SNAP/RADTRAD 5.0

Software Architecture Document

ISL-ESD-TR-20-01

Nolan Bartlow

INFORMATION SYSTEMS LABORATORIES

Vienna, VA and Idaho Falls, ID

2/4/2020

CONTENTS

1 Introduction

The purpose of this report is to provide a simplified architectural picture of the SNAP/RADTRAD source code. The term software architecture refers to the structure of a code or software system. This report documents the RADTRAD software architecture utilizing two perspectives. These perspectives are called the process view (see Section 2) and the development view (see Section 3), and are widely applied for documentation purposes in the software industry. The process view describes the major system methods and functions, and how they interact and communicate during runtime. The development view is more of an index which describes how the system is laid out, component by component, by illustrating the how the RADTRAD packages are organized.

1.1 Definitions

The SNAP/RADTRAD code is written in the Java programming language. The following provides definitions for some of the major features and structures of Java that relate to the RADTRAD code.

package: a group of related classes and interfaces.

class: a template for creating an object.

object: an instance of a class.

method: a piece of code that performs some function and can be invoked by name.

interface: a completely abstract class that defines a group of related methods. These methods are implemented in classes that explicitly *implement* the interface.

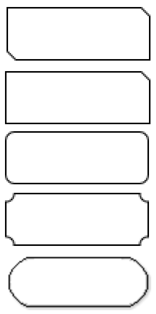
constructor: a special method in a class that initializes a newly created object.

field: a variable inside of a class.

For information on these and other Java concepts, see the Java Glossary at Oracle.com.

1.2 Diagrams

This document often uses diagrams to illustrate the code architecture. The follow provides definitions for the diagram components used.



– folder

– package

– class

– interface

– method

2 Process View

This section describes the RADTRAD code architecture from the process view perspective. The process view diagram shown in Figure 2-1 illustrates how RADTRAD functionally operates from the start to the finish of a case execution. The eight numbered steps represent the **radcalc** method. The radcalc method is the driving method for all calculational aspects of RADTRAD. Primarily, it performs steps one through eight iteratively until the final time step is reached, at which point the radcalc method ends. The steps illustrated by Figure 2-1 are described briefly below.

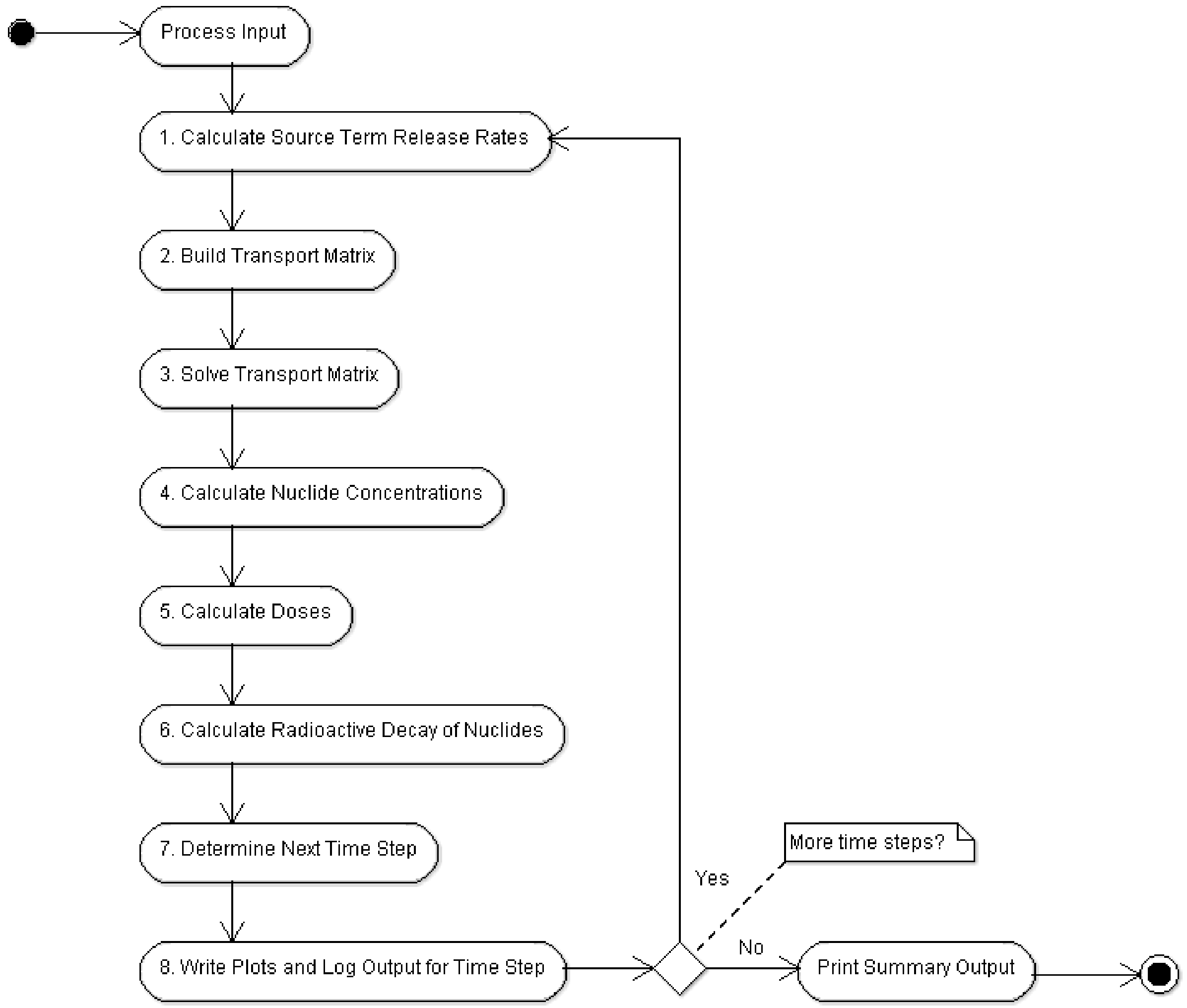

Process Input
1. Calculate Source Term Release Rates
2. Build Transport Matrix
3. Solve Transport Matrix
4. Calculate Nuclide Concentrations
5. Calculate Doses
6. Calculate Radioactive Decay of Nuclides
7. Determine Next Time Step
8. Write Plots and Log Output for Time Step
More time steps?
Yes
No
Print Summary Output

Figure 2-1 Simplified process of one RADTRAD code execution

Process Input (Section 2.1): The first step of the RADRAD procedure is to read in the input provided by the user and to translate the model by mapping the components to objects that can be easily accessed throughout the procedure. The input processing occurs in three main steps: 1) parsing the command line options, 2) mapping the components from the main input file, and 3) reading in the additional source term data and the data pertaining to nuclides. This process is described in greater detail in Section 2.1.

