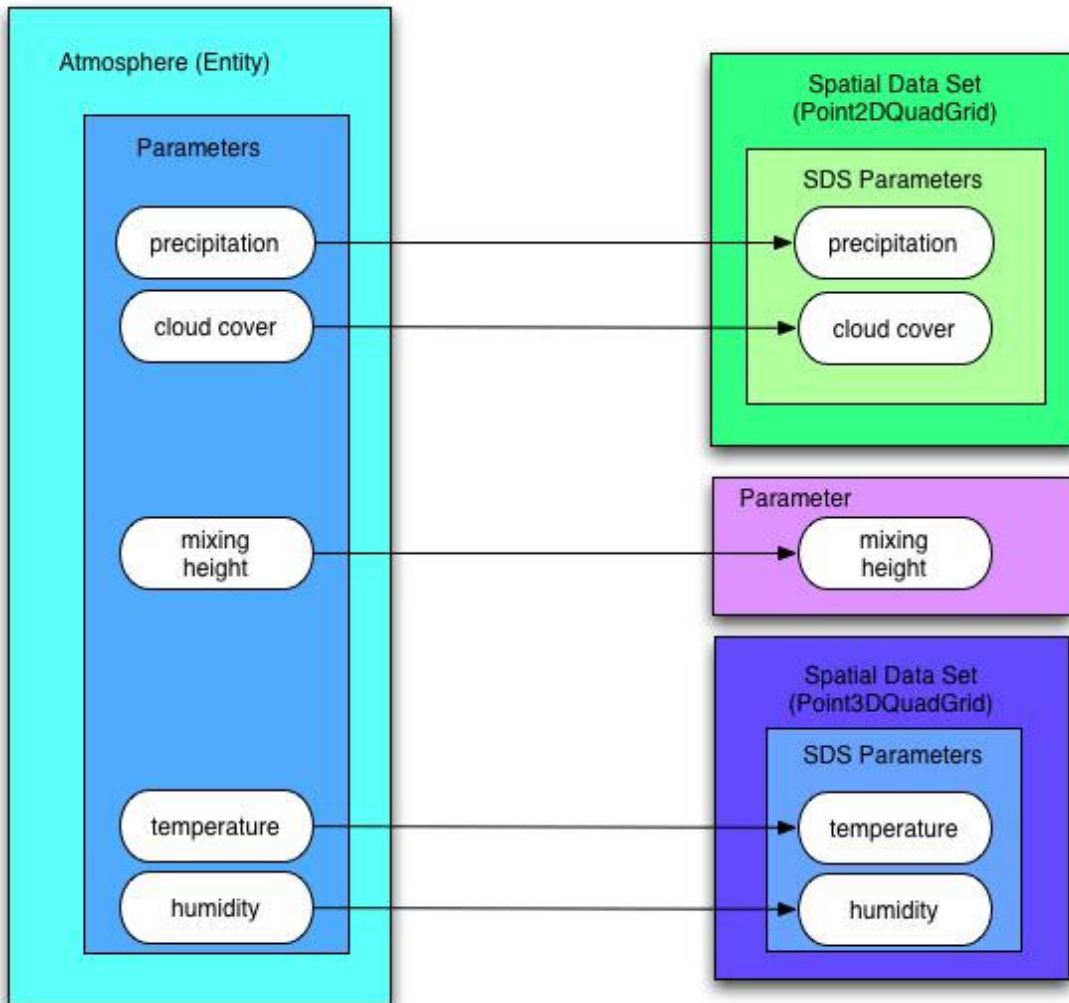
Figure 5 shows an example atmosphere entity and its parameters. The 2D parameters, precipitation and cloud cover, are to be represented by a **Point2DQuadGrid** SDS. The 3D parameters, temperature and humidity, are represented by a **Point3DQuadGrid** SDS. The mixing height parameter is a single value and therefore may be represented as a regular parameter object. To the atmosphere entity, these are all just parameters, and their representations are a mix of parameters and SDSs.

SDS attributes include a topology (e.g. **2DQuadGrid**), a coordinate system, an interpolator (algorithm used to interpolate values), and parameters. In Figure 5, the precipitation and cloud cover parameters share the same topology and coordinate system (by the fact that they reside within the same SDS container). The temperature and humidity parameters share a 3D topology and coordinate system.

The accessor method for an SDS needs to specify the location and coordinate system for the parameter needed and returns a (possibly interpolated) value. The `getCurrentValue` method is shown below (where the interpolator value is optional in the method call):

```
anSDS.getCurrentValue(SDSPParameter p, Location l,
    CoordinateSystem s, Interpolator i);
```

The accessor automatically invokes a coordinate transformation if the specified coordinate system does not match that of the SDS holding the parameter. There is a default interpolator for each SDS, but it can be overridden with the above call. If it is left out of the call, the default interpolator is used.



**Figure 5 Example Atmosphere Entity with Possible SDS Parameter Representation**

Interpolator examples include:

- Calculate a bilinear interpolation value from the gridded data at the specified location and
- Calculate value at the nearest gridded data point to the specified location.

There is a corresponding `SDSParameter` and `SDSMetaParameter` (which extends the parameter and meta parameter, respectively). These classes are used when you call `ParameterizedObject.addSDSMetaParameter` and `ParameterizedObject.associateSDSToParameter` (or `reassociateSDSToParameter` to change the associated

SDS), and they do not need to be manipulated by the developer directly. `SDSParameter` may also be annotated and have a historical key as well, since it extends a parameter.

The methods `ParameterizedObject.getCoordinateSystem` and `getSDSTopology` will return the coordinate system and topology, respectively, for the SDS associated with an `SDSParameter`.

## 2.1.2 Creating Entity Subclasses

Any domain objects that need to have models attached to them and need parameters should extend `anl.dias.core.Entity`, which extends `ParameterizedObject` and therefore may use parameters to hold attribute data.

Aspects are an abstract class and are owned by entities. Aspects abstractly represent behaviors of the entity and are used within DIAS to connect a process and model to model the implementation of that behavior of an entity.

Entities represent domain objects and hold meta aspects and meta parameters that define the possible behavior and attributes, respectively, that the entity may model. By capturing overall parameters and aspects of an entity, the entity subclass may be reused in many different types of simulations. By connecting different processes and models that implement aspects in different ways, modelers may run multiple simulations to explore differences in results in a “plug and play” manner.

All Entities are held in an `anl.dias.simulation.Frame` during the simulation. The analysis frame is just a global container class that holds the collection of entities, possibly within a geographic region. The `anl.dias.simulation.RegionDefinition` class is a placeholder class intended to define the geographical region for a frame. Currently, it just holds a name and description used to define an overall scenario. Below is an example instantiation from the Farm Tax demonstration code in `anl.csiro.farntax.simulation.FarmTaxSimServerImpl`.  
`buildContext:`

```
RegionDefinition rd = new
RegionDefinition(simParams.getScenarioName(),
simParams.getScenarioDescription());
```

## 2.1.3 Creating Process Subclasses

The class `anl.dias.core.Process` needs to be extended in order to be associated with an entity’s aspect of behavior. This class extends `ParameterizedObject` and therefore may use parameters to hold any needed attribute data.

Processes are always associated with a model. The process object is used to map data from the entity point of view to the model point of view. It also allows you to code any data translation needed for the model to run.

The process class assembles the needed input data for a model from its associated entity (or from any other entities) in the analysis frame, calls the appropriate model method, and unpackages the output data from the model and sets the appropriate parameters in the entity (or entities).

#### 2.1.4 Creating Model Subclasses

The class `anl.dias.core.Model` is the actual implementation of an aspect of behavior. The reason process and model are separated is that models may be internal or external to DIAS. Also, a single model may be called by more than one process if it implements multiple aspects.

Internal models are written in Java and extend `anl.dias.core.Model`. External models extend `anl.dias.core.rmi.RMIExternalModel` and must have a model controller associated with them. RMI is the preferred method of communicating between DIAS and an external model controller, but DIAS has optional `CORBAExternalModel` and `SOAPExternalModel` implementations as well.

A special type of internal model is the `CourseOfActionModel`. This type of model is used to model extended transactions between entities, using the Framework for Addressing Cooperative Extended Transactions (FACET) (Christansen 2000), another ANL framework that is integrated within the DIAS framework. See Section 2.3.3 (p. 30) for an in-depth description of this type of model.

#### 2.1.5 Creating ModelState Subclasses (optional)

The `anl.dias.core.ModelState` serves as a common data block containing all data that may be used for a model for an entity of a particular type. `ModelState` instances are specific to individual instances of an entity, whereas the model is general to all instances of entity. In other words, each entity has its own set of data to be used for a model, but all entities that call the same model point to that one instance of model. Therefore, the executing process for an entity will assemble the needed input data and store it in the `ModelState`, which is passed to the model so that it may execute for that instance of the entity (data). Conversely, the output is stored in the `ModelState` and passed back to the process, where the process unpackages it and sets the appropriate parameters in the entity.

A model may instantiate the `anl.dias.core.ModelState` class if all you need is this passing mechanism. If, however, you need something more long-lived, you can create your own subclass of `ModelState` and add any attributes you would need (keeping in mind that the `ModelState` instance is reused for each call between the associated process and model, and only the input and output parameters are specific to an instance of the entity). An example of creating

your own subclass is perhaps the need to generate random numbers, where you want the same number stream used for all calls of a model.

### **2.1.6 Creating the `ApplicationData` Subclass**

Each DIAS simulation has a subclass of `anl.dias.simulation.ApplicationData`, which is used during execution and usually holds global data for the simulation entities. It is often used to start the scheduling of events to keep the simulation flowing, but it is not required to do so. It is a convenient place to write any type of output at the end of the simulation as well as initializing the `JeoViewer`, if the `JeoViewer` is being used in the application. The `JeoViewer` (Lurie 2002) is an object-oriented GIS that can show the spatial state of a DIAS simulation. The `JeoViewer` should be initialized from the `ApplicationData` subclass if you want to watch the simulation output on the `JeoViewer` screen while it executes. If you do this, the `JeoViewer` GUI will be executing on the server side (something you need to consider if you are running in distributed mode across two machines because the client may be showing a GUI, and the server will be showing the `JeoViewer`, perhaps on another machine).

### **2.1.7 Creating the `ContextBuilder` Subclass**

The `anl.dias.simulation.ContextBuilder` is the DIAS object that instantiates a simulation scenario for execution. It creates the analysis frame, instantiates all of the initial entities in that frame, makes the E-A-P-M connections, instantiates the parameters that will be used for the simulation, and initializes them to their defaults. A simulation server will instantiate a `ContextBuilder` and execute the resulting context when the client tells the server to start execution. Using the `ContextBuilder` approach allows multiple types of simulations to be run using the same set of “playing pieces” (i.e., entities, parameters, processes, models). By setting up different connections (e.g., running different models that represent an entity’s aspects) at run-time, different contexts can be executed and their results compared.

### **2.1.8 Creating the `SimulationParameters` Subclass**

The `anl.dias.simulation.input.SimulationParameters` subclass holds needed information on setting up a simulation run. Every simulation needs a start and stop time, scenario input and output directories, and, if it is using the `JeoViewer`, a geographic data input directory. The `SimulationParameters` class is passed from the client to the server and is used to hold all the scenario information needed to build and execute a simulation. Often, the client side of a simulation has a GUI, or may read in files, to set up scenario parameters. When that type of information is stored in the `SimulationParameters` subclass, it is passed from the client to the server automatically, where it can be used to build the context.

### 2.1.9 Creating the SimulationServerImpl Subclass

Every DIAS application needs to subclass `anl.dias.simulation.distributed.SimulationServerImpl`. This is the heart of the simulation, where the context is built, execution occurs via the event manager thread, and the simulation is terminated. The `SimulationServer` is requested by a `SimulationClient` (through the broker), and its execution is controlled by the client.

### 2.1.10 Creating the Simulation Client

The last object that needs to be subclassed is `anl.dias.simulation.distributed.AbstractSimulationClient`. The application inherits the request for a server, the RMI communication (if running remotely), and the optional use of a simulation manager window to allow the user to start, stop, pause, and resume the simulation server. The client is designed to run without a GUI, with only the simulation manager window as the GUI, or with an application-specific GUI. Subclasses of `AbstractSimulationClient` are responsible for implementing application-specific GUIs.

## 2.2 Coding Simulation Statics

Simulation statics comprise the instantiation and static initialization of the “playing pieces” in the simulation, that is, the entity subclasses, process and model subclasses, and the context builder.

### 2.2.1 Entity Subclasses

All entity subclasses must implement a static block and two static methods. The static block registers the subclass with the entity superclass (by calling `initializeEntityMetaData` and passing in the entity subclass). The static methods `initializeMetaAspects` and `initializeMetaParameters` are where the meta aspects and meta parameters are defined for the subclass. These methods are called by `initializeEntityMetaData` and an `anl.util.system.MissingRequiredMethodException` will be thrown if they do not exist in the subclass. See the example code below:

```
public class Atmosphere extends Entity {
    static {
        initializeEntityMetaData(Atmosphere.class);
    }
    public static void initializeMetaAspects()
    {
        Class cls = Atmosphere.class;
        addMetaAspect(cls, "precipitate", "Precipitate");
    }
}
```



Meta input parameters and meta output parameters are defined for a process along with a parameter-ready method name (for input parameters). When processes are chosen for a particular context (see `ContextBuilder.getModelsForRegion`), the meta input and output parameters are used by the DIAS framework to instantiate parameters of entities during context build time. Thus, this approach defines only the parameters needed for the specific simulation run.

The parameter-ready method is used when a process is executed to perform the behavior specified in an entity's aspect. Before the process can start, the parameter-ready method is checked for each process input. If the method returns *true* for each parameter checked, the process can then start execution. The framework keeps track of the parameter-ready status for all the parameters and rechecks each time a parameter value is changed within the simulation. Once all parameters are ready to go, the process is scheduled to execute. There are three pre-defined methods available from the process class:

- `parameterReadyIfNotNull` – returns *true* if the parameter value is not null
- `parameterReadyIfNull` – returns *true* if the parameter value is null
- `parameterAlwaysReady` – always returns *true*

You can use one of the above methods as the parameter ready method, or you can write your own that returns a boolean value. The signature must follow:

```
public boolean parameterReadyMethod(Parameter parameter)
```

The `parameterReadyMethod` is then implemented by a process subclass and can be named anything, provided it matches the string that is specified in the `addMetaInputParameter` method. The framework uses Java Reflection to make sure that the method exists when the `addMetaInputParameter` method is called, and an `IllegalArgumentException` will be thrown if it does not exist.

An `InconsistentParameterUsageException` may be thrown at runtime if multiple processes within a simulation specify the same parameter of the same entity class and one or more specifies that it needs the parameter annotated, and one or more specifies that it does not need it annotated. A parameter cannot be both.

A `NoSuchInputParameterException` is thrown if an input parameter was not declared in `initializeInputParameters` when the connection between the entity and the local process name is tried during process execution.

A `NoSuchOutputParameterException` is thrown if an output parameter was not declared in `initializeOutputParameters` when the connection between the entity and the local process name is tried during process execution.

### 2.2.3 The SimulationServerImpl Subclass

When the simulation server is executed, it starts by calling `SimulationServerImpl.buildContext`, which is an abstract method that needs to be implemented by the subclass. This method needs to set up a region definition (a holder for the scenario name and description), instantiate the subclass of context builder for the simulation, and call `ContextBuilder.buildContext`, passing in the simulation parameters as well as the list of models that are to be used for the run.