Construct and Solve System Transport Matrix (Section 2.2): This step covers steps one through four of the radcalc method, as illustrated by Figure 2-1. At this point in the RADTRAD procedure, the radcalc method – the main RADTRAD calculational method – is active. The first major calculational step of the radcalc method is to build the system transport matrix for the nuclide transport matrix differential equation. During this routine, transport coefficients are calculated using transport mechanisms that are selected according to the user provided transport components, such as pipes, filters, and leak rate tables. This process and its various steps are described in greater detail in Section 2.2

Calculate Doses (Section 2.3): This step covers step five of the radcalc method, as illustrated by Figure 2-1. After the system transport matrix is constructed, the radcalc method begins the process to calculate doses, which is the primary purpose of a RADTRAD execution. This process is described in greater detail in Section 2.3.

Prepare for Next Time Step (Section 2.4): This step covers steps six and seven of the radcalc method, as illustrated by Figure 2-1. Once doses have been calculated, RADTRAD must perform two major additional steps before the next iteration of radcalc can begin: 1) accounting for radioactive decay and daughtering, and 2) determining the size of the next time step. This process is described in greater detail in Section 2.4.

Print Output (Section 2.5): This step covers step eight of the radcalc method as well as the final step of a simulation, as illustrated by Figure 2-1. As calculations are performed in the radcalc method, results are printed to the output files. This process is carried out by various output methods and is described in greater detail in Section 2.5.

2.1 Process Input

This section describes the first step of the RADTRAD simulation, as illustrated in Figure 2-1. This process can be visualized in three steps, as shown in Figure 2-2.

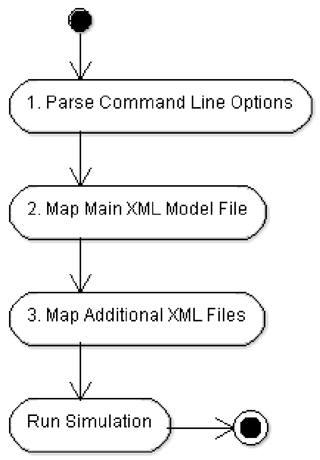

1. Parse Command Line Options
2. Map Main XML Model File
3. Map Additional XML Files
Run Simulation

Figure 2-2 Illustration of the three RADTRAD input processing steps

These steps are described in the subsections below.

2.1.1 Parse Command Line Options

If run from the command line, the first process that RADTRAD carries out is to parse the command line options provided by the user. For a full list of these options, see Section 3.1 of Reference [3].

2.1.2 Map Main XML Model File

The main RADTRAD input file is primarily processed by the **mapModel** method. This method is called at the beginning of a RADTRAD execution before any calculations are performed. The purpose of the mapModel method is to read and process the PSX file (main input file) provided by the user. The PSX file contains most of the data in the user constructed transport model, along with all model options. The mapModel method performs this by calling eight unique methods that perform input processing for each type of component in the model. Three of these methods – the **mapModelingParameters** method, the **mapCompartments** method, and the **mapPaths** method – call additional mapping methods to process specific components of the original component.

Table 2-1 Methods for Reading and Mapping PSX Input

Method Name	Description
mapModel	This method is responsible for processing the input from the PSX file and mapping the model components to objects that can be accessed later in the RADTRAD procedure. The structure of each component object is described in greater detail in Section 3.2. This method calls the below methods in order to read the PSX file and construct the model.
mapModelingParameters	This method is responsible for processing the many general model options provided by the user. This method calls the four methods below (mapModelParameters, mapSimulationParameters, mapOutputParameters, mapNrcOutputParameters) to process four different types of general model input. This input is mapped to the highest-level component object: the Model.java class object described in Section 3.2.1
mapModelParameters	This method is responsible for processing the general model and simulation parameters such as the start and duration of the accident, the onset of the gap release, the plant power level, and the atmospheric dispersion inflow/outflow pathway information. General model information is mapped to the Model.java class object and simulation specific model information is mapped to the Simulation.java class object. Each of these classes are described in Sections 3.2.1 and 3.2.8, respectively.
mapSimulationParameters	This method is responsible for processing simulation and timestep options such as the timestep algorithm type and the maximum timestep error. Additionally, this method sets up the output frequency table. This information is mapped to the Simulation.java class object, described in Section 3.2.8.
mapOutputParameters	This method is responsible for processing various output options. If an output option is set in the input, this method sets a flag to print this option later. For example, if the flag "showModel" is set to true in the PSX file, a flag will be set to call the print_model method to print detailed model information at the time of output. This information is mapped to the Simulation.java class object, described in Section 3.2.8. See Section 2.4 for a more detailed description of the main RADTRAD output methods.
mapNrcOutputParameters	This method is responsible for processing various NRC output options, such as the input echo option.
mapNuclideDatabase	This method constructs an instance of the NuclideReader class which is later used with the read method to process the NIX file, as described in Section 2.1.3.
mapSources	This method is responsible for processing miscellaneous source input such as source term fractions and iodine type. This information is mapped to the Model.java class object described in Section 3.2.1. Additionally, this method

Method Name	Description
	constructs an instance of the SourceReleaseReader class and the InventoryReader class which are later used with the read method to process the SRX and ICX files, as described in Section 2.1.3.
mapCompartments	This method is responsible for processing user input information for each compartment such as volume and type. This information is mapped to the Compartment.java objects described in Section 3.2.2. This method calls the three methods below (mapNaturalDeposition, mapSpray, mapCompartmentFilter) to process additional compartment components.
mapNaturalDeposition	This method processes the natural deposition removal rate vs. time table data for compartments with user specified natural deposition tables. This information is mapped to the Deposition.java objects described in Section 3.2.11.
mapSpray	This method processes the compartment spray removal rate vs. time table data for compartments with user specified spray tables. If the Powers model is specified for elemental iodine, this method processes the flux and height vs time table data along with the user specified Powers percentile value. This information is mapped to the Spray.java objects described in Section 3.2.10.
mapCompartmentFilter	This method processes the filter flow rate and filter efficiency vs. time table data for compartments with user specified filter tables. This information is mapped to the Filter.java objects described in Section 3.2.9.
mapDoseLocations	This method is responsible for processing all information associated with the dose locations identified by the user, such as the dose location name and compartment identifier. This method also processes the breathing rate vs. time table data and the occupancy factor vs. time table data if these components are specified by the user. This information is mapped to the DoseLocation.java objects, the Breathing.java objects, and the Occupancy.java objects described in Sections 3.2.4, 3.2.16, and 3.2.17, respectively.
mapChiQTables	This method is responsible for processing the atmospheric dispersion factor (chi/Q) vs. time table data that is associated with environment dose locations. This information is mapped to the AtmosphericDispersion.java objects described in Section 3.2.5.
mapPaths	This method is responsible for processing user input information for each pathway such as type and pathway location (which two compartments the pathway lies between). This method calls the eight methods below (mapFilterTable, mapPathPipingOrganicModel, mapPathPipingElementalModel, mapPathPipingAerosolModel, mapPathNobleGasModel, mapPathLeakTable, mapPathBrockTable, mapGenericTable) to process additional compartment components and to process the transport type between compartments. For example, if the pathway is identified as a pipe with user specified table data for each iodine transport group, the three mapPathPiping methods will be called to process this data. This information is mapped to the Pathway.java objects described in Section 3.2.3.
mapFilterTable	This method processes the filter flow rate and filter efficiency vs. time table data for pathways with user specified filter tables. This information is mapped to the Filter.java objects described in Section 3.2.9.
mapPathPipingOrganicModel	This method processes the flow rate and decontamination factor vs. time table data for pathways identified as pipes with a user specified transport

Method Name	Description
	mechanism for the organic iodine transport group. This information is mapped to the Pipe.java objects described in Section 3.2.13.
mapPathPipingElementalModel	This method processes the flow rate and decontamination factor vs. time table data for pathways identified as pipes with a user specified transport mechanism for the elemental iodine transport group. This information is mapped to the Pipe.java objects described in Section 3.2.13.
mapPathPipingAerosolModel	This method processes the flow rate and decontamination factor vs. time table data for pathways identified as pipes with a user specified transport mechanism for the aerosol iodine transport group. This information is mapped to the Pipe.java objects described in Section 3.2.13.
mapPathNobleGasModel	This method processes the flow rate vs. time table data for pathways identified as generic with a user specified flow rate for the noble gas transport group. This information is mapped to the Flowpath.java objects described in Section 3.2.12.
mapPathLeakTable	This method processes the flow rate vs. time table data for pathways identified as air leakage paths. This information is mapped to the Flowpath.java objects described in Section 3.2.12.
mapPathBrockTable	This method processes the flow rate vs. time table data for pathways identified as pipes with a user specified transport mechanism for the aerosol iodine transport group if the Brockmann model is selected. This information is mapped to the BrockmannPipe.java objects described in Section 3.2.14.
mapGenericTable	This method processes the flow rate and decontamination factor vs. time table data for user specified generic pathways. This method is only called to process table data for the iodine transport groups. This information is mapped to the Flowpath.java objects described in Section 3.2.12.
mapDecay	This method is responsible for processing the input for radioactive decay type. If radioactive decay or daughter production is turned on, this method sets a decay flag which is referenced in the calculation portion of RADTRAD to perform nuclide decay calculations. These flags are stored in Model.java class object, described in Section 3.2.1.

2.1.3 Map Additional XML Files

Additional RADTRAD input related to the source and nuclide information is processed next. This job is performed by the **readExternalFiles** method. This method calls the **read** method five times; once for each non-PSX input file type. The read method is called from the ResourceReader.java class described in Section 3.3.3. The read method is overridden by each ResourceReader.java daughter class (described in the subsections of 3.3.3) in order to process each different type of external input file. The file types have the following extensions: ICX, NIX, DFX, XML, and SRC. The logical structure of the read function and each reader class is explained in greater detail in Table 2-2.

Table 2-2 Methods for Reading Additional Input

Method Name	Description
readExternalFiles	This method calls the read method five times. The read method resides in the ResourceReader class, which is extended by the five reader classes: InventoryReader, NuclideReader, DoseFactorReader, ChemicalGroupReader, and SourceReleaseReader. Each of these classes contains a read method that overrides the ResourceReader read method to process a specific type of external input file. In other words, the read method performs a different

Method Name	Description
	function each time it is called in the readExternalFiles method. The purpose of each read method call is described below.
readICXNames	This method is small but has the important responsibility of adding nuclide components to the RADTRAD transport model. The nuclide objects can then be accessed in future methods.
read (InventoryReader)	This method call is responsible for processing the ICX input file which contains all inventory data for the RADTRAD case. The data specifies the amount of each individual nuclide (in Curies per megawatt) for the inventory.
read (NuclideReader)	This method call is responsible for processing the NIX input file which contains all generic nuclide data for the RADTRAD case. The data specifies the mass, the half-life, and the daughter production for each nuclide.
read (DoseFactorReader)	This method call is responsible for processing the DFX input file which contains all dose factor data for the RADTRAD case. The data specifies the nuclide dose conversion factor for each relevant human organ.
read (ChemicalGroupReader)	This method call is responsible for processing the chemical group data for the RADTRAD case. This data is stored in an XML file. The ten chemical groups are (1) noble gases, (2) halogens, (3) alkali metals, (4) tellurium, (5) alkali earth metals, (6) noble metals, (7) cerium, (8) lanthanides, (9) others, and (10) nonradioactive aerosols.
read (SourceReleaseReader)	This method call is responsible for processing the SRX input file which contains all source data for the RADTRAD case. The data specifies the gap, early, ex-vessel, and late release fractions for each chemical group.

2.2 Construct and Solve System Transport Matrix

This section covers steps 1-4 of radcalc, as illustrated in Figure 2-1. In these steps, the nuclide transport differential equations are set up, solved, and nuclide concentrations for each compartment are updated. In Section 2.2.1, the governing equations of this process are listed and described. Section 2.2.2 describes the RADTRAD methods that perform these steps, referring to the equations and terms where applicable. For a more detailed explanation of these equations, see Section 4.1 of Reference [2].

2.2.1 Theory

The governing equations in RADTRAD are first-order differential equations which describe the change of radioactive nuclide concentration in each compartment through nuclide transport into and out of the compartment by various methods. Equation 2-1 describes the change of the number of atoms in a hypothetical compartment k within a set of n interconnected compartments.

$$\frac{dN_k^i}{dt} = R_k^i - \sum_{j=1}^{n} \lambda_{j,k}^i \cdot N_k^i + \sum_{j=1, j\neq k}^{n} \lambda_{k,j}^i \cdot N_j^i \qquad \text{Equation 2-1}$$

where

N_k^i	=	inventory of species i in compartment k [atoms]
R_k^i	=	source release rate of species i in compartment k [atoms/s]
$\lambda_{j,k}^i$	=	removal coefficient of species i transporting from compartment k to compartment j [1/s]
$\lambda_{k,j}^i$	=	removal coefficient of species i transporting from compartment j to compartment k[1/s]
N_j^i	=	inventory of species i in compartment j [atoms]

This equation has three general terms. The first term, R_k^i, is the source release rate as described above. The second and third terms are the summations shown. The first summation represents the removal of atoms from compartment k (note that this term is subtracted) due to loss mechanisms within the compartment and transfer mechanisms to other compartments. The second summation represents the addition of atoms to compartment k (note that this term is added) due to transfer mechanisms from other compartments. The general matrix formulation of Equation 2-1 is shown in Equation 2-2.