### 2.2.4 Context Builder

When the `ContextBuilder.buildContext` method is called from the simulation server, the superclass takes the list of models (assembled from `getModelsForRegion` and passed into the method) and sets the models to be used; it then calls the methods `createApplicationData` and `createInitialEntities`, which must be implemented by the subclass. See Figure 6 for a flow chart of the context build process.

After the initial set of entities is instantiated, the E-A-P-M connections are made for each type of entity, and the needed parameters for those models are initialized and a default is set within each entity instance. This behavior is inherited, and all the application developer has to do is implement three abstract methods to make it specific to a domain. These methods are:

1. `getModelsForRegion`
2. `createApplicationData`
3. `createInitialEntities`

The `getModelsForRegion` method returns a map of the E-A-P-M connections created via the `addProcessToModelCategoryMap` method. Use the `addProcessToModelCategoryMap` method to do this. The input parameters are:

- Category map (Map)
- Category of the model (String)
- Name of the model (String)
- Entity class (Class)
- Name of the aspect (String)
- Process class (Class)
- Model class (Class)

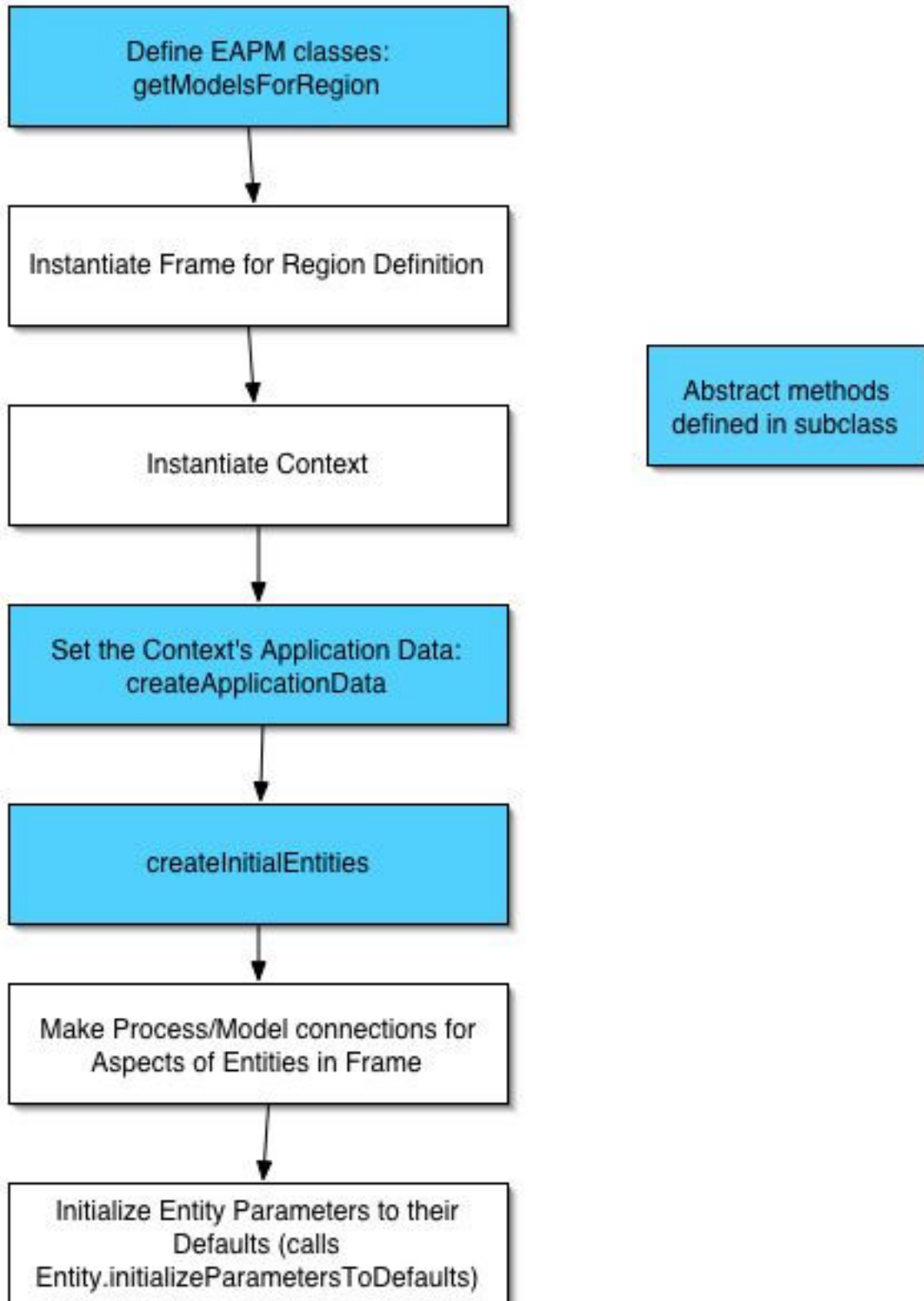


Figure 6 Flow Chart Showing Context Build Process

See the example code below:

```
public Map getModelsForRegion(RegionDefinition arg0) {
    Map categories = new HashMap();
    addProcessToModelCategoryMap(categories, "Farming",
    "Raining", Atmosphere.class,
    "precipitation", PrecipitationProcess.class,
    PrecipitationModel.class);
}
```

The `createApplicationData` method must return a new instance of the subclass of `ApplicationData`.

The `createInitialEntities` method needs to create the initial entities for the simulation. Entities that may show up or leave a simulation during execution are initialized slightly differently (see Section 3.2, p. 35). The `newEntityDuringBuild` method must be called in order to have the parameters instantiated properly for each entity after its constructor is called. See the code example below:

```
Atmosphere atmosphere = new Atmosphere(frame);
newEntityDuringBuild(atmosphere);
```

Only after the parameters have been instantiated can you call `setValue` for a parameter. Values can be read from file or a database, or any other source, but if they are to be stored in an entity's parameter, the `setValue` method must be used. If you set a value here for any parameter, do not also set a value in the `Entity.initializeParametersToDefault` method because it will be overwritten, since the default method is called after `createInitialEntities` (see Figure 6).

The method `addParameterDependency` method is called from `createInitialEntities` for any parameters that need to get initialized but are not defined within any process's `MetaInputParameters` or `MetaOutputParameters`. Between `addParameterDependency` and `MetaInput/OutputParameters`, all needed parameters for a simulation must be declared. If they are not, and the parameter is called by `setValue` or `getValue`, a `ParameterNotInstantiated` exception will be thrown during runtime.

The DIAS Framework uses the models map to create `anl.dias.simulation.ProcessConnections`, which holds the entity, aspect (name), process, and model instances that make up an E-A-P-M connection. The call that makes the physical connection so that the DIAS framework knows when a process/model is to be executed for an entity's aspect is made at the end of the context build stage (see Figure 6). Note that if an application needs to override the E-A-P-M connections normally made by the framework during the simulation, for instance if you want to connect a different process/model for a single instance of an entity (overriding the one that is set for all other entities of that same type), you can instantiate a process connection with

only the instances involved and call the connect method on it programmatically. But you will also have to call the disconnect method if the connection is to be severed during the simulation. This technique could be used if an application needs to have different process/model connections attached to the same subclass of entity or aspect during the simulation.

## 2.3 Coding Simulation Dynamics

This section describes the methods that are overridden in the subclasses to define the execution of the simulation. The implementation differs depending on whether the model will be internal or external. At the heart of a DIAS simulation is a simulation event manager, which provides the “dynamics” of the simulation. Any object in a DIAS simulation may register to receive events and may schedule events with the simulation manager. Simulation time is moved forward by processing the events in order until the end of the simulation is reached (see Section 3.3, p. 37, for further discussion).

### 2.3.1 Internal Models

#### 2.3.1.1 ApplicationData Subclass

The methods `registerForEvents` and `unregisterForEvents` can be overridden in the `ApplicationData` subclass. If they are overridden, call the superclass method as well.

#### 2.3.1.2 Entity Subclasses

The entity subclasses may also override `registerForEvents` and `unregisterForEvents` as well, and also have to call the superclass method if overridden.

The method `initializeForSimulation` may be overridden if there is something domain-specific that needs to be done for an entity at the time of simulation initialization. The superclass method must still be called, however, since it calls the `registerForEvents` method and initializes entity aspects.

The `terminateForSimulation` method is called when the simulation finishes, which in turn calls `unregisterForEvents` and terminates any aspects (and corresponding process/model connections). You may override this method if there is anything domain-specific that needs to be done at the end of the simulation, but, again, you must also call the superclass method.

### 2.3.1.3 Process Subclasses

Like the entities and `ApplicationData`, processes may also override `registerForEvents` and `unregisterForEvents`. You need to call the superclass method since it needs to register for the special `executeProcess` event that the framework calls to start executing the process.

Because the `connectInputAndOutputParameter` method is an abstract method, subclasses must declare the connections of the input and output parameters with the specific instance parameters of the entity to which the process is connected. The methods `connectInputParameter` and `connectOutputParameter` are provided for you to use within the `connectInputAndOutputParameter` method. Pass in the local name for the parameter and the entity as method parameters. The local name should match the local name specified in the `MetaInputParameter` and `MetaOutputParameter` declarations called from the static initialization block.

You do not need to specify all of the parameters declared in the `MetaInput/OutputParameters`, but only the ones where you want to take advantage of the automatic dependency checking for process execution. Recall that the `parameterReadyMethod` declared in the `addMetaInputParameter` method is called for every input parameter when the process is told to execute. The methods check if all the parameters are “ready,” and if so, the process may start to execute. However, if any of the parameters are not ready at that time, the process is queued until such time that the parameters may be ready. In order for the system to “know” when to check again, you should declare dependent parameters using the `connectInputParameter` method. If an input parameter is connected in this way, whenever the `setValue` method is called on the entity for that parameter, it triggers the DIAS framework to check each process that said it was dependent on that parameter value if it is now ready for execution, and if so, the process may start execution. Note that the corresponding method `connectOutputParameter`, is not really used and probably should just be marked as deprecated.

The methods `initializeAtSimulationStart` and `terminateAfterExecution` need to return boolean values that tell the framework how the process should be executed. The methods return values in opposite pairs:

- If `initializeAtSimulationStart` returns *true*, the `terminateAfterExecution` method should return *false*. This indicates that the process will be initialized at the start of the simulation and not terminated until the end of the simulation. This setup is normally done for processes that are connected to models that have an initialization setup, any number of executions during a simulation, and terminate after the simulation is done.
- If `initializeAtSimulationStart` returns *false*, the `terminateAfterExecution` method should return *true*. This indicates that the process will initialize, execute, and terminate each time it is requested to execute. This setup is normally done for processes that are connected to models that do not maintain a state between execution calls.

The `buildProcessInputData` method is used to gather all the input data for the process that is needed for the execution of the model. Any object can be returned, but if multiple objects are needed for passing, a map is a convenient holder of the data. The object is stored in the `ModelState` instance which is passed to the model (via the method defined in the `startProcess` method).

The `startProcess` method makes the actual call to the entry method in the corresponding model. It usually follows the pattern:

```
getModel().entryMethodName(ModelState)
```

The `entryMethodName` can be any user-defined method name, but it needs to be implemented in the model subclass.

The `unpackProcessOutputData` is called by the framework after the model finishes executing. The object passed as the parameter to this method is the object stored when the model called `ModelState.setOutputData`. The object stores the output data and gets unpackaged here. The method usually sets any parameters of entities and/or it may schedule future events (see Section 3.3, p. 37).

The method `processInitializing` can be overridden if there is something that needs to be specifically done at initialization time for the process. There is no need to call the superclass method, since it does nothing. It is a “hook” to give the developer an option.

The method `isAutoLaunchable` defaults to returning *false* in the superclass. If your process subclass overrides this method to return *true*, that indicates to the framework that the process will automatically execute when all its parameters are ready. The default value of *false* indicates that the process can only execute by calling `performAspect` on the corresponding entity (see Section 3.5, p. 38).

The method `isSynchronous` may be overwritten for the process. All processes default to *true*, meaning that the process/model execution blocks until execution is done. The `processDone` method is called automatically then if the process is synchronous. If, however, you override this method to return *false*, you must call `processDone` when model execution is finished.

The method `executionEventPriority` sets the default priority for the execution of this process (`SimulationUtilities.getDefaultPriority`). If you override this method, you increase or decrease the event priority for a specific process (see Section 3.4, p. 38).

#### 2.3.1.4 Model Subclasses

The `newModelState` method needs to return an instance of `ModelState` for the process and model to pass back and forth. This could be an instance of `anl.dias.core.ModelState`, or a domain-specific subclass created for the application (see Section 2.1.5, p. 14).

The `getName` method needs to return a string. The name is used for logging and debugging purposes.

Every model subclass needs to implement the method that was declared in the corresponding `Process.startProcess` method with the `ModelState` as its parameter. This method is the entry method for the model. The method `ModelState.getInputData` is used to retrieve the object that was returned from `Process.buildProcessInputData`, and the method `ModelState.setOutputData` is used to package the output data from the model to be passed back to the process where it can be unpacked. In between these two calls is all the domain-specific code that actually implements the modeling.

### 2.3.2 External Models

The subclasses for external models are slightly different and add a couple of new methods that need to be implemented. By default, DIAS uses Java RMI as the communication mechanism between the DIAS simulation and the external model's model controller. Therefore, the model controller is written in Java, whereas the external model can be implemented in any language. If you wish to write the model controller in a non-Java language, DIAS provides a CORBA framework to use, which follows closely to the RMI framework discussed in this section.

#### 2.3.2.1 Model Controller

Following the RMI paradigm, you first need to define a remote interface for the model controller that extends `anl.dias.core.rmi.AbstractRMIModelController` (which extends `java.rmi.Remote`). There are three methods that are called remotely, and thus must throw `java.rmi.RemoteException`:

1. `initializeModel`
2. `executeModelForTimestep`
3. `terminateModel`

The `initializeModel` and `terminateModel` method names are defaulted to these names, but they can also be user-defined (see Section 2.3.2.3). The execution method is user-defined, much in the same manner as an internal DIAS model. The initialize and execute methods take an object as an input parameter and return an object. Since DIAS employs RMI, these objects, and everything contained within them, must implement the `java.io.Serializable` interface. Therefore, by convention, we normally create a domain-specific class that implements `java.io.Serializable` and holds all the data needed for either input or output. See the following code example:

```
public interface FreemansCropModelController extends
RMIModelController {
    public FTOutputData initializeModel(FTInitData data)
throws RemoteException;
    public FTOutputData executeCropGrowth(FTInputData
inputData) throws RemoteException;
    public void terminateModel() throws RemoteException;
}
```

Next, you need to create an implementation of the model controller. The class must implement the remote interface defined above and should extend `anl.dias.core.rmi.AbstractRMIModelController`. The `AbstractRMIModelController` class does the RMI registration binding and unbinding, exporting and unexporting of the object with the RMI registry. The subclass *must* call `super.initialize` from its constructor (passing in *true* as a parameter) in order to start up an `RMISecurityManager` and run the model controller remotely. If you want to reuse the model controller subclass as a non-remote class, you can pass *false* into the initialize method.

In addition to the three methods to be implemented from the remote interface, the method `getLookupName` must also be implemented. This should return the Universal Resource Identifier (URI)<sup>1</sup> as a string for registering in the RMI registry. This string will be the URI that the client (DIAS) needs in order to lookup the reference to the model controller so that it may call the methods remotely.

### 2.3.2.2 External Process

If a Process is going to be connected to an external model, it still extends `anl.dias.core.Process` and implements the methods discussed above as with internal models. However, the `buildProcessInputData`, `startProcess`, and `unpackageProcessOutputData` deal with slightly modified objects in order to communicate via RMI. Since the remote interface to the model controller takes one input object and returns an object, corresponding classes must be created to hold all the input and output data for the method calls and they must implement `java.io.Serializable` to work with RMI. These classes would then be used in the process subclass.

The `buildProcessInputData` method makes an instance of the serializable input data class (`FTInputData` in the example), fills it with the data needed for the time step (just like in the internal model example), and returns the object reference.

The `startProcess` method calls the user-defined method of the corresponding model, but the `ModelState` that is passed in is the `RMIModelState` or a derivative.

---

<sup>1</sup> See <http://www.w3.org/Addressing/> for information on URI syntax and standards.

Finally, the `unpackageProcessOutputData` gets the serializable output data object (`FTOutputData` in the example) that it can use to access any of the model output data and perform the same operations that can be done using the output from an internal model.

Note that the same technique can be used for internal models, in that a specialized input class and output class can be used instead of the traditional map as described in the internal model section. It is up to the user to define (and cast properly) which objects are used in the `buildProcessInputData` and `unpackageProcessOutputData` methods. In the case of an internal model, the object subclasses need not be serializable, but they can be without affecting execution.

### 2.3.2.3 External Model

A model that is going to be connected to an external model needs to extend `anl.dias.core.rmi.RMIExternalModel`. Along with the usual model methods to be implemented by a subclass, there are some more that are specific to using an external model. The `modelControllerURI` method returns the URI (as a string) of the model controller reference within an RMI registry. It must match the URI specified when the model controller exported itself to the registry. As part of the initialization process, the reference to the model controller is looked up using the URI string.

The `newRMIModelState` method is used in much the same manner as `newModelState` for internal models. It is overwritten to return an instance of the `RMIModelState` subclass (see Section 2.3.2.4). The default superclass method returns an instance of `RMIModelState`. You only need to overwrite this method if you are returning a domain-specific subclass of `RMIModelState`.

The `getInitializationDataClass` method needs to return the class of the object being used for initialization of the external model (`FTInitData` in the example). The `RMIExternalModel` uses Java Reflection to lookup the initialization method using this class as the method parameter.

The `modelInitializationData` method instantiates the initialize data object, fills it with the appropriate data, and returns the object reference.

The protected method `setInitializationOutputData` can be overwritten to accept any output data object that may be returned from the initialization of the model. For example, if something that the model initializes is needed within the DIAS simulation, it can be returned in an object (defined in the remote interface as the return parameter from the `initializeModel` method) and accessed by overwriting `setInitializationOutputData` to set any needed variables on the simulation side. The super method does nothing by default.

If you want to call your initialize and terminate model methods something other than the default (`initializeModel` and `terminateModel`), then you need to overwrite the `getInitializeMethodName` and `getTerminateMethodName` to return the appropriate string.

This method name will have to be used in the remote interface as well as the model controller implementation. The framework uses Java Reflection to invoke these methods.

Finally, you need to implement the remote call to the model controller via the user-defined execute method (called from the corresponding process). As an example, the `growCrop` method is called each time step in the simulation, it in turn makes the remote method call and must follow the correct steps in order to correctly pass parameters. See the following code example:

```
public void growCrop(RMIModelState state) {
// 1. for an external model, get the corresponding Model
Controller
    FreemansCropModelController mc =
(FreemansCropModelController)state.getModelController();
// 2. get the input data (created in
GrowProcess.buildProcessInputData)
    FTInputData idata = (FTInputData)state.getInputData();
    try {
        // 3. call the user-defined execution method
(method call on the Remote interface)
        FTOutputData odata = mc.executeCropGrowth(idata);
        // 4. set the output data, so that it gets
returned to the Process (GrowProces.unpackProcessOutputData)
        state.setOutputData(odata);
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
```

#### 2.3.2.4 External ModelState

The default class used for external models is `anl.dias.core.rmi.RMIModelState`. If there is a need to hold something domain-specific in this class or to override any methods, you can make a domain-specific subclass. You will need to overwrite the `newRMIModelState` method of your `RMIExternalModel` subclass so that it will be used instead of the default.

The `RMIModelState` does the actual lookup of the model controller reference in the RMI registry and holds onto the reference. That reference is used by the DIAS framework to make the remote method calls to the model controller.

The model controller reference is looked up and the `initialize` method called when the `newModelState` method is called by the framework. This class also sets up an `RMIStateManager` if one has not been set up.

### 2.3.2.5 RMI Security and Other RMI Issues

The DIAS simulation (the DIAS server) is considered the RMI “client” whereas the model controller is treated as the RMI “server.” Since both the RMI client and the server are instantiating an `RMI Security Manager`, policy files need to be set up to assign the proper privileges.

The property `-Djava.security.policy` is used on the command line when starting both the model controller in its JVM and the DIAS simulation (RMI client) within its JVM. The `server.policy` is defined for the model controller, and a separate file, often named `client.policy`, is used for the DIAS simulation. Refer to the run scripts in the Farm Tax demonstration for an example of how to set this property and to see the contents of an RMI policy file.

Property files for the server and client are also used to define the URI for the model controller registry and lookup. This makes it more portable, since you can define the host name and port within the property file instead of hard-coding it.

Note that if you are running the model controller in a separate JVM, you must use the RMI compiler (`rmic`, which comes with the Java Developer’s Kit tools) to create a stub file. Refer to <http://java.sun.com> tutorials for more on the RMI. See the Ant<sup>2</sup> build file in the demonstration code to see how to call the `rmic` from Ant.

### 2.3.2.6 Types of Interfaces to External Models

When writing the model controller, you must decide how you will call the external program. If the model is in Java, you may just call any methods, perhaps via RMI. If the program is in another language, you could use CORBA.

Another method of calling the external program is to use Java Native Interface (JNI) to make calls directly. This method requires knowledge of C and possibly bindings to Fortran, and it risks the introduction of any type of error that may occur in these languages. An exception in C or Fortran will most likely dump the JVM that the model controller is executing within, since the shared library is loaded into the JVM when a JNI interface is used.

The third way to interface with an external program is to use `java.lang.Runtime.exec()` (or a system call to an executable or script). You can pass command line arguments and read/write stdin, stdout, and stderr. If the model reads and writes files, you will have to write out input files from the model controller for use in the external model, and read in output files created from the external model, package the data, and send it back to the simulation. This is the most common “black box” way of calling legacy models, that is, without knowing details of their internal operation.

---

<sup>2</sup> Apache Ant is a java-based open-source build tool. See <http://ant.apache.org/> for more information on Ant. See the manual for your Integrated Development Environment (IDE) for integration of Ant.

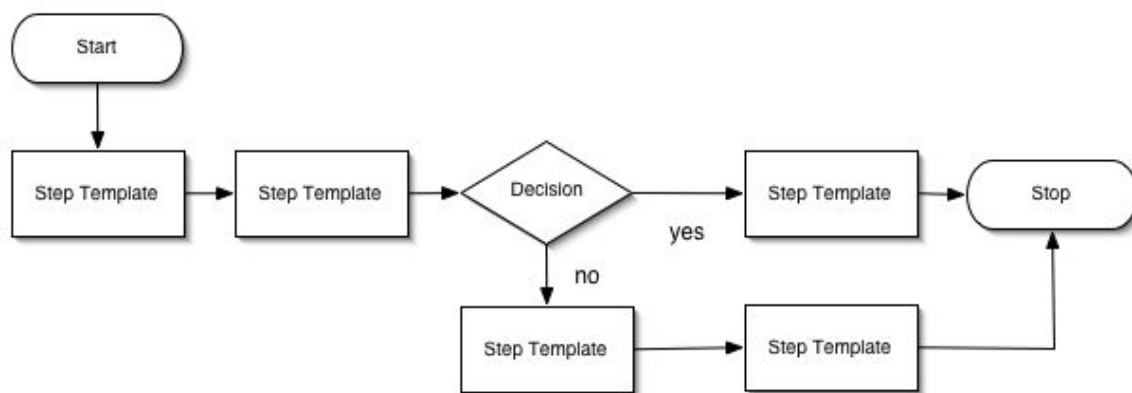
### 2.3.2.7 FACET Processes and Models

The course of action (COA) class is a special type of DIAS process and model. The COA classes are the DIAS implementation of the FACET concept. The intent of the COA architecture is to be able to build simulation models of societal behavior patterns — interaction between participants in a task with contention for resources — within a mainstream DIAS simulation. A COA Model is like a flowchart of individual steps and a flow based on decisions and is used to model interactions between participants. A COA represents an aspect of an entity just like any other process or model within DIAS. Figure 7 shows a representation of a COA Model.

The steps in a COA model are actually step template objects, where the modeler defines the participating agents (DIAS entities or other participants), any resources required by the participants, the duration a step may take, and whether or not there is a timeout or exception method associated with the step.

The FACET main concepts and classes within DIAS include the following and are located in the `anl.dias.coa` package:

- **CourseOfActionProcess** – a subclass of DIAS Process used to start and stop corresponding COA models. They override the same methods that other Process subclasses must implement.
- **CourseOfActionModel** – a subclass of DIAS Model used to set up Steps and their Participants to describe an interaction process.
- **StepTemplate** and **StepParticipantTemplate** – classes used for instantiating the steps and participants for an execution of an instance of COA (the instance of COA is actually the model state for the corresponding COA process and COA model classes – it holds the actual instances of the steps and participants for an invocation of the COA model).



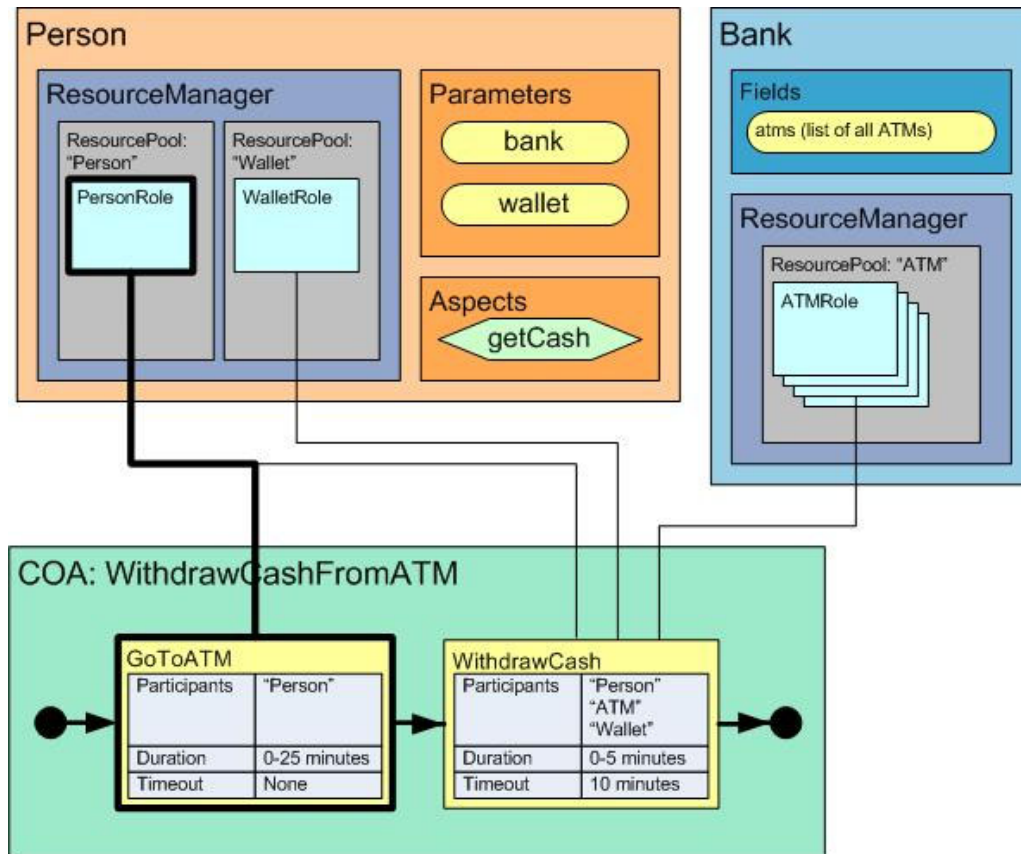
**Figure 7 Step Templates Defining a Course of Action Model**

- **Step** – a procedure, or submodel, in a COA model that may have duration, participants, and at least one exception step (i.e., a step timeout).
- **Participant/ParticipantOwner** – the interface that allows the class to participate in COAs.
- **Agenda** – a to-do list of COA steps that require the participation of a specific participant in a COA performing a specific role.
- **Role** – Participants may participate in any number of COAs in different roles. Subclasses of role override the `whatNext` method which allows different roles to look at the owner’s agenda and decide which COA step they will do next.
- **ResourceManager** – manages instances of roles.
- **ResourceManagerOwner** – implementing the `ResourceManagerOwner` allows the class to be a resource manager.
- **ResourcePool** – a pool of potential participants (based on role) managed by a resource manager.

The remainder of this subsection is an example to illustrate the COA concepts.

Suppose that a bank customer entity has a `getCash` aspect, which is connected to a `WithdrawCashFromATM` process and `WithdrawCashFromATM` model (see the classes within the `anl.csiro.farntax.coa` package in the Farm Tax demonstration code for the implementation of this example). When the `getCash` aspect is triggered, the bank customer entity will attempt execution of the withdraw process (just like any other DIAS process). Since the `WithdrawCashFromATM` process is a COA process, however, it will start “executing” right away (whereas a regular DIAS process will attempt execution only after all its parameters are ready — see Section 3.5). The `buildProcessInputData` method in this case is used to include any specific instances needed within the model steps. In the example, we want the instance of the entity (the bank customer [or farmer in Farm Tax]) to be the instance of “person” participating in this COA’s execution. The key, person in the Farm Tax example, must match the key used in defining the `StepParticipantTemplate` within the COA model. A COA instance is created, instantiating the steps and the participants from the COA model `StepTemplates` and `StepParticipantTemplates`.

The entry step is “enabled” to execute, and the framework will “invite” all the participants to join the step by asking the resource manager for a participant from their resource pool based on a suitable role instance. In the case of the Farm Tax demonstration, the bank customer is the resource manager for itself and its wallet: it maintains two resource pools for itself, one each for the roles of customer and wallet. The bank class is the resource manager for the ATMs (as ATM role objects). Figure 8 shows a diagram of the pieces (note that `Person` and `PersonRole` are replaced with `BankCustomer` and `CustomerRole`, respectively, in the Farm Tax demonstration). When a resource manager is asked for a resource, it looks in its resource



**Figure 8 Schematic of the WithdrawCashFromATM COA Model**

pool for an instance of the role requested and then asks the resource to participate in the step. The invited participants will then look at their agendas and decide whether or not to participate at this time. If a participant agrees, it is committed to the COA step and waits for the other participants to commit. In this way, the model can represent queuing delays due to the waiting for resources and participants to commit to a step. This process is asynchronous, meaning that if all participants cannot commit when asked, the step is suspended and execution may continue with the rest of the simulation. When a resource is returned to its resource manager's resource pool (perhaps after finishing a step in another COA), the resource manager asks again if it will participate in the step. After any number of iterations, when the last resource finally commits, the step may then execute. At the end of a step, decision logic in the step determines the next step to execute, the participants are dismissed (if the `StepTemplate` indicates that the participant is to be dismissed between steps) and the invitation process starts anew.