$$\begin{pmatrix} \dot{N}_1^i \\ \dot{N}_2^i \\ \dot{N}_3^i \\ \vdots \\ \dot{N}_n^i \end{pmatrix} = \begin{pmatrix} R_1^i \\ R_2^i \\ R_3^i \\ \vdots \\ R_n^i \end{pmatrix} + \begin{pmatrix} -\Lambda_{1,1}^i & \Lambda_{1,2}^i & \Lambda_{1,3}^i & \cdots & \Lambda_{1,n}^i \\ \Lambda_{2,1}^i & -\Lambda_{2,2}^i & \Lambda_{2,3}^i & \cdots & \Lambda_{2,n}^i \\ \Lambda_{3,1}^i & \Lambda_{3,2}^i & \ddots & \cdots & \Lambda_{3,n}^i \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \Lambda_{n,1}^i & \Lambda_{n,2}^i & \Lambda_{n,3}^i & \cdots & -\Lambda_{n,n}^i \end{pmatrix} \begin{pmatrix} N_1^i \\ N_2^i \\ N_3^i \\ \vdots \\ N_n^i \end{pmatrix} \qquad \text{Equation 2-2}$$

where

$\Lambda_{x,y}^i$	=	transfer rate of species i from compartment y to compartment x [1/s]
$\dot{N}_z^i$	=	time derivative of the inventory of species i in some compartment z [atoms]

In the large Λ coefficient matrix shown in Equation 2-2, the removal of atoms from a compartment is represented by the diagonal terms, each of which are negative, while the addition of atoms to a compartment is represented by the off-diagonal terms. Each diagonal term consists of the following possible terms shown in Equation 2-3.

$$\Lambda_{j,j}^i = \lambda_{j,spray}^i + \lambda_{j,deposition}^i + \lambda_{j,recirculation}^i + \sum_{k=1,k\neq j}^{n} \lambda_{k,j}^i \qquad \text{Equation 2-3}$$

where

$\lambda_{j,spray}^i$	=	spray removal coefficient of species i for compartment j [1/s]
$\lambda_{j,deposition}^i$	=	natural deposition removal coefficient of species i for compartment j [1/s]
$\lambda_{j,recirculation}^i$	=	recirculation filter removal coefficient of species i for compartment j [1/s]
$\lambda_{k,j}^i$	=	removal coefficient of species i from compartment j to compartment k [1/s]

The first three terms in Equation 2-3 represent removal of atoms from the j compartment by mechanisms within a compartment. These mechanisms are removal by spray, natural deposition, and recirculation filters. The fourth term in Equation 2-3 represents the sum of removal of atoms through flow out of compartment j and into all other compartments k. This term is expanded in Equation 2-4 to show each possible outflow mechanism.

$$\lambda_{k,j}^i = \lambda_{k,j,filter}^i + \lambda_{k,j,leak}^i + \lambda_{k,j,pipe}^i + \lambda_{k,j,generic}^i \qquad \text{Equation 2-4}$$

where

$\lambda_{k,j,filter}^i$	=	filter removal coefficient of species i from compartment j to compartment k [1/s]
$\lambda_{k,j,leak}^i$	=	leakage removal coefficient of species i from compartment j to compartment k [1/s]
$\lambda_{k,j,pipe}^i$	=	pipe removal coefficient of species i from compartment j to compartment k [1/s]
$\lambda_{k,j,generic}^i$	=	generic pathway removal coefficient of species i from compartment j to compartment k [1/s]

The mechanisms illustrated by Equation 2-4 are the same mechanisms by which a compartment gains atoms from other compartments (represented by the off diagonal terms Equation 2-2). However, flow into a compartment is more complex because of the removal mechanisms of the pathway, such as deposition within a pipe or filter. Each of these coefficients is described in greater detail in Equation 2-5.

$$\Lambda_{j,k}^{i} = \left(1 - \frac{Eff_{j,k,filter}}{100}\right)\lambda_{j,k,filter}^{i} + \lambda_{j,k,leak}^{i} + \left(\frac{1}{DF_{j,k,pipe}}\right)\lambda_{j,k,pipe}^{i} + \left(\frac{1}{DF_{j,k,generic}}\right)\lambda_{j,k,generic}^{i} \qquad \text{Equation 2-5}$$

where

$Eff_{j,k,filter}$	=	efficiency of pathway filter from compartment k to compartment j [1/s]
$DF_{j,k,pipe}$	=	decontamination factor of pipe from compartment k to compartment j [1/s]
$DF_{j,k,generic}$	=	decontamination factor of generic pathway from compartment k to compartment j [1/s]

Equation 2-2 is solved in RADTRAD using a matrix exponential method which is used to solve systems of linear differential equations of the form shown in Equation 2-6.

$$\dot{N} = R + \Lambda N \qquad \text{Equation 2-6}$$

Equation 2-6 can be viewed as a simplified form of Equation 2-2 where R is the source release rate and Λ is the coefficient matrix. A solution to this differential equation is calculated with the RADTRAD solver in the form shown in Equation 2-7.

$$N_{k}^{i} = \sum_{j=1}^{n} E_{k,j}^{itg} \cdot N_{0\ k}^{i} + \sum_{s=1}^{m} F_{k,s}^{i} \cdot R_{s}^{i} \qquad \text{Equation 2-7}$$

where

$E_{k,j}^{itg}$	=	inventory contribution factor of transport group itg from compartment i to compartment k
$F_{k,s}^{i}$	=	source contribution factor of species i from source s to compartment k
$N_{0\ k}^{i}$	=	the initial concentration of atoms of species i in compartment k
R_{s}^{i}	=	total release rate of species i from source s

The RADTRAD method implementation of these equations is described in the following subsections.

2.2.2 Methods

The following subsections describe the methods that implement the theory discussed in Section 2.2.1.

2.2.2.1 Calculate Source Term Release Rates

This section describes step 1 of radcalc, as illustrated in Figure 2-1. The calculation of the inventory release rates into the source term compartment is controlled by two methods: **source** and **sterm**. The source method sets the initial concentration of nuclides in the source term compartment. This only needs to be performed for the first time step, as the xnupdt method described in Section 2.2.2.4 updates the nuclide concentrations for each compartment using the transport equations described above. For every other time step, the only calculational responsibility of the source method is to call the sterm method. The sterm method calculates the release rate of each source for each nuclide. This is the $\cdot\, R_{s}^{i}$ term described in Equation 2-7.

2.2.2.2 Build Transport Matrix

This section describes step 2 of radcalc, as illustrated in Figure 2-1. The Λ coefficient transport matrix shown in Equation 2-2 is constructed by the method **buildTransportMatrix**. In order to calculate each of the Λ coefficients, buildTransportMatrix method calls eleven other methods which are described below. The naming convention for these methods is the transport mechanism followed by the type of transport. Three general types of transport are specified, as follows:

loss: removal within a compartment, represented by the first three terms of Equation 2-3

out: removal by way of a pathway, represented by the terms in Equation 2-4.

in: gain by way of a pathway, represented by the off diagonal terms of the transport matrix in Equation 2-2.

The individual methods of these types called by buildTransportMatrix are described in Table 2-3, Table 2-4, and Table 2-5. These tables provide a description of each method along with the term it calculates from the equations described in 2.2.