The COA framework relies heavily upon Java Reflection; therefore, the definition of `StepTemplates` requires names for the duration method, the perform method, and the next step method. The templates are instantiated in the COA model's constructor, and the methods are defined in the rest of the COA subclass. Below is the constructor for the `WithdrawCashFromATM` model:

```

public WithdrawCashFromATM() {
    super();
    goToATMDurationRandom = new
    RandomUniform();withdrawDurationRandom = new
    RandomUniform();StepParticipantTemplate partTemplate;
    ActionStepTemplate action;

    // create first step
    action = new ActionStepTemplate("GoToATM",this,
    "nextStepAfterGoToATM",
    "goToATM", TimeInterval.seconds(0), "goToATMDuration");

    partTemplate = new StepParticipantTemplate("person",
    "BankCustomer", "personAttention",this, new
    Requestee("person"),false,false);

    // add participants action.addParticipant(partTemplate);
    // add it to the COA
    addStep(action);
    // tag it as the first step
    setEntryStep("GoToATM")

    // create second step
    action = new ActionStepTemplate("WithdrawCash",this,
    "nextStepAfterWithdraw","withdrawCash",
    TimeInterval.minutes(10), "withdrawalCashDuration");

    // create and add participants
    partTemplate = new StepParticipantTemplate("person",
    "BankCustomer",new Integer(1), new
    Requestee("person"),true,false);

    action.addParticipant(partTemplate);
    partTemplate = new StepParticipantTemplate("wallet",
    "Wallet",new Integer(1), new
    Requestee("person"),true,false);

    action.addParticipant(partTemplate);
    partTemplate = new StepParticipantTemplate("atm", "ATM",
    new Integer(1), new
    Requestee("getBank",BankCustomer.class),true,false);
    action.addParticipant(partTemplate);

    // add it to the COA
    addStep(action);
}

```

In the first `StepTemplate`, the `goToATMDuration` method needs to be implemented in the COA model subclass. It needs to return an `anl.util.time.TimeInterval` for the duration of the step, but this may be calculated within the method or just simply returned as a value (e.g., 10 minutes). This method is called at the beginning of the execution of the step, after all participants have committed. Therefore, if you need to know when a step begins, you can put logging or other processing in the duration method. The perform method, `goToATM` in this case, is also defined in the class. This method should contain the logic for performing the step. It is actually executed at the end of the step (start time plus step duration time). Finally, the next step method is called, `nextStepAfterGoToATM` in this case. The method must return a string containing the next step's name or return null if there are no more steps to be executed. This method would hold any logic needed for a decision on the next step. Otherwise, it can return the same step name each time it is called. All these methods have two parameters passed to them from the framework: a map of local variables and a map of arguments. This is the way that the actual instances of the participants (and anything else that needs to be saved and passed between steps in a COA) are accessible within the methods. The map of local variables includes anything declared in the COA model template stage as being "local" to the execution of the COA, and the arguments map contains the participants and anything else put into the map from the `COAProcess.buildProcessInputData` method. The keys are defined within the `StepParticipantTemplates` and are stored in the map with those keys.

Any object implementing `ResourceManagerOwner` needs to provide an instance variable of `ResourceManager` and implement the `getResourceManager` and `setResourceManager`, methods which are simple getter and setter methods. The framework will instantiate a resource manager instance when `enableResourceManagement` is called upon the object (see the example of the bank in `FarmTaxContextBldr.createInitialEntities`):

```
Bank bank = new Bank();
ResourceManager.enableResourceManagement(bank);
```

The same requirements apply to any object implementing `ParticipantOwner`: it needs to provide an instance variable of participant and implement the `getParticipant` and `setParticipant` methods. The framework will instantiate a participant instance when `enableParticipation` is called upon the object. In the example, the bank is a resource manager of the ATM, but the ATM is the participant in the step in the withdraw COA. The code below shows the ATM being set up:

```
Atm atm = new ATM("ATM "+i);
Participant.enableParticipation(atm);
```

Finally, after the ATM is created, the bank needs to become its resource manager. The following code shows how to add the manageable role types, then add the instances for that role:

```
ResourceManager bankResMgr = bank.getResourceManager();
bankResMgr.addManageableRole("ATM", ATMRole.class);
bankResMgr.manageResource(atm, new String[] {"ATM"}, 0);
```

A DIAS Entity can implement the `ResourceManagerOwner` interface, the `ParticipantOwner` interface, or both depending on what is needed. The methods must be implemented the same way, and parameters can be used to hold the participant instance and/or resource manager instance instead of making instance variables. See the code for `BankCustomer` (which extends entity):

```
public static void initializeMetaParameters() {
    Class cls = BankCustomer.class;
    addMetaParameter(cls, "resourceManager");
    addMetaParameter(cls, "participant");
}
```

### 3 The Inner Workings of the DIAS Framework

#### 3.1 Context Execution

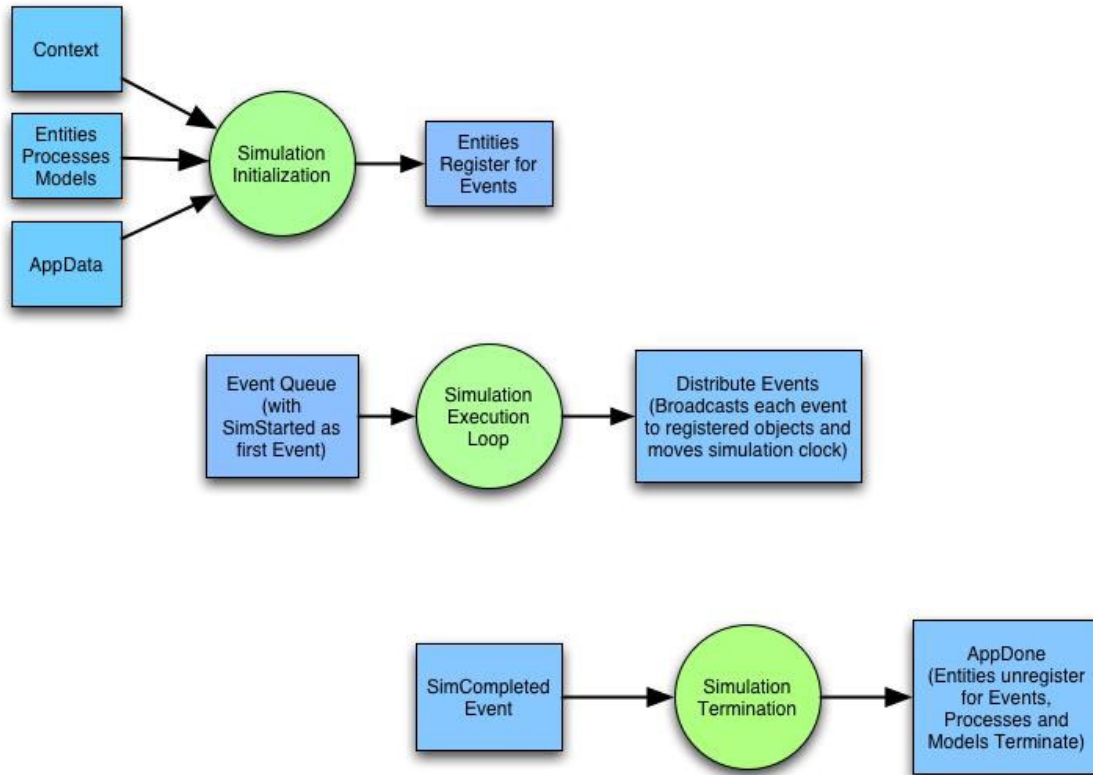
Context execution consists of three stages:

1. Simulation initialization
2. Simulation execution
3. Simulation termination

In the initialization stage, the built context and all the entities in the analysis frame — including the run-time specification of the process and models connection and the application data — register with the simulation manager to receive the events they are interested in. The execution of the simulation starts with the first event on the queue (`SimulationStarted`) and broadcasts the event to any registered listeners. The simulation manager continues retrieving events from the queue and broadcasting them until the `SimulationCompleted` event is done. At that time, everything that is running shuts down: listeners unregister for events, models are terminated, and processes are disconnected from the entities and aspects. Figure 9 shows a simplified schematic of the context execution elements.

#### 3.2 How to Create Entities in a Frame

The `ContextBuilder.createInitialEntities` method is where the initial entities for a simulation are built (see Section 2.2.4, p. 19). Three helper methods are available, one of which must be called on each entity in order to have the E-A-P-M connections and parameters initialized properly for the simulation.



**Figure 9 Context Execution Steps**

The method `newEntityAfterBuild` must be called if you create a new entity after the context build phase. This method instantiates the parameters and process connections, assigns a default value to the entity parameters, and initializes any of the connected process's parameters to their defaults.

The method `newEntityDuringSimulation` must be called (after `newEntityAfterBuild`) if an entity is created during the simulation (e.g., a new farmer). The method calls the `connectInputAndOutputParameters` method for each process connected to the entity and initializes the aspects for the entity. These are things that are usually done automatically by the framework for the initial entities established during the context build stage, but it needs to be manually called by the developer if done after the build stage or during the simulation execution.

If you remove entities from the frame during the simulation, for example, if you are modeling the death or destruction of an entity, you will have to call `Entity.terminateForSimulation`. This method will call `unregisterForEvents` and `terminateAspects`. If your entity subclass overrides either of these methods, you must also call the superclass method. The termination of aspects propagates to the currently connected processes/models to stop their execution as well. After calling the `terminateForSimulation` method to remove the entity from the simulation frame entirely, call `Entity.removeEntityFromFrame`.

### 3.3 The Simulation Manager and Scheduling Events

The DIAS simulation manager executes a simulation in its own Java thread and is responsible for the flow of time within a simulation. The simulation manager schedules the `simulationStarted` event to occur at the context start time and schedules the `simulationCompleted` event for the context end time.

The simulation manager takes care of pausing and resuming the simulation, so the developer need not have access to those events. All events in the simulation are class `anl.dias.simulation.SimEvent`, which holds:

- Sender of the event,
- Simulation timestamp that the event is scheduled for, and
- Any optional data object used to hold event-specific data to be passed to event handlers.

Developers can have any simulation object (entity, application data, or process) register to receive the `simulationStarted` and/or `simulationCompleted` events and write event handlers to do any domain-specific processing at the start or end of the simulation. At least one domain object needs to process the `simulationStarted` event, so that it can schedule other domain-specific events. All entities are registered to receive the `simulationCompleted` event by the framework, so if you override this method, you need to call the superclass's method. It is important to call the superclass handler, so that the framework can do cleanup, such as detaching processes from aspects.

Event handlers may schedule other events, may get the sender object out of the `SimEvent`, may create entities or remove them from the frame (see Section 3.2, p. 35), and may call `performAspect` on an entity. The code executed within event handlers is what keeps a simulation moving forward in time and contains or invokes most of the domain-specific code for the application.

Any object may register to receive an event that is sent by any other object, or from a specific sender. For example, if you are modeling a death event for an individual, a population entity probably would be interested in knowing when an entity in its population dies. It can register for all death events sent from any individual. Its death event handler could then tally some statistics, and then unregister to no longer receive events from that individual. The individual entity also registers for the death event, but it is only interested in its own death, not all deaths in the simulation, so it can register for the event where the sender of the event is itself (*this*).

In this example, the death event of an individual means that no more processing is going to be done for it. You need to call `Entity.terminateForSimulation` on the entity. This method unregisters for all the events it had been registered for and terminates any aspects, which in turn terminates its corresponding processes.

### 3.4 Event Priorities and Simulation Utilities

The simulation manager (implementation) defines default event priorities. The priority of a `SimEvent` is used to more finely control which event gets executed if more than one event is scheduled for the same time. The public static final constants defined in `anl.dias.simulation.SimulationManagerImpl` are as follows:

```
public static final int MaxPriority      = 100;
public static final int MaxUserPriority = 95;
public static final int MinUserPriority = 5;
public static final int MinPriority     = 0;
public static final int DefaultPriority = 50;
```

The `simulationStarted` and `simulationCompleted` events are defaulted to `MaxUserPriority+1`. The user can access the default priority by calling `anl.dias.simulation.SimulationUtilities.getDefaultPriority()`. In fact, the `SimulationUtilities` class is a singleton class within DIAS that can be accessed from anywhere in a DIAS simulation. Useful methods on this class include:

- `getSimulationManager`
- `getFrame`
- `getContext`

And useful shortcuts to the event priority constants include:

- `MaxUserPriority`
- `MinUserPriority`
- `DefaultPriority`

### 3.5 How Processes Execute

If a process is `AutoLaunchable`, it does not need to execute via a call to an entity's `performAspect` method and the process may start whenever its parameters are ready for execution. If a process is not `AutoLaunchable`, the `performAspect` method of the entity needs to be called to start it. If the process did not initialize at simulation start, the `initializeProcess` method is called at this time. Initializing a process means that the instance of `ModelState` (the value returned from `Model.newModelState`) is instantiated and the method `processInitializing` is called. The process also registers for any events it may be interested in receiving. The request to start execution is intercepted by the framework and processed to see if all the execution parameters are ready (the "Attempt Execution" box in Figure 10): if all the parameters have met their parameter-ready state, then an `executeProcess` event is scheduled with the simulation

manager (at the `executionEventPriority` of the process). If not all the parameters are ready, then the next time a `setValue` method is called on any parameter in an entity (or process) its `valueChanged` method is called, which loops through its list of processes (for which the parameter is used as an input) and calls `inputParameterChanged` for each. This method checks to see if the process is waiting for execution (in other words, a `performAspect` attempted to execute this process previously), or if the process is `autoLaunchable`. If either of these conditions is true, it attempts execution again. Therefore, after the last input parameter for a process is set and meets its ready state (the parameter ready method returns a value of `true`), a process may be scheduled to actually execute. See Figure 10 for a flow chart of this processing sequence.

Once the execution is scheduled, the `Process.execute` method is called. Here is where the `ModelState` input data is set from the `buildProcessInputData` method. The `startProcess` method is called, in which the process subclass calls the domain-specific method on the corresponding model subclass, sending in the `ModelState`. Note that the process, by default, is synchronous, meaning that the model execution is a blocking call: the model execution finishes before returning control to the `startProcess` method. Then, the method calls `processDone`. If you override the `isSynchronous` method to return a value of `false`, you may make your model execution nonblocking (asynchronous), *but* you will have to then manually call `processDone` when you know that your model has finished execution, because the `processDone` method calls `unpackProcessOutputData`, sending in the output data from the `ModelState`. If the `terminateAfterExecution` method returns `true`, the `terminateProcess` method is called. Otherwise (if it is `false`), it gets called at the end of the simulation. The `terminateProcess` calls `processTerminating` (which you may override with domain-specific code if necessary), disconnects the `ModelState`, and unregisters for events that it had registered for in the initialization step. See Figure 11 for a flow chart of this processing.

### 3.6 Start Simulation Processing

When a simulation is started, processing starts in a separate Java thread. Figure 12 shows the flow chart for the simulation initialization process. Executing the simulation is sometimes referred to as “context execution,” since it is the built context, with all its entities and their process/model connections, that is running in the simulation. The first step is to set the current simulation time to the context start time. The event queue is reset. The DIAS event manager maintains the queue and keeps the information regarding which objects registered to receive what events. The run state of the simulation is set to *initializing* (the possible run states are *initial*, *initializing*, *paused*, *running*, and *done*). The simulation manager will then schedule the `simulationCompleted` event to occur at the context end time.