Table 2-3 Description of buildTransportMatrix "_loss" Methods

Method Name	**Calculated Coefficient**		**Description**
	Value	**Equation**	
depositionLoss	$\lambda^{i}_{j,deposition}$	Equation 2-3	This method gathers the user-defined removal coefficient for aerosol deposition within a compartment. Alternatively, there is an option to calculate the removal coefficient using Henry's model or Powers' model of natural deposition. These models are discussed in Sections 4.5.2.1 and 4.5.2.2 of Reference [2].
sprayLoss	$\lambda^{i}_{j,spray}$		This method gathers the user-defined removal coefficient for spray within a compartment. Alternatively, there is an option to calculate the removal coefficient using Henry's model of spray deposition. This model is discussed in Sections 4.5.1 of Reference [2].
compFilterLoss	$\lambda^{i}_{j,recirculation}$		This method gathers the user-defined removal coefficient for recirculation through a filter within a compartment.

Table 2-4 Description of buildTransportMatrix "_out" Methods

Method Name	**Calculated Coefficient**		**Description**
	Value	**Equation**	
leakOut	$\lambda^{i}_{k,j,leak}$	Equation 2-4	This method gathers the user-defined removal coefficient for flow out of a compartment due to air leakage.
pipeOut	$\lambda^{i}_{k,j,pipe}$		These methods gather the user-defined flow rates out of a compartment via a pipe, filter, or generic pathway, respectively. The flow rates are divided by the compartment volume to give the coefficient.
filterOut	$\lambda^{i}_{k,j,filter}$		
genericOut	$\lambda^{i}_{k,j,generic}$		

Table 2-5 Description of buildTransportMatrix "_in" Methods

Method Name	Calculated Coefficient		Description
	Value	Equation	
leakIn	$\lambda^i_{k,j,leak}$	Equation 2-5	This method gathers the user-defined gain coefficient for flow into a compartment due to air leakage.
pipeIn	$\left(\frac{1}{DF_{j,k,pipe}}\right)\lambda^i_{j,k,pipe}$		This method gathers the user-defined flow rate into a compartment via a pipe along with the decontamination factor for the pipe. The flow rate is divided by the compartment volume to give the coefficient, which is then divided by the decontamination factor. The Brockmann and Bixler model can be used to calculate pipe deposition. This model is discussed in Sections 4.5.3.1, 4.5.3.2, and 4.5.3.3 of Reference [2].
pathFilterIn	$\left(1-\frac{Eff_{j,k,filter}}{100}\right)\lambda^i_{j,k,filter}$		This method gathers the user-defined flow rate into a compartment via a filtered pathway along with the efficiency of the filter. The flow rate is divided by the compartment volume to give the coefficient, which is then multiplied by (1 – filter$_{\text{efficiency}}$/100).
genericIn	$\left(\frac{1}{DF_{j,k,generic}}\right)\lambda^i_{j,k,generic}$		This method gathers the user-defined flow rate into a compartment via a generic pathway along with the decontamination factor for the pathway. The flow rate is divided by the compartment volume to give the coefficient, which is then divided by the decontamination factor.

Once each of these terms are calculated, they are stored in the coeff array listed in Table 3-1 which is returned to radcalc.

2.2.2.3 Solve Transport Matrix

This section describes step 3 of radcalc, as illustrated in Figure 2-1. As stated in Section 2.2, Equation 2-6 can be viewed as a simplified form of Equation 2-2 where R (calculated in Section 2.2.2.1) is the source release rate and Λ (calculated in Section 2.2.2.2) is the coefficient matrix. Once both terms are known, a solution is found with the **solver** method. This method calculates the inventory and source contribution factors defined in Equation 2-7: the $E^{itg}_{k,j}$ and $\mathrm{F}^i_{k,s}$ terms, respectively. These terms are the arrays inventory_cf and source_cf, listed in Table 3-1.

2.2.2.4 Calculate Nuclide Concentrations

This section describes step 4 of radcalc, as illustrated in Figure 2-1. Once the transport matrix solution is found, all terms from Equation 2-7 are known and the equation can be solved. The N^i_k term is calculated by the method **xnupdt**. This term is the array xn listed in Table 3-1. This method calculates the updated nuclide concentration of each nuclide in each compartment for a given timestep and thus concludes the nuclide transport algorithms of radcalc.

2.3 Calculate Doses

This section covers step 5 of radcalc, as illustrated in Figure 2-1. In this step, the doses are calculated from the updated compartment concentrations at the dose locations. In Section 2.3.1, the governing equations of this process are listed and described. Section 2.3.2 describes the RADTRAD methods that perform these steps, referring to the equations and terms where applicable. For a more detailed explanation of these equations, see Section 4.1 of Reference [2].

2.3.1 Theory

The dose to an individual in the environmental compartment is calculated at two dose locations: the EAB and LPZ. The dose due to cloudshine at each of these dose locations is calculated using Equation 2-8.

$$D_{c,n}^{env} = A_n^{env}\left(\frac{\chi}{Q}\right)DCF_{c,n} \qquad \text{Equation 2-8}$$

where

$\mathrm{D}_{c,n}^{env}$	=	cloudshine dose due to nuclide n in the environment compartment [Sv]
A_n^{env}	=	activity of nuclide n in the environmental compartment [Bq]
$\frac{\chi}{Q}$	=	atmospheric relative concentration (atmospheric dispersion factor) for the dose location [s/m^3]
$DCF_{c,n}$	=	cloudshine dose conversion factor for nuclide n [Sv-m^3/Bq-s]

The activity is calculated from two terms: the updated compartment nuclide concentrations in the environmental compartment and the nuclide decay constant. This is shown by Equation 2-9.

$$A_n^{env} = \lambda_n N_n^{env} \qquad \text{Equation 2-9}$$

where

λ_n	=	radiological decay constant for nuclide n [1/s]
N_n^{env}	=	number of atoms for nuclide n in environmental compartment [atoms]

The dose in the environmental compartment due to inhalation is calculated using Equation 2-10.

$$D_{i,n}^{env} = A_n^{env}\left(\frac{\chi}{Q}\right)BR * DCF_{i,n} \qquad \text{Equation 2-10}$$

where

$\mathrm{D}_{i,n}^{env}$	=	inhalation dose due to nuclide n in the environment compartment [Sv]
A_n^{env}	=	activity of nuclide n in the environmental compartment [Bq]
BR	=	breathing rate for the dose location [s/m^3]
$DCF_{i,n}$	=	inhalation dose conversion factor for nuclide n [Sv/Bq]

The dose to an individual in the control room compartment is calculated from the time-integrated concentration of nuclides in the compartment. For cloudshine, the dose is calculated using Equation 2-11.

$$D_{c,n}^{CR} = \left(\frac{DCF_{c,n}}{G_F}\right)OF\int C_n^{CR}(t)\,dt \qquad \text{Equation 2-11}$$

where

$\mathrm{D}_{c,n}^{CR}$	=	cloudshine dose due to nuclide n in the control room [Sv]
$DCF_{c,n}$	=	cloudshine dose conversion factor for nuclide n [Sv-m^3/Bq-s]
OF	=	occupancy factor for the dose location

$C_n^{CR}(t)$	=	instantaneous concentration of nuclide n in the control room [Bq-s]
G_F	=	Murphy-Campe geometric factor which relates the dose from an infinite cloud to the dose from a cloud of volume (V) [1/m^3]. This factor is defined in Equation 2-12.

$$G_F = \left(\frac{351.6}{V^{0.338}}\right) \qquad \text{Equation 2-12}$$

The dose in the control room compartment due to inhalation is calculated using Equation 2-13.

$$D_{i,n}^{CR} = BR * OF * DCF_{i,n} \int C_n^{CR}(t)\,dt \qquad \text{Equation 2-13}$$

where

$\mathrm{D}_{i,n}^{CR}$	=	inhalation dose due to nuclide n in the control room [Sv]
BR	=	breathing rate for the dose location [m^3/s]
$DCF_{i,n}$	=	inhalation dose conversion factor for nuclide n [Sv/Bq]

The RADTRAD method implementation of these equations is described in the following subsections.