Next, the manager will call the `initializeForSimulation` method on each entity in the context’s frame. The `initializeForSimulation` method may be overridden by subclasses, but it *must* call the superclass method in order to inherit important processing. The inherited behavior is for the entity to register for events that it is interested in and to initialize its aspects. The `initializeAspects` method, in turn, checks to see if its associated process is set to initialize at

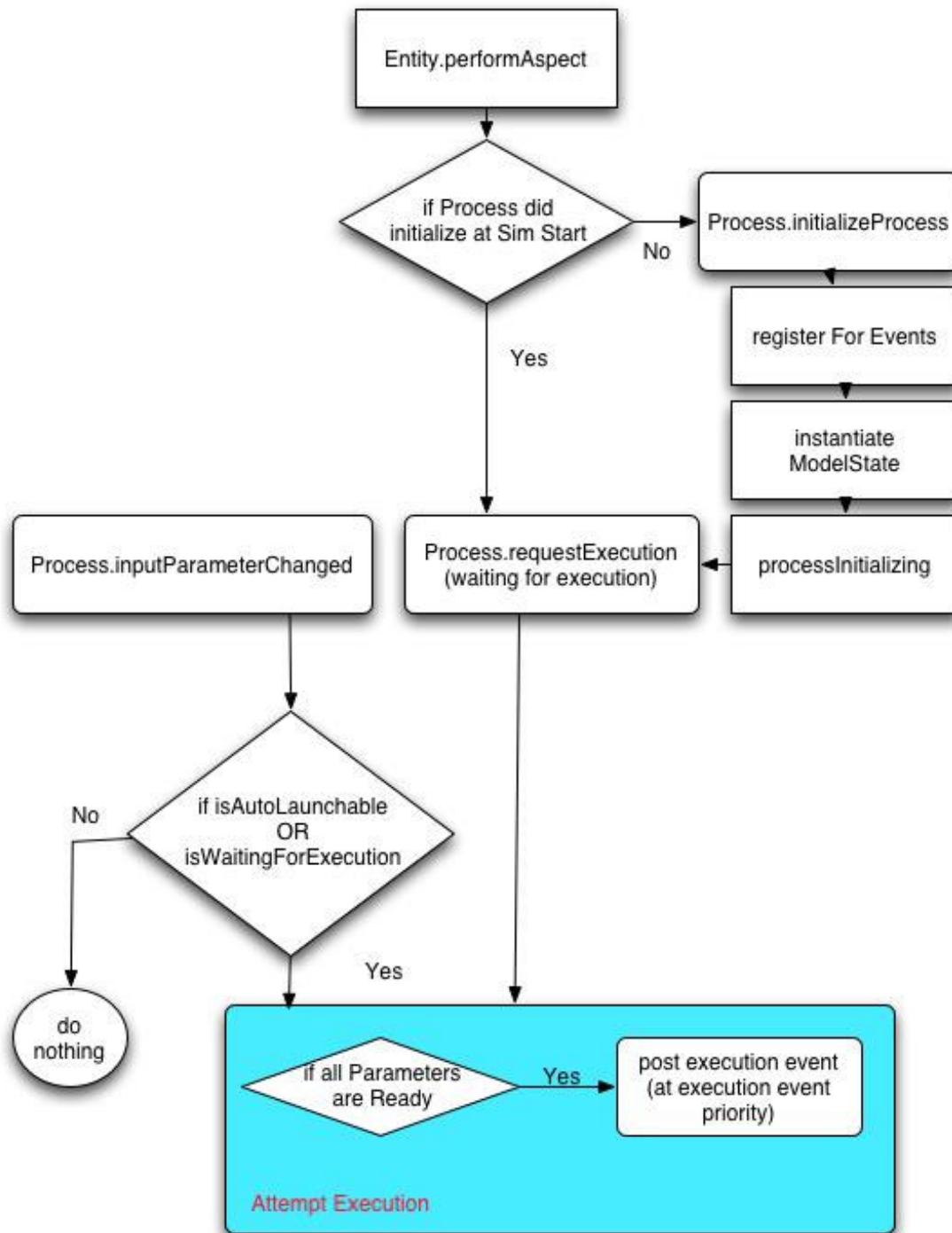


Figure 10 Flow Chart for the Attempt to Execute a Process

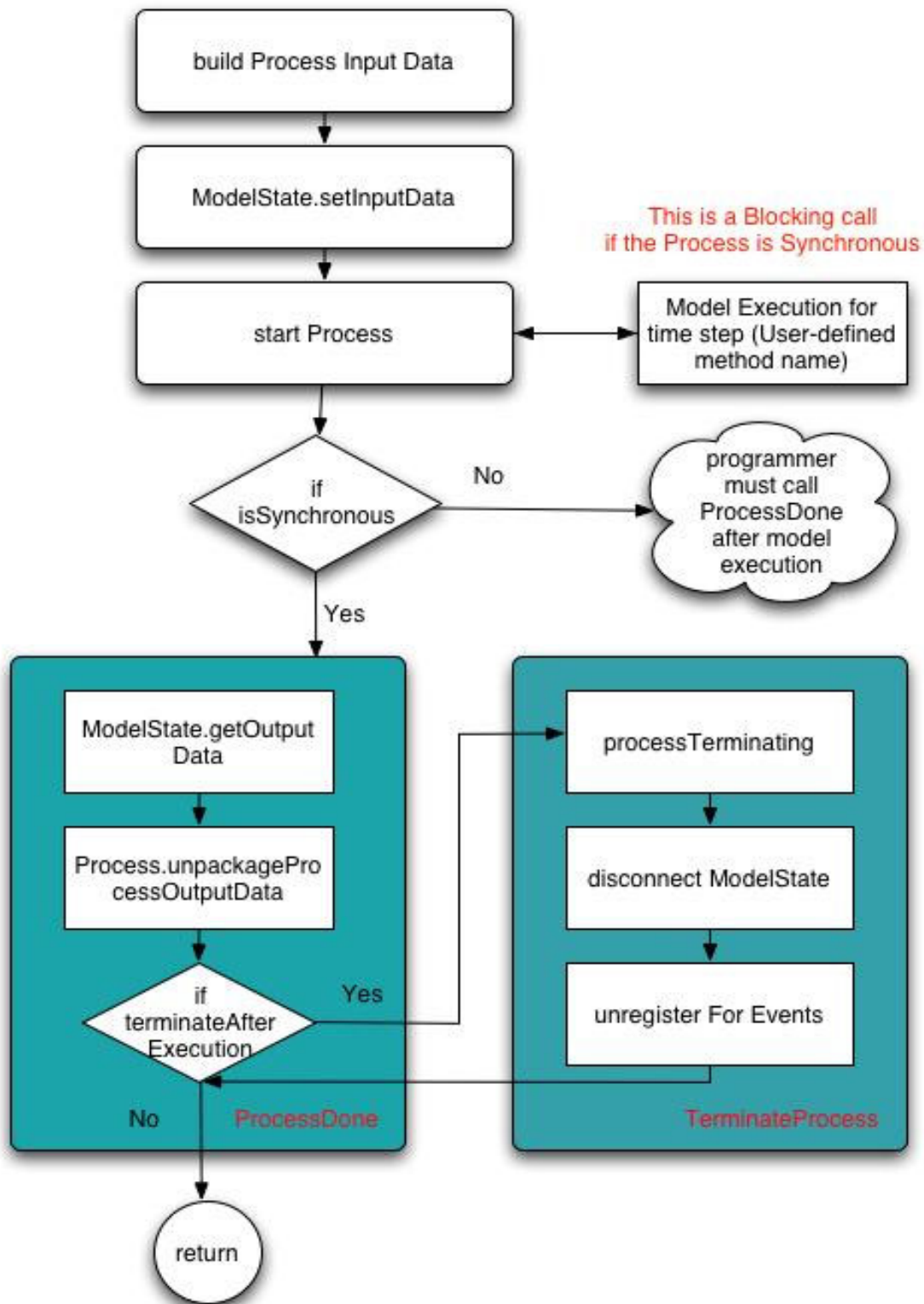


Figure 11 Flow Chart for Process Execution

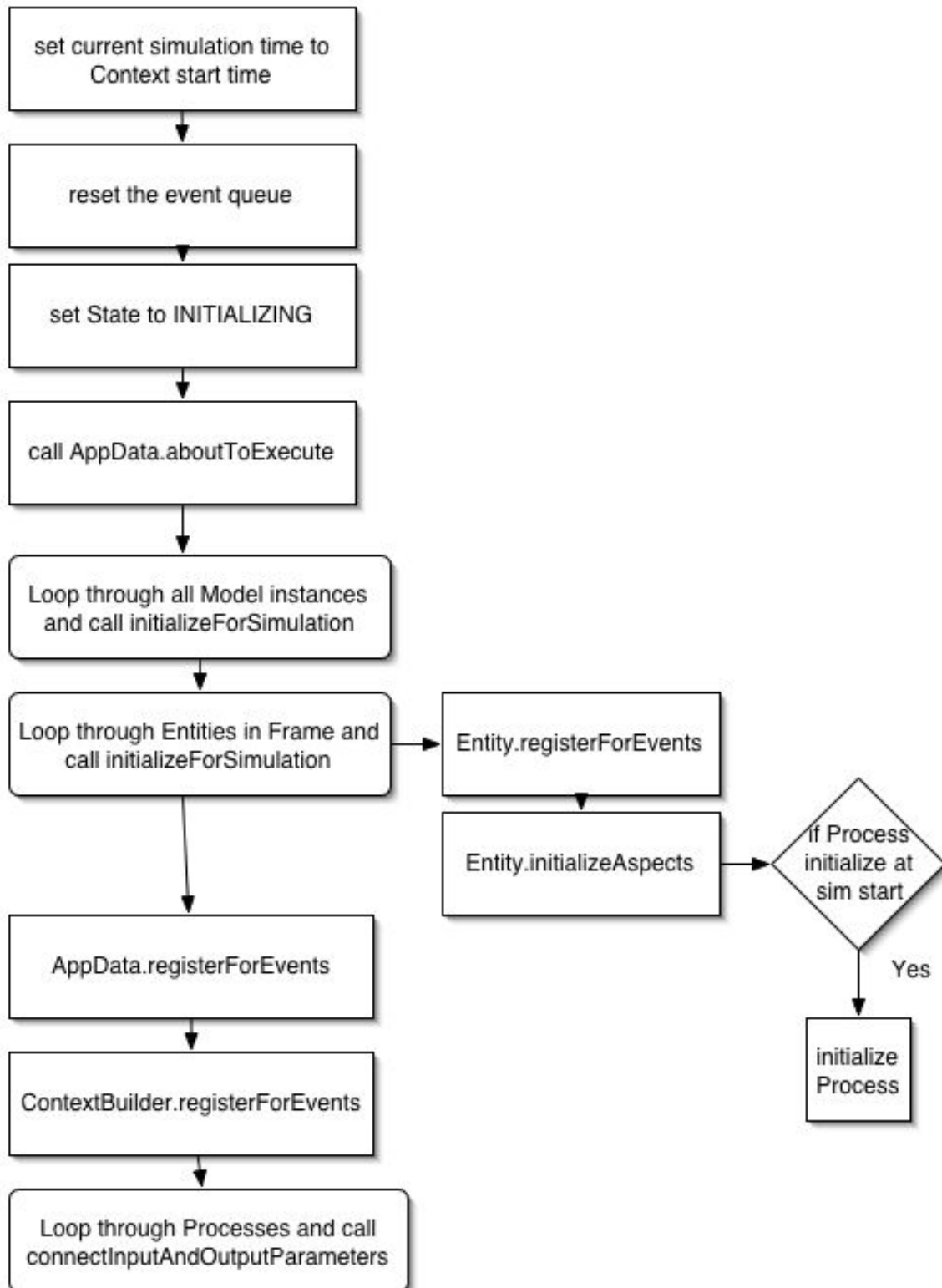


Figure 12 Flow Chart for the Simulation Initialization Process

simulation start, and if so, the method will call `Process.initializeProcess`. (Recall that the `initializeProcess` method has the process register for any events it is interested in and that it creates a new `ModelState` subclass.)

After the entities are initialized, the `ApplicationData` and `ContextBuilder` subclasses are allowed to register for any events they might be interested in.

Finally, the simulation manager will call `connectInputAndOutputParameters` on each process in the context frame.

### 3.7 Simulation Execution

Simulation execution processing is illustrated in Figure 13. Execution begins with the scheduling of the `simulationStarted` event at the context start time and the `simulationCompleted` event at context end time. The simulation manager then sets its run state to *running*.

The main simulation loop continues until the `simulationCompleted` event is reached or if the simulation is terminated by the user (or GUI). The simulation manager uses Java thread synchronization in the event loop to allow the GUI thread access to pause and/or resume the simulation execution thread. If the event loop has the lock, it retrieves the topmost event from the event queue and sets the simulation time clock to that event's time. The manager then delivers the event to all interested parties by looping through the registered objects for the event and calling the associated event handler. The `SimEvent` is passed into the handler so that the method has access to the timestamp, sender, and any data associated with the event.

If the simulation is terminated by the user or if the manager encounters the `simulationCompleted` event in the queue, then it breaks out of the loop and delivers the `simulationCompleted` event to all registered objects. All entities in a simulation register to receive the `simulationCompleted` event by default. The event handler in `Entity` cleans up loose ends for the entity by calling `terminateForSimulation` on the entity. If you override the `simulationCompleted` method in your subclass, you must call the superclass method, since this is where it calls `unregisterForEvents` and `terminateAspects` for the entity. The `terminateAspects` method calls each aspect's connected process and calls `terminateProcess` on it. This detaches the `Process` from the `ModelState` and calls `unregisterForEvents` on the process.

After any interested simulation objects process the `simulationCompleted` event, the simulation manager calls the `ApplicationData`'s `unregisterForEvents` method, sets the run state to *done*, and then calls the `ApplicationData`'s `simulationDone` method, where any application cleanup activities can be coded.

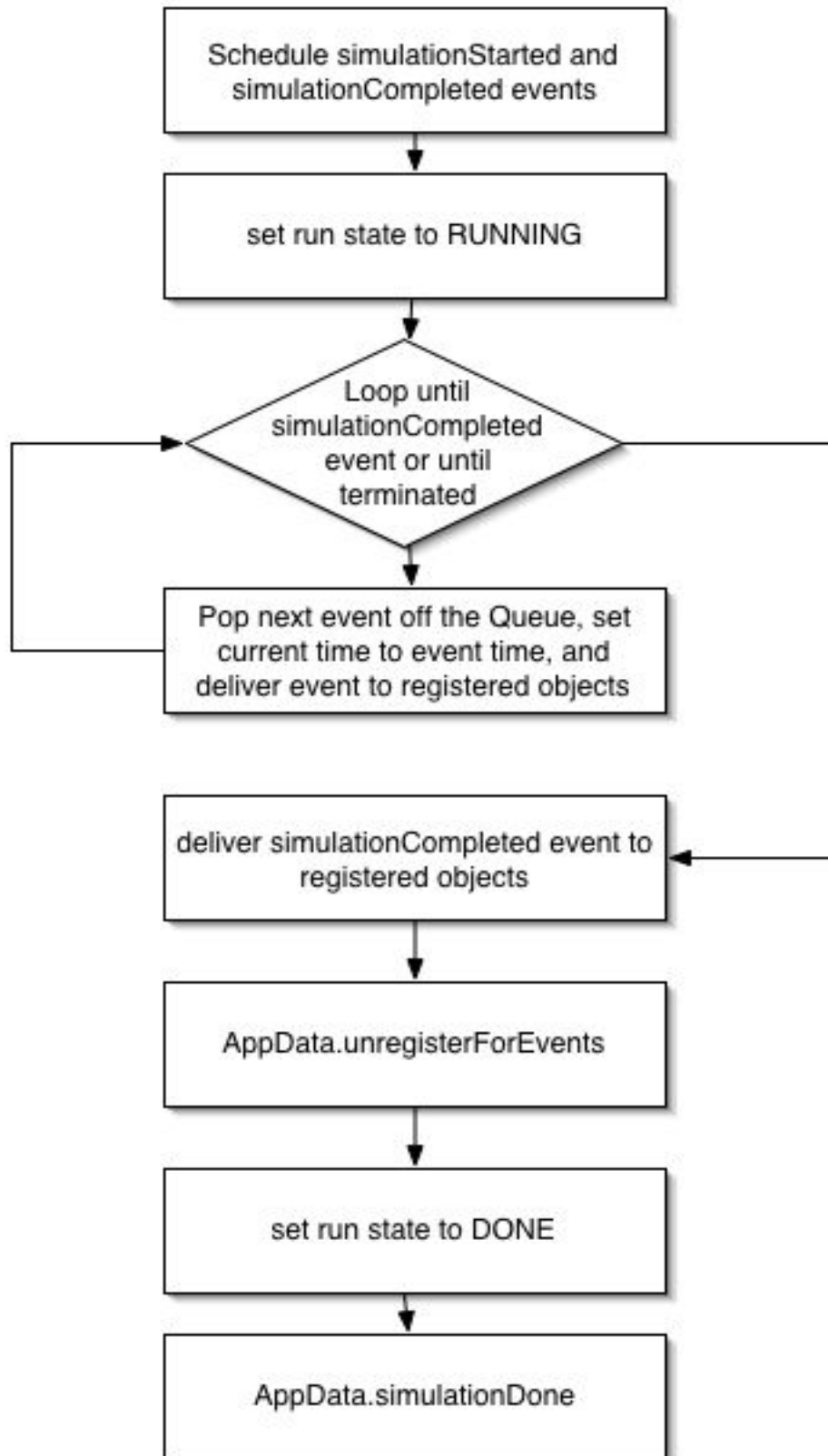


Figure 13 Flow Chart of Simulation Execution Processing

### 3.8 Running the Client

Figure 14 shows the flow chart for the DIAS client initiating a simulation. When a `AbstractSimulationClient` subclass is instantiated, an initialization method is called by the framework. The `initialize` method calls two abstract methods that need to be implemented by the subclass: `initializeSimSvrBroker` and `initializeSimParams`. `InitializeSimSvrBroker` sets up each component for running either fully distributed, within the same JVM, or some in-between combination (see Section 1.2 for discussion on distributed execution modes). The `initializeSimParams` method instantiates `anl.dias.simulation.input.SimulationParameters`, or a domain-specific subclass, and returns the object reference. It may also read any parameters from a file at this time or any other initialization parameters needed for the simulation.

The framework checks whether any part of the simulation is to be run remotely and, if so, instantiates an `RMIStateManager` and exports the client to the RMI registry.

Depending on whether the simulation is going to run “headless” or with a GUI, either the `initializeAndRun` method or the `startSimMgrWin` method would be called, respectively. Figure 15 shows the flow chart for running a simulation without a GUI, Figure 16 shows the simulation manager window, and Figure 17 shows the simulation manager execution logic. In Figure 15, the `AbstractSimulationClient.initializeAndRun` method starts by calling `initializeSimulationServer`. This is where the client requests a simulation server from the simulation broker. If the simulation server is running in the same JVM as the client, the singleton instance of `SimulationServer` is returned. Otherwise, the remote reference is returned to the client. The simulation server that gets returned has its time period set (start and stop times for the simulation), and the simulation parameters are passed from the client to the assigned server.

Next, the `AbstractSimulationClient.run` method is called. This calls the `SimulationServer.execute` method. If the server is being run remotely, it is a remote method call, if not, it is a local call. The server executes the simulation (described in the context build and context execution sections above), and when it is done, it calls the client’s `simulationDone` method. Again, if it is being run remotely, the call is a remote method call.

Finally, the `AbstractSimulationClient.cleanup` method is called. This releases the simulation server (for possible use for another client request) and, if running remotely, unexports the client from the RMI registry.

Figure 18 shows the flow of logic when the simulation server is released. It terminates the simulation, calls the abstract method `releaseData`, and sets its status as *available* (for possible use in another client request). The `releaseData` method is implemented by the simulation server subclass and is a “hook” that allows you to do any data cleanup (e.g., close or delete temp files that you may have used during the simulation).

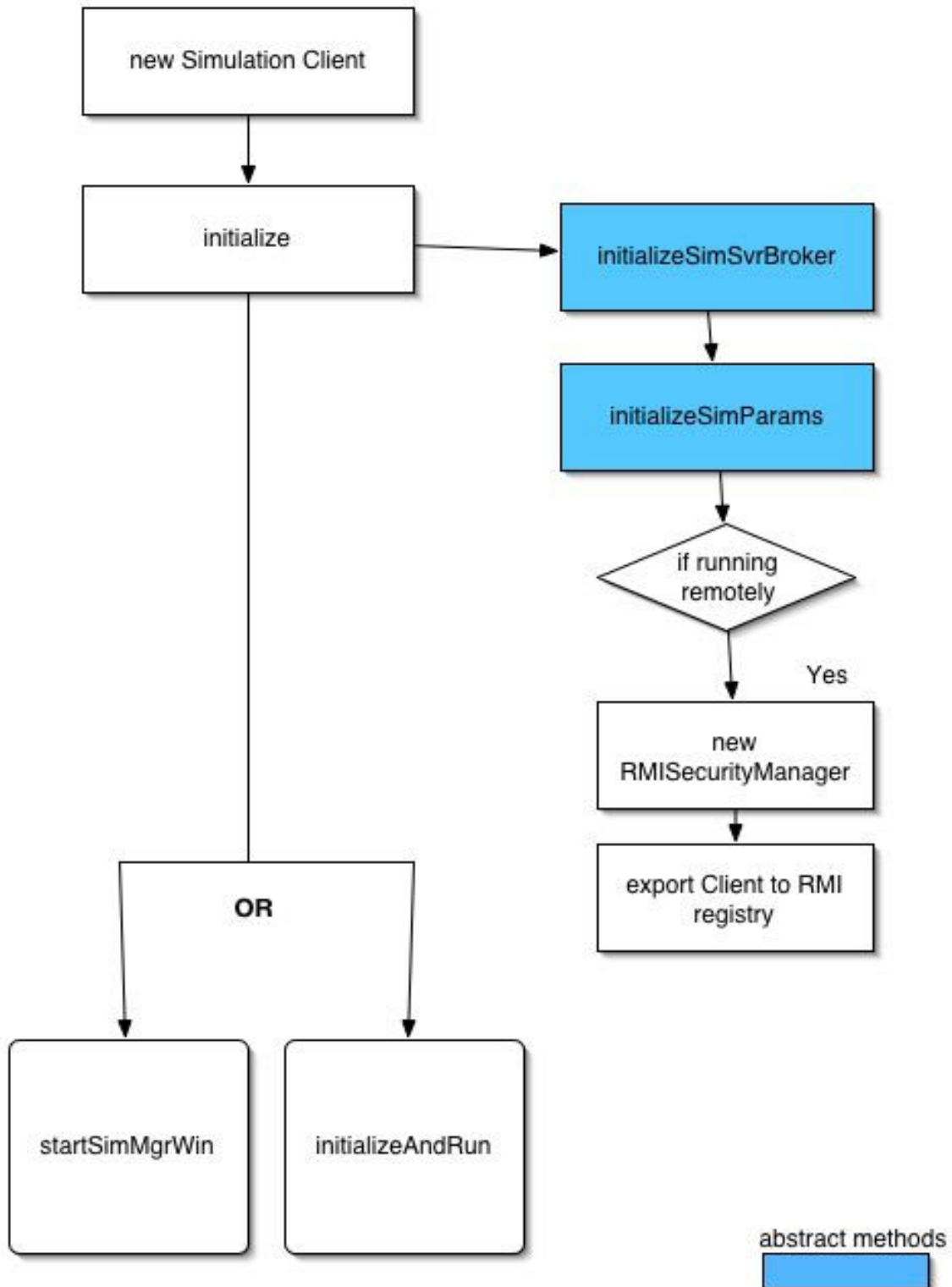


Figure 14 How a Client Initiates a Simulation Run

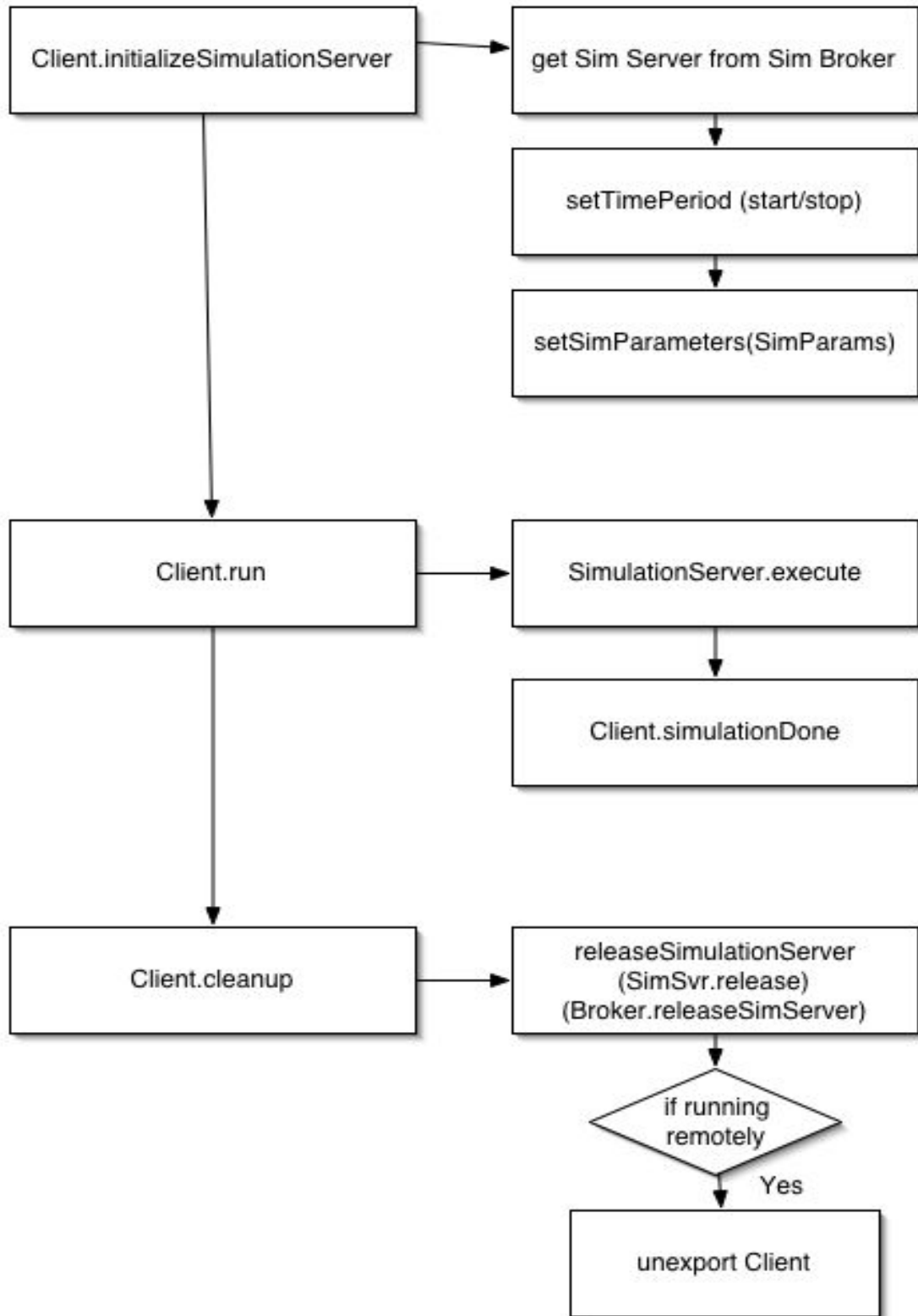
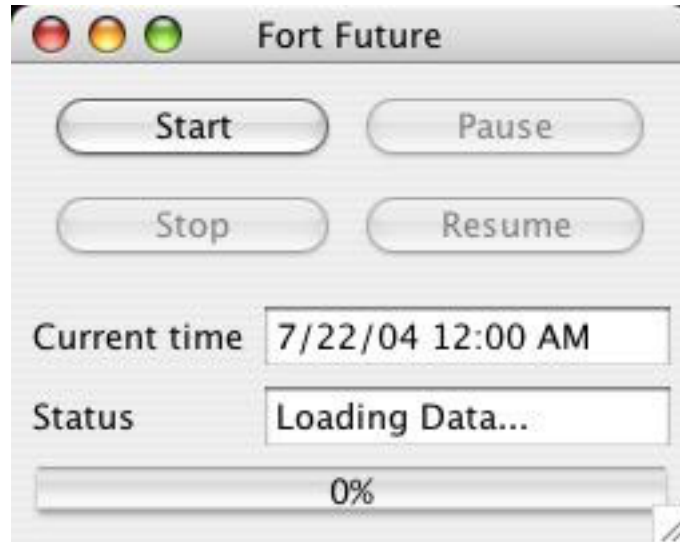