2.3.2 Methods

The **calculateDose** method is responsible for calculating the environment and control room doses for receptors located at user-defined locations (e.g., exclusion area boundary, low population zone, control room). Called by the radcalc method, calculateDose calls the **dosesCR** and **dosesENV** methods to calculate the doses at the control room and environment, respectively. All dependent variables in the dose equations above are provided by the user other than the A_n^{env} and C_n^{CR} terms which are calculated from the updated nuclide concentrations determined from the previous step.

The dosesENV method carries out Equation 2-8, Equation 2-9, and Equation 2-10 and returns the calculated doses to the calculateDose method. The dosesCR method carries out Equation 2-11, Equation 2-12, and Equation 2-13 and also returns the calculated doses to the calculateDose method (C_n^{CR} from Equation 2-13 is approximated using the trapezoid rule). The calculateDose method then sorts these doses into their proper arrays depending on dose location, organ type, and dose route. Later, radcalc calls the commitAccumDose method which sums the doses for each timestep to obtain a running total for whole body dose, skin dose, thyroid dose, and the total effective dose equivalent.

2.4 Prepare for Next Time Step

This section covers steps 6 and 7 of radcalc, as illustrated in Figure 2-1. Each of these steps are necessary in order to prepare for the next time step. Section 2.4.1 describes an overview of this process along with the theory for step 6. Section 2.4.2 describes the RADTRAD methods that perform these steps.

2.4.1 Overview and Theory

For the first step of this process (step 6 of Figure 2-1), if one or both the radioactive decay and daughter production options are enabled by the user, RADTRAD updates the compartment concentrations to account for these behaviors. For simplicity, this process is modeled separately from the transport mechanisms described in Section 2.2. Decay and ingrowth are accounted for by updating the compartment nuclide concentrations once per time step, compartment nuclide concentrations via transport are updated and doses are calculated. Equation 2-14 shows the change in nuclide concentration in a compartment over time through decay and ingrowth. For a more detailed explanation of this process, see Section 4.1 of Reference [2].

$$\frac{dN_j^i}{dt} = \sum\nolimits_{v=1}^{n-1} \lambda^v \beta^{i,v} N_j^v + \lambda^i N_j^i \qquad \text{Equation 2-14}$$

where

N_j^i	=	number of atoms of nuclide i in compartment j [atoms]
λ_v	=	decay constant of nuclide v [1/s]
$\beta^{i,v}$	=	fraction of nuclide v that decays to nuclide i
N_j^v	=	number of atoms of nuclide v in compartment j [atoms]
λ^i	=	decay constant of nuclide i [1/s]

The second step in this process (step 6 Figure 2-1) is to determine the time step that will be used for the next radcalc iteration.

2.4.2 Methods

The first step of this process is controlled by the **dkngro** method. The dkngro method updates the compartment nuclide concentrations to account for radioactive decay and ingrowth. An exact solution to Equation 2-14 is not obtained by this method. Rather, an approximation is calculated by saving the compartment nuclide concentrations to a temporary array, and then performing the decay and ingrowth in two steps. In the first step, decay is calculated for each nuclide and the compartment nuclide concentration array is updated. This step is analogous to the second term in Equation 2-14, $\lambda^i N_j^i$. In the second step, daughter ingrowth is calculated for each nuclide using the temporary array (the compartment nuclide concentration prior radioactive decay) and the compartment nuclide concentration array is updated again. This step is analogous to the first term in Equation 2-14, $\sum_{v=1}^{n-1} \lambda^v \beta^{i,v} N_j^v$.

The second step of this process is controlled by the **autodt** method. This method returns a time step size that varies depending upon various conditions of the simulation. If the adaptive timestepping option is enabled by the user, this method will call other methods that carry out adaptive timestepping algorithms. For more information regarding this process, see Reference [3].

2.5 Print Output

This section describes step 8 of radcalc as well as the final step of a RADTRAD simulation, as illustrated in Figure 2-1. At the end of a radcalc iteration, the writePlotPoint method is called which writes data to the plot file and ultimately calls many other routines that print output incrementally in the simulation process. In addition, at the end of a simulation, a summary of doses is printed to the main output file by the printout_summary method. Currently, the overall RADTRAD output process is quite complex. Table 2-6 lists the main printing methods of RADTRAD and provides a brief description of each. As described in Section 3.1.3, the majority of the RADTRAD printing methods are contained in the Fileinfo.java class.

Table 2-6 Description of the Main Printing Methods in RADTRAD

Method Name	Class Location	Description
printNRCinput	NrcOutput.java	This method prints various NRC-formatted inputs to the NRC.OUT file.
writePlotPoint	Radtrad.java	The primary function of this method is to write data to the plot file and call the printout method which in turn calls various output routines. This method is called at the end of each timestep if the plot all points option is activated.
printout	Fileinfo.java	This method handles the majority of the RADTRAD output and calls the following methods: print_models, printMasses, new_print_masses,

Method Name	Class Location	Description
		printNRCoutput. A key function of this method is to print doses at each dose location at specific simulation times.
print_models	Fileinfo.java	If requested by the user, this method will print detailed information about the transport model at specific simulation times.
printMasses	Fileinfo.java	These methods print out nuclide concentration information at specific simulation times.
new_print_masses	Fileinfo.java	
printNRCoutput	NrcOutput.java	The printNRCoutput method prints various output parameters such as dose and activity distributions in NRC-formatted output style.
print_psf	Fileinfo.java	If requested by the user, this method will be called once at the beginning of a simulation This method prints a general plant description near the top of the OUT file. It loops through each compartment object and prints an overview of the user provided transport model.
print_sdf	Fileinfo.java	If requested by the user, this method will be called once at the beginning of the simulation. This method prints a general scenario description near the top of the OUT file. The type of information printed includes plant power, radioactive decay model, and nuclide source data.
print_rft	Fileinfo.java	This method is called by the print_sdf method. It prints the release fraction and timings information for each chemical group near the top of the OUT file.
printout_init	Fileinfo.java	This method is called once at the beginning of the simulation. This method prints the massive “OUTPUT” heading in the OUT file.
printStats	Fileinfo.java	This method is called at the end of the radcalc method. It prints runtime statistics to the STATS file such as CPU time information.
print_header	Fileinfo.java	This method is called a few times throughout the RADTRAD simulation process. In one call, during input processing, the method is passed an argument that causes it to print PSF input header information to the OUT file. In the other calls, the method is passed an argument that causes it to print a generic header to the OUT file.
printout_summary	Fileinfo.java	This method is called once at the end of the simulation. It prints the final summary of dose information to the OUT file.