Figure 15 Initialize and Run Logic Without a Client GUI



**Figure 16 Simulation Manager Window**

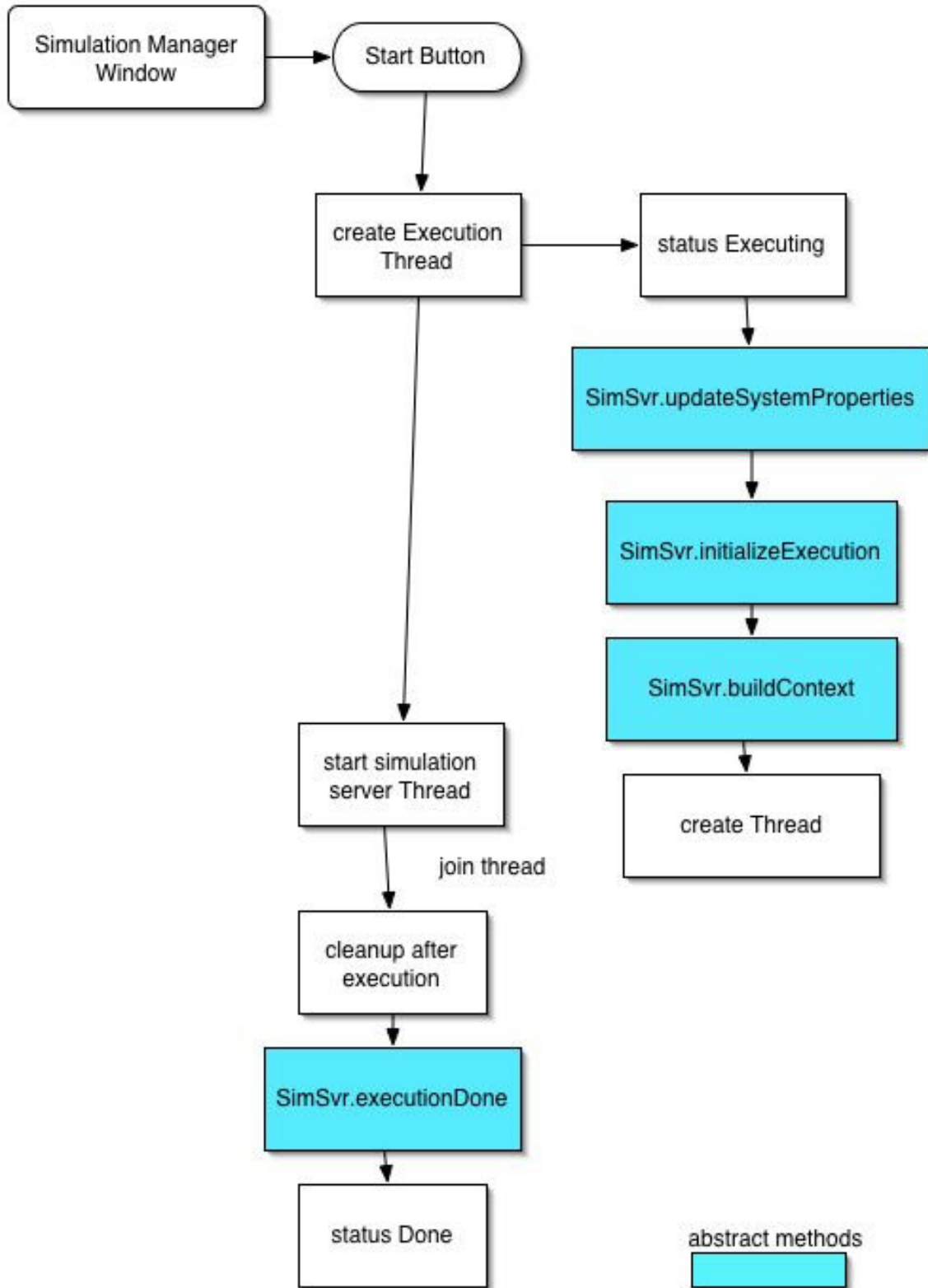
### 3.9 Controlling Execution with Simulation Manager Window

The simulation execution process can be controlled by the user via a GUI or within code. The four functions are:

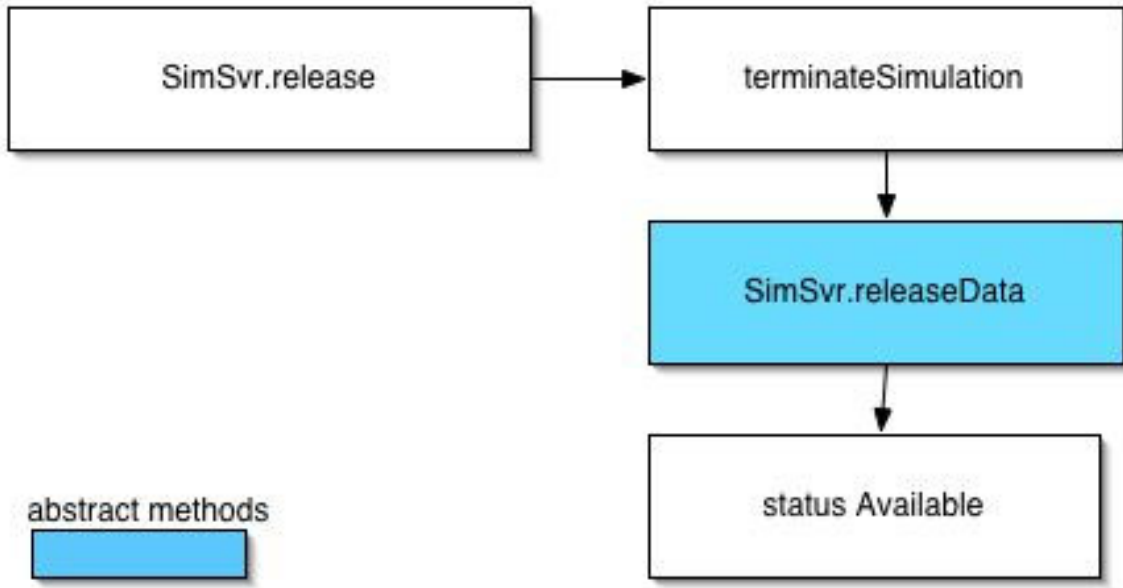
- Start simulation
- Pause simulation
- Resume simulation
- Stop simulation

Figure 16 is a snapshot of the default DIAS simulation manager window, showing the status, current time, a progress bar, and buttons to control the simulation execution.

Figure 17 shows the logic behind the execution that occurs when the “Start” button is pressed from the simulation manager window. The execution thread is created and the status is changed to *executing*. The abstract methods shown are called by the framework to your subclass’s implementation. `SimulationServer.updateSystemProperties` is a “hook” that lets you set any needed system properties before the simulation starts executing. The `SimSvr.initializeExecution` gives the subclass a chance to do any domain-specific setup before `buildContext` is called. The `SimSvr.buildContext` is where you set up a context builder and create a context (see Section 2.2.4, p. 19). Finally, the simulation manager thread is created.



**Figure 17 Logic for Execution of the Simulation via the Simulation Manager Window**



**Figure 18 Logic During Client Release of the Simulation Server**

After the execution thread is created, the simulation server thread is started and the simulation executes. The simulation manager window waits for the simulation server thread to finish, showing status, current simulation time, and progress in the window, as it is a listener to the simulation manager thread. Once the simulation is done, a cleanup method is called, which calls the `SimSvr.executionDone` method and sets the status to *done*. Again, the `executionDone` method is a “hook” to allow you to do any domain-specific cleanup after a simulation has completed.

### 3.10 Distributed Execution Flags

DIAS uses flags to indicate how the execution mode is to be set up. The `anl.util.system.PropertiesUtils.loadProperties` method loads Java properties defined in a comma-separated list of filenames and adds them to the system properties. Any property that can be set on the command line can be set within a file and loaded with this utility.

As discussed in Section 1.2, DIAS can be run fully distributed (Figure 2) or within a single JVM (Figure 3). Through the setting of two flags, a DIAS simulation can be partially distributed across JVMs. The possible combinations are illustrated in Table 1 and described below:

1. Fully distributed (as shown in Figure 2 and Rows 6–8 in Table 1),
2. Simulation broker with simulation server in one JVM, and simulation client in another JVM (Rows 2–3 in Table 1),

**Table 1 Distributed Execution Flag Settings**

JVM Processes	Run Sim Broker Remote Flag	Run Sim Server Remote Flag
Sim Broker, Sim Server, and Sim Client	False	False
Sim Broker and Sim Server	True	False
Sim Client	True	True
Sim Server	True	True
Sim Broker and Sim Client	True	True
Sim Broker	True	True
Sim Server	True	True
Sim Client	True	True

- Simulation broker with simulation client in one JVM, and simulation server in another JVM (Rows 4–5 in Table 1), and
- Single JVM (Figure 3 and Row 1 in Table 1).

If you want the simulation broker to bind into the RMI registry, you will need to set the simulation broker remote flag to *true*.

Example code for setting the distribution flags is located in the `initializeSimSvrBroker` method:

```
SimulationServerBrokerImpl.initialize(brokerRemote, serverRemote,
FarmTaxSimulationServerImpl.class, brokerURL);
```

The `brokerRemote` and `serverRemote` flags are boolean (Table 1), the `FarmTaxSimulationServerImpl.class` is the subclass of `SimulationServer` for the application, and the `brokerURL` is the export URI registered in the RMI registry if running remotely (e.g., `rmi://hostname:port/ApplicationSimulationServer`).

Along with the `DIAS PropertiesUtils`, we also use Java Developer Kit 1.4 Logging throughout DIAS. Logging should be set up from the main routine of each process that is to run in a JVM (client, server, and/or broker). At a minimum, call `Logger.getLogger("anl").setUseParentHandler(false)`, so the DIAS logging messages will *not* write to both the JVM system logger and the DIAS logger. All top-level logs for your application should not use the parent logger (system logger) if you want to maintain fine control over the logging in a simulation.

#### 4 References

Christiansen, J.H., 2000, "FACET: A Simulation Software Framework for Modeling Complex Societal Processes and Interactions," *Proc. 2000 Summer Computer Simulation Conference*, Society for Computer Simulation International, July 16-20, Vancouver, B.C., Canada.

Lurie, G.R., P.J. Sydelko, P.J., and T.N. Taxon, 2002, "An Object-Oriented Geographic Information System Toolkit for Web-Based and Dynamic Decision Analysis Applications," *J. Geographic Information and Decision Analysis*, **6**(2):108–116.

Simunich, K.L., P. Sydelko, J. Dolph, and J. Christiansen, 2002, "Dynamic Information Architecture System (DIAS): Multiple Model Simulation Management," *2<sup>nd</sup> Federal Interagency Hydrologic Modeling Conference*, Las Vegas, NV, Jul 28 – Aug 1.

## Appendix A: Farm Tax DIAS Example

To create an example application in DIAS, ANL staff worked with the Commonwealth Scientific and Industrial Research Organisation (CSIRO) of Australia to design a Farm Tax domain to model within the DIAS framework. The source code and documentation are available from ANL and usually distributed on the CD with this manual and the DIAS program. Readers may find it helpful to consult the Farm Tax program while reading this developer's guide.

The archive file contains the following files when uncompressed:

- readme.txt – explanation of files and how to execute the code
- build.xml – an ANT build file used to export project files from the Eclipse IDE to a delivery directory
- runMC.sh (.bat) – run script for executing the External Model Controller
- runFarmTax.sh (.bat) – run script for executing the DIAS client/server (which connects to the model controller when needed)
- FarmTaxModel.command – a Mac OS X executable for the Fortran code (called from the model controller)
- FarmTaxModel.exe – a Windows executable for the Fortran code
- docs - The Farm Tax model description, design notes, and figures
- src - contains the source code for the model; `farmtaxdemo-src.jar` contains the java source code
- FarmTaxModel.f90 – the Fortran source code for the external model
- javadoc – the javadocs for the Farm Tax demonstration program
- jar – the various jarfiles needed for the model and the rest of DIAS/JeoViewer
- lib – the JNI native libraries for macosx, linux, and win32 (solaris available)
- input – the input files for the model (crop and soil values)
- props – property files used by the run scripts (read into the simulation)
- log – directory containing the log files for each run script

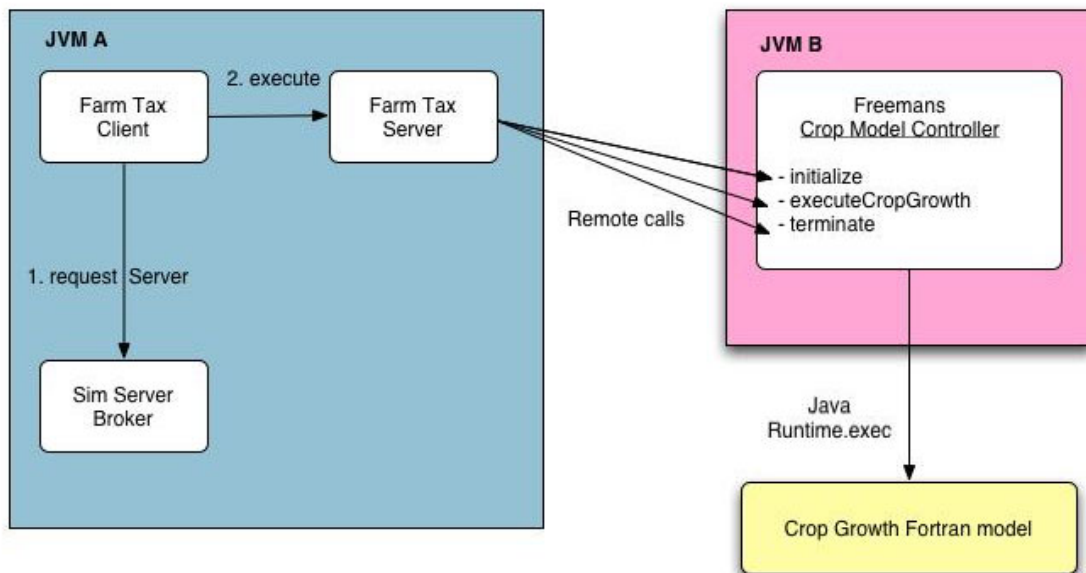
Figure A.1 shows how the Farm Tax demonstration is currently set up to run. The crop growth model is an external Fortran model that is used to implement the behavior of crop growth (see Table A.1). In this case, the DIAS client and server (as well as the simulation broker), are executed in a single JVM and the Freemans crop model controller is executed in another JVM (see Section 3.10).

The Fortran model is called using `Java Runtime.exec()` from the `executeCropGrowth` method in the model controller. The model controller methods are called from the Farm Tax server as remote method calls since the `anl.csiro.farntax.model.GrowModel` is implemented as an RMI external model (see Section 2.3.2 External Models, p. 25).

The `docs/FarmTax EAPM.xls` file is the design document for the Farm Tax domain. It is a spreadsheet that delineates the entity classes, the E-A-P-M connections, attributes, process inputs/outputs, and events. This type of template is a good way to organize how a simulation domain can map into the DIAS framework. After the templates are filled out, you can start coding the application from it.

Table A.1 corresponds to Sheet 1 in the spreadsheet document and shows all the entities in the demo: atmosphere, crop, farmer, government, and bank customer. Each entity has some attributes (modeled as parameters) and most have aspects. The process and models are listed as well and the Java class names correspond to the names in the table. All but one of the models are internal, that is, written in Java inside DIAS; the Grow Crop model is external (Grow Crop).

Table A.2 corresponds to Sheet 2 and shows the process input and output parameters for the various models. The input and output parameters of processes should all be accessible from either regular domain objects or entities in the simulation.



**Figure A.1 Farm Tax Execution Mode**

**Table A.1 Farm Tax E-A-P-M Design**

<b>Entity Name</b>	<b>Entity Attributes</b>	<b>Entity Aspects</b>	<b>Process</b>	<b>Model</b>	<b>Model Type</b>
Atmosphere	annualRainfall	precipitation	PrecipitationProcess	PrecipitationModel	internal
Crop	field	grow	GrowProcess	GrowModel	external
	profitMargin yield	harvestCrop	HarvestCropProcess	HarvestCropModel	internal
Farmer	bankBalance	subsidise	SubsidiseProcess	SubsidiseModel	internal
	taxRate				
	fields crops				
Government	revenue	none	none	none	
	revenueCap				
	farmers				
	freeTradeAct				
Bank Customer	name	getCash	WithdrawCashFromATMProcess	WithdrawCashFromATM	COA
	wallet				
	bank				
	minimumCashOnHand				
	resourceManager				
	participant				
	departureTime				

**Table A.2 Process/Model Input/Output Parameters**

<b>Entity Name</b>	<b>Process Input Parameters</b>	<b>Process Output Parameters</b>	<b>Models</b>	<b>Model Object Method</b>
Atmosphere		currentRainfall	PrecipitateModel	calculatePrecipitation
Crop	currentRainfall soil type percent clay required rainfall soil % clay crop type	yield	GrowModel	growCrop
Crop	yield	profitMargin	HarvestCropModel	harvestCrop
Farmer	revenueCap bankBalance taxRate	bankBalance taxRate subsidy	SubsidiseModel	subsidiseCrop

Table A.3 is a list of all the simulation events that can be scheduled within this application. It is provided because the method names are just Java strings in the code that are looked up via Java Reflection, and it is hard to find them all by searching the code. This table lists the sender of the type of event and any optional data that may be associated with the event. Table A.4 augments Table A.3 in that it lists which entities/objects have registered to receive events and which events the entities/objects can schedule.

Not shown in this document are the model state data map sheet and the COA sheet from the spreadsheet document. The model state table lists the input and output data held in the **ModelState** object for the various models. It also lists the remote objects that transport the input and output data to the external model. Lastly, the COA sheet lists the COA elements for the example internal COA model for the application. This template is used to set up the step names, durations, timeout parameters, participants, role type, and resources needed for each step in the COA (see Section 2.3.3, p. 30)

Figure A.2 represents a sequence diagram for the Farm Tax application. Each row consists of an entity or object that can schedule an event, receive an event (via an event handler), or perform an aspect. Simulation time is the horizontal axis in the diagram. The time step for the simulation is yearly. The **ApplicationData** subclass receives the **simulationStarted** event and is the timekeeper of the system. It schedules the application-specific event, **newYear**, to start the

**Table A.3 Event Data Map**

<b>Event</b>	<b>Event Data</b>	<b>Event Sender</b>
simulationStarted	none	SimulationManager
newYear	none	FarmTaxAppData
doneGrowing	none	Crop
doneHarvesting	none	Crop
payTaxes	taxes	Farmer

**Table A.4 Entity/Process Event Map**

<b>Entity/Object</b>	<b>Event Handler</b>	<b>Event Scheduled from</b>	<b>Action</b>
Atmosphere	newYear	anyone (AppData)	Aspect->precipitation
Crop	newYear	anyone (AppData)	Aspect->grow
	doneGrowing	self	Aspect->harvestCrop
Farmer	newYear	anyone (AppData)	schedule event->getCash Aspect->subsidize
	doneHarvesting	the Farmer's Crops	schedule event -> payTaxes
Government	payTaxes	anyone (all Farmers)	
FarmTaxAppData	simulationStarted	SimulationManager	
	newYear	anyone (self)	schedule event->newYear
	simulationCompleted	SimulationManager	

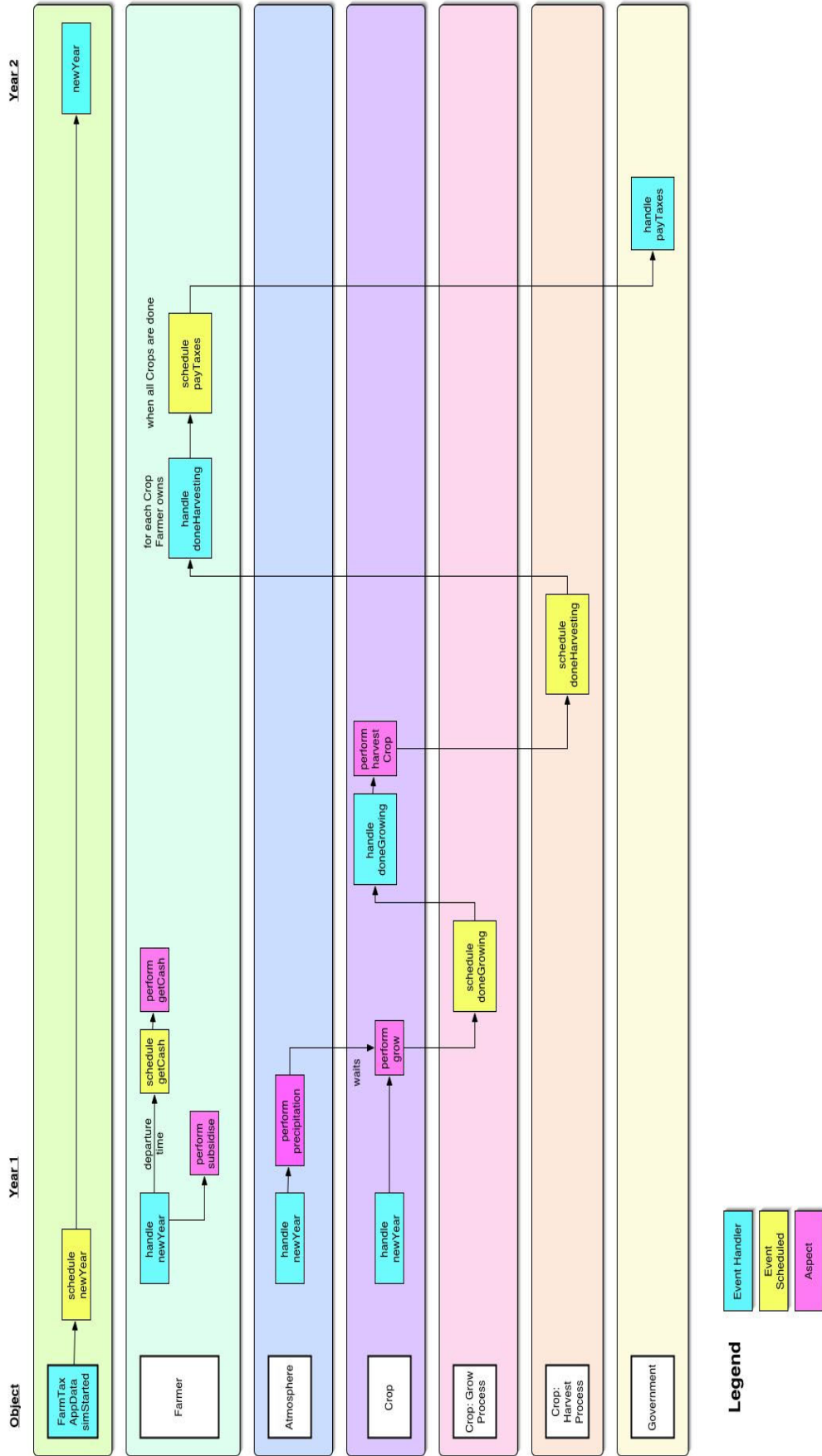


Figure A.2 Event Sequence for Farm Tax Demonstration Program

simulation. Since it registered to receive the `newYear` event, its event handler will be called and it will subsequently schedule the next `newYear` event to occur one year from the current timestamp.

The `props/init.properties` file provides input to the application and sets the number of ATMs in the simulation to 1 (to show resource contention) and the number of farmers to 2. The number of fields for a farmer is randomly chosen between 1 and 5 for each farmer (in `anl.csiro.farntax.simulation.FarmTaxContextBldr.createInitialEntities`). A single crop type is grown in each field.

The farmer, atmosphere, and crop entities also are registered to receive the `newYear` event. The order in which each object processes the event is undefined, so if your application needs to make sure a specific order is attained, you will either need to use a different event name or schedule it using a higher priority. In this case, the order of processing does not matter.

The atmosphere object (a singleton in the demonstration) performs its precipitation aspect in response to the `newYear` event. The precipitation process/model will run and the year's annual rainfall is calculated.

Each crop object will get the `newYear` event and activate its grow aspect, but the actual aspect will not start until the precipitation aspect has finished. We used the parameter triggering mechanism (see Section 3.4, p. 38) to make the grow aspect wait until the annual rainfall parameter is set each year before it starts (using a custom parameter-ready method). As part of the crop growth process execution, it schedules the `doneGrowing` event to signify when a crop has finished processing and set its parameters. The corresponding crop object had registered to receive `doneGrowing` events for itself (it is not interested in all `doneGrowing` events for every crop). The `doneGrowing` event handler for crop then calls `performAspect` to harvest the crop. When the crop harvest process is done, it schedules the `doneHarvesting` event. Each farmer had registered for the `doneHarvesting` event for each of his crops that he is managing. It keeps a tally of the yield of each crop and when the last one is done harvesting for the year, it schedules a `payTaxes` event.

The government singleton has an event handler for the `payTaxes` event. It is registered to receive any `payTaxes` from any Entity in the simulation. It also tallies the yearly revenue.

Each farmer in the simulation also handles the `newYear` event. Two things are triggered for the farmer: perform subsidize aspect and schedule a `getCash` event. The subsidize aspect is another internal model that calculates a subsidy for this year's crop. The `getCash` event is scheduled for some departure time (calculated at simulation initialization time). When the simulation clock gets to that time, the `getCash` event handler for the farmer calls the `getCash` aspect. The `getCash` aspect is the COA type of internal model and its execution is discussed in Section 2.3.2 (p. 30). The javadoc documentation is included with the source code and execution scripts. We suggest you read `readme.txt` as well as the code and javadoc before running the Farm Tax DIAS example application.

The demonstration code was intended to show a developer the basic elements of a DIAS simulation. The decision to write each model as an internal or external model was arbitrary and one external model was chosen to show how that connection works. All the models could have been implemented as internal or external models in this example; however, the rule of thumb is that external models are only wrapped with a model controller if it exists already or is in a language other than Java.

The results from running the demonstration code are print statements that write to the console. The user may follow the path of the simulation by referring to Figure A.2 and reading through the print log file (located in the run directory `log/farmtax.log`). The file `props/debug.properties` sets the logging levels for print statements. Logging shows the steps in creating the initial entities in the context builder, when the simulation starts and ends execution, when events (such as `newYear`) are processed by entities, and the duration and decisions within the steps of the `WithdrawCashFromATM` COA. Each run will be different due to the random draws done within the COA and the rand choice of crop type for each farmer.

## Appendix B: DIAS Exceptions

This appendix presents and explains various exception messages that may occur during simulation development. The exception is first presented in a monospaced font, followed by the explanation in a normal typeface. All examples are from the Farm Tax demonstration.

```
***--- Exception -----***
```

The runtime exception:

```
java.lang.IllegalArgumentException:
```

```
ParameterReadyMethod:
```

```
anl.csiro.farmtax.model.GrowProcess.parameterReadyIfCurrent (ParameterR
eadyData prd) does not exist
```

has occurred in the static method :

```
anl.csiro.farmtax.model.GrowProcess>>initializeInputParameters
which was called during the static initialization
of the classanl.csiro.farmtax.model.GrowProcess
```

```
***-----***
```

This runtime exception is thrown when the framework does not find the `parameterReadyIfCurrent` method in the `GrowProcess` subclass when the `initializeInputParameters` declared that this was the parameter-ready method for annualRainfall. The existence of the method is checked at runtime, when the `addMetalInputParameter` method is executed.

```
***--- Exception -----***
```

```
SEVERE: RuntimeException occurred during execution on simulation
server anl.csiro.farmtax.simulation.FarmTaxSimServerImpl@c9a690
```

```
anl.dias.core.NoSuchAspectException: Aspect: [precipitation] not found
for entity class: [class anl.csiro.farmtax.entities.Atmosphere]
```

```
***-----***
```

This exception is caused by not including the precipitation aspect in `Atmosphere`. `initializeMetaAspects` and declaring it as needed within `FarmTaxContextBldr`. `getModelsForRegion`.

```

***--- Exception -----***
SEVERE: RuntimeException occurred during execution on simulation
server anl.csiro.farntax.simulation.FarmTaxSimServerImpl@7c4c51
anl.dias.core.NoSuchAspectException: Aspect: [subsidise] not found for
entity class: [class anl.csiro.farntax.entities.Farmer]
***-----***

```

This runtime exception is caused by a mistake in the spelling of an aspect name between `Farmer.initializeMetaAspects` and `FarmTaxContextBlDr.getModelsForRegion`.

```

***--- Exception -----***
SEVERE:
anl.util.system.MissingRequiredMethodException: Method: public static
void initializeInputParameters() does not exist in class
anl.csiro.farntax.model.HarvestCropProcess
SEVERE:
anl.util.system.MissingRequiredMethodException: Method: public static
void initializeOutputParameters() does not exist in class
anl.csiro.farntax.model.SubsidiseProcess
***-----***

```

This missing method exception is caused by forgetting to implement the static method in the subclass.

```

***--- Exception -----***
Nov 2, 2004 2:43:31 PM
anl.dias.simulation.distributed.AbstractSimulationClient run
SEVERE: RuntimeException occurred during execution on simulation
server anl.csiro.farntax.simulation.FarmTaxSimServerImpl@c9a690
anl.dias.core.NoSuchParameterException: [Farmer 0]: taxRate
***-----***

```

This runtime exception is caused by not having an `addMetaParameter("taxRate")` in `Farmer.initializeMetaParameters` when the parameters were getting instantiated and the `SubsidiseProcess` declared it as a needed `metaInputParameter`.

```

***--- Exception -----***
SEVERE: An exception occurred during simulation initialization
SEVERE: anl.dias.core.NoSuchParameterException: Desmodium- silverleaf:
atmosphere
SEVERE:      at
anl.csiro.farntax.model.GrowProcess.connectInputAndOutputParameters (Gr
owProcess.java:185)
***-----***

```

This invalid parameter exception is caused by trying to access a crop parameter that does not exist.

```

***--- Exception -----***
The runtime exception:
  java.lang.NullPointerException
has occurred in the static method :

anl.csiro.farntax.model.HarvestCropProcess>>initializeInputParameters
which was called during the static initialization
of the classanl.csiro.farntax.model.HarvestCropProcess
***-----***

```

This exception is caused by calling `addMetaOutputParameter` instead of `addMetaInputParameter`.

```

***--- Exception -----***
Nov 2, 2004 3:04:53 PM anl.dias.simulation.SimulationManagerImpl run
SEVERE:
An exception occurred during simulation initialization
anl.dias.simulation.EventHandlerNameNotFoundException
Entity event handler method:
anl.csiro.farntax.entities.Atmosphere.>>newYear(anl.dias.simulation.Si
mEvent)
***-----***

```

This initialization exception is caused by not having the correct event handler method signature for the `newYear` event that it was attempting to register for. If an entity declares that it wants to receive an event in its `registerForEvents` method, for example:

```
simMgr.registerForEvent(this, "newYear");
```

then it must implement the event handler with the signature:

```
public void newYear(SimEvent event)
```

The method existence is checked at runtime when the `registerForEvents` method is executed.

```
***--- Exception -----***
SEVERE: RuntimeException occurred during execution on simulation
server anl.csiro.farntax.simulation.FarmTaxSimServerImpl@62e295
SEVERE: anl.dias.core.ParameterNotInstantiatedException: BankCustomer
[Farmer 0]: fields
SEVERE:      at
anl.csiro.farntax.simulation.FarmTaxContextBlDr.buildContext(FarmTaxCo
ntextBlDr.java:84)
***-----***
```

This exception is caused by using `setValue("fields")` for the farmer class within `createInitialEntities` without declaring `addParameterDependency` before setting the value (it was not declared in any of the `MetaInput/OutputParameters` for the processes being used; therefore, it has to be declared with the `addParameterDependency` method).

```
***--- Exception -----***
SEVERE: An exception occurred during simulation initialization
SEVERE: anl.dias.simulation.EventHandlerNameNotFoundException
Entity event handler method:
anl.csiro.farntax.entities.Government>>newYear(anl.dias.simulation.Sim
Event)
***-----***
```

This last exception is caused by the government entity registering to receive the `newYear` event, but without implementing the event handler.