2.5.1 Chain of RADTRAD Printing Methods

As briefly described in Section 3.3.4, the log method – called frequently in RADTRAD by printing methods – calls the lowest level printing method in RADTRAD: the addMessage method. The addMessage method prints each message argument it receives to the proper output file. There are various small methods in the Fileinfo.java class that simply pass their arguments to the log method, which in turn passes its arguments to the addMessage method. These small methods are listed and described in Table 2-7. Each of the higher-level printing methods described in the subsections below call these lower-level methods to transmit the desired output down to the addMessage method.

Table 2-7 Description of the small printing methods that ultimately call the log method

Method Name	Class Location	Description
log	Fileinfo.java	Passes message to Logger.java log method
out	Fileinfo.java	
screen	Fileinfo.java	
warning	Fileinfo.java	
error	Fileinfo.java	
logf	Fileinfo.java	Passes formatted message to Fileinfo.java log method

Method Name	Class Location	Description
outf	Fileinfo.java	Passes formatted message to Fileinfo.java out method
screenf	Fileinfo.java	Passes formatted message to Fileinfo.java screen method
warningf	Fileinfo.java	Passes formatted message to Fileinfo.java warning method
errorf	Fileinfo.java	Passes formatted message to Fileinfo.java error method
nrcf	NrcOutput.java	Passes formatted message to Logger.java log method
nrc	NrcOutput.java	Passes message to Logger.java log method

3 Development View

This section describes the RADTRAD code architecture from the development view perspective. The RADTRAD source code is separated into various components called packages. Each package is organized to contain pieces of code that are related in some way. This grouping structure allows for improved code readability and maintainability. The development view diagrams included in this section illustrate how the RADTRAD code is organized by listing the most important java files that are present in each RADTRAD package. Each package and java file is then described in this section with emphasis placed upon the major elements and features of the code. Figure 3-1 illustrates each RADTRAD package. The following sections describe the most significant packages in greater detail.

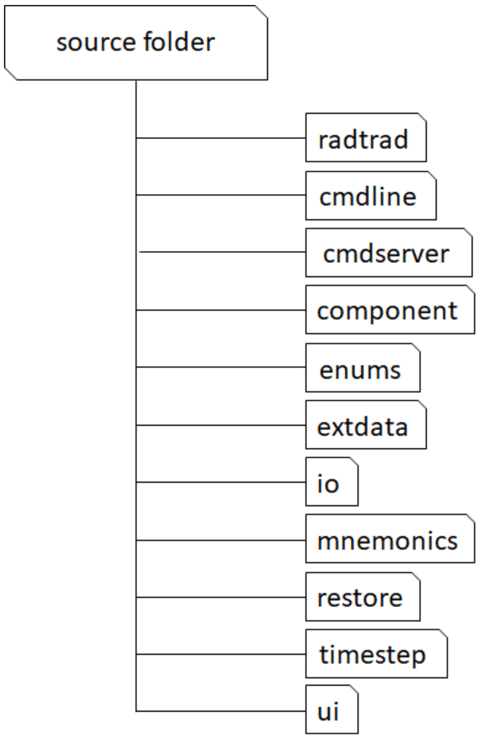


Figure 3-1 Illustration of the 11 packages containing Java files from in the RADTRAD source folder

3.1 Package radtrad

The radtrad package is the central calculational package of the RADTRAD code. All major calculations – building the system transport matrix, calculating nuclide concentrations, and ultimately calculating doses – are performed using methods found within this package. Additionally, most of the output printing routines are found within this package. Figure 3-2 shows a list of the most significant java files in the radtrad package.

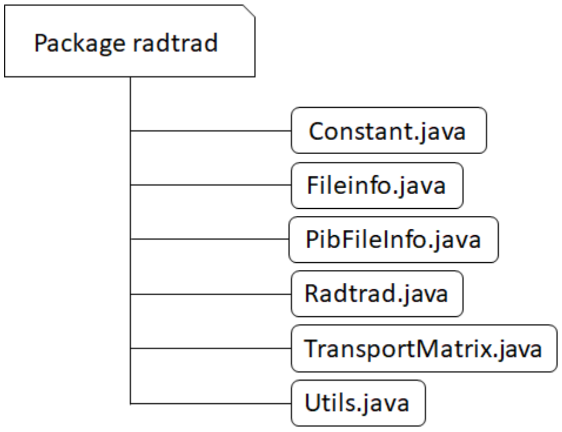


Figure 3-2 Illustration of the most significant Java files contained in the radtrad package

3.1.1 Radtrad.java

The Radtrad.java class is the primary RADTRAD calculational class. Radtrad.java contains the radcalc method, which is the driving function for all calculational aspects of RADTRAD. Additionally, most of the central methods that the radcalc method calls to perform nuclide and dose calculations are stored in Radtrad.java. These methods are described in Section 2. At the beginning of a RADTRAD execution, the Radtrad.java class is constructed once, and there is one instance of the Radtrad.java class per execution.

The Radtrad.java class is also primarily responsible for command line option configuration. When RADTRAD is executed from the command line, the Radtrad.java main method checks for command line options before beginning with execution of the radcalc method.

3.1.2 TransportMatrix.java

The TransportMatrix.java class contains the buildTransportMatrix method along with all sub-methods that the buildTransportMatrix method makes calls to. These methods are described in Section 2.2. At the beginning of a RADTRAD execution, the TransportMatrix.java class is constructed once, and there is one instance of the TransportMatrix.java class per execution. The sole purpose of the TransportMatrix.java object is to construct the system transport matrix for the nuclide transport matrix differential equation. The system matrix is used in the radcalc method to calculate nuclide concentrations for each compartment and doses for each specified dose location.

3.1.3 Fileinfo.java

The Fileinfo.java class is a very large class that is the primary repository for the printing methods called throughout the radcalc method. These methods are described in 2.4. Additionally, the Fileinfo.java class contains methods that facilitate input processing between the Radtrad.java object and the io package which is directly responsible for the processing of input files (see Section 3.3). At the beginning of a RADTRAD execution, the Fileinfo.java class is constructed once, and there is one instance of the Fileinfo.java class per execution.

3.1.4 PibFileInfo.java

The Fileinfo.java class is the primary repository for the plot file generation methods in RADTRAD. Near the beginning of a RADTRAD execution, the PibFileinfo.java class is constructed once, and there is one instance of the PibFileinfo.java class per execution.

3.1.5 Constant.java

The Constants.java class contains various static final variables and static conversion methods (such as Avogadro's number and temperature unit conversion methods) that are used throughout the RADTRAD code. All variables and methods within this class are static. Thus, the Constants.java class is used many times but is never instantiated as an object.

3.1.6 Utils.java

The Utils.java class contains various static methods that are called throughout the RADTRAD code for utility purposes. One of these is a lookup table method which is used to return table values for specific components corresponding to a specific simulation time. Additionally, the system transport matrix solver equations are stored in the Utils.java class. Because all of its methods are static, the Utils.java class is called many times but never instantiated as an object.

3.2 Package component

The component package is the central repository for all classes associated with the user provided transport model. Figure 3-3 shows a list of every java class within the component package. Each of these classes within the component package represents a type of model component that the user can specify. During the input processing phase, as described in Section 2.1, the user provided transport model is mapped to RADTRAD components. With every model component that is read-in, a new component object is constructed from the component package.

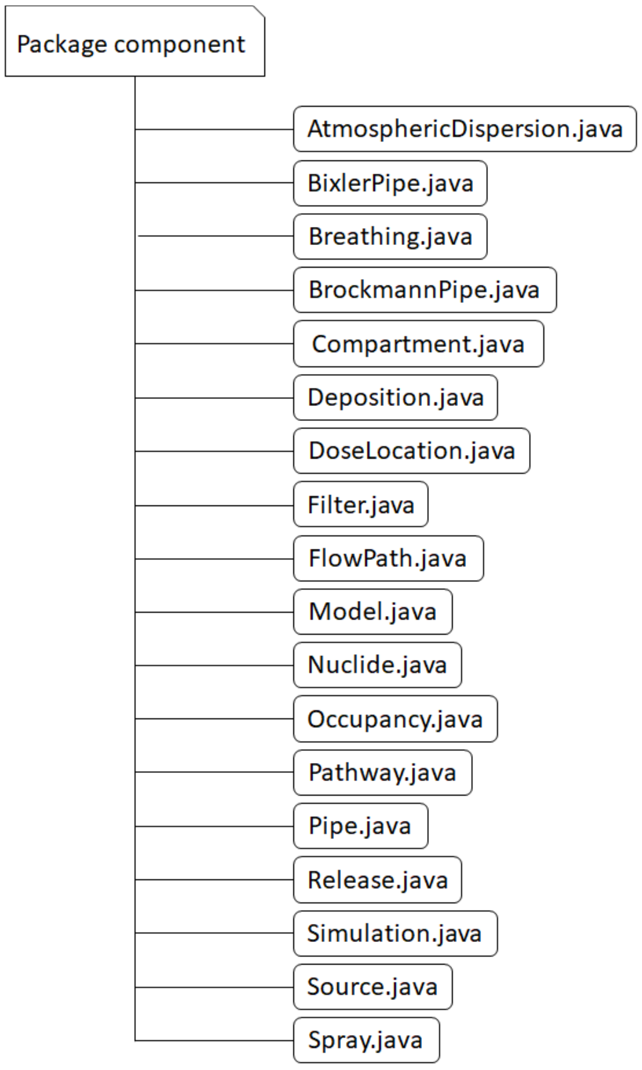


Figure 3-3 Illustration of the java files contained in the component package

3.2.1 Model.java

The Model.java class is the primary RADTRAD class for the user provided transport model. At the beginning of a RADTRAD execution, the Model.java class is constructed once, and there is one instance of the Model.java class per execution. The Model.java class object is the data structure that holds all model components from the user provided transport model. The object stores all compartments, pathways, dose locations, atmospheric dispersion tables, sources, releases, and nuclides in seven separate ArrayList data structures. It also holds one Simulation.java class object, various specific variables and arrays pertaining to the user provided model options, and methods that allow for additional components to be added when read-in or retrieved when called.

Each component that is stored in the Model.java class object is an instance of that component class from the component package. For example, all compartments are stored in the Model.java object, and each compartment is constructed from the Compartment.java class within the component package. Each compartment is its own unique object, and stores other objects, such as Filter.java class objects. Thus, the Model.java object is the highest-level data structure for the user constructed input model. Once the entire transport model is processed, every aspect of the model can be accessed from the Model.java class object.

3.2.2 Compartment.java

The Compartment.java class is a blueprint for all possible compartment objects. The Compartment.java class is constructed every time a new compartment is read-in from the user provided transport model. There is no limit to the number of compartments that can be constructed in one RADTRAD simulation. All compartment objects are stored in the “compartments” ArrayList in the Model.java class object.

The Compartment.java class contains various specific variables and arrays pertaining to the user provided compartment options, and methods that allow for additional components to be added when read-in or retrieved when called. Each compartment may contain a filter, spray, and deposition object.

3.2.3 Pathway.java

The Pathway.java class is a blueprint for all possible pathway objects. The Pathway.java class is constructed every time a new pathway is read-in from the user provided transport model. There is no limit to the number of pathways that can be constructed in one RADTRAD simulation. All pathway objects are stored in the “pathways” ArrayList in the Model.java class object.

The Pathway.java class contains various specific variables and arrays pertaining to the user provided pathway options, and methods that allow for additional components to be added when read-in or retrieved when called. Each pathway may contain a filter, flowpath, or pipe object. A single pathway can only contain one of these types of objects, because these objects determine the pathway typology, and one pathway can only have one type. These pathway types (filter, flowpath, and pipe) are constructed from other component classes that are described below in Sections 3.2.9, 3.2.12, and 3.2.13.

3.2.4 DoseLocation.java

The DoseLocation.java class is a blueprint for all possible dose location objects. The DoseLocation.java class is constructed every time a new dose location is read-in from the user provided transport model. There is no limit to the number of dose locations that can be constructed in one RADTRAD simulation, but each model must contain an environmental dose location. All dose location objects are stored in the “doseLocations” ArrayList in the Model.java class object.

The DoseLocation.java class contains a few specific variables pertaining to the user provided compartment options such as the compartment number, and methods that allow for additional components to be added when read-in or retrieved when called. Each compartment may contain a breathing or occupancy object, described in Sections 3.2.16 and 3.2.17, respectively. Occupancy objects are only added for the control room dose and normal dose location type.

3.2.5 AtmosphericDispersion.java

The AtmosphericDispersion.java class is a blueprint for all possible atmospheric dispersion objects. The AtmosphericDispersion.java class is constructed every time a new atmospheric dispersion table is read-in from the user provided transport model. There is no limit to the number of atmospheric dispersion tables that can be added in one RADTRAD simulation. All atmospheric dispersion objects are stored in the “atmosphericDispersions” ArrayList in the Model.java class object.

The AtmosphericDispersion.java class contains two arrays to form an atmospheric dispersion table (atmospheric dispersion factors vs. time), and a lookup method that allows for atmospheric dispersion factors to be retrieved at a specific simulation time. Atmospheric dispersion tables are added for compartments in the model that are connected through an environmental pathway.

3.2.6 Source.java and Release.java

The Source.java class is a blueprint for all possible source objects. The Source.java class is constructed every time a new source is read-in from the user input (SRX file). There can be many sources in a RADTRAD execution, but there is typically only one. All source objects are stored in the "sources" ArrayList in the Model.java class object. The Source.java class contains variables for the iodine form fractions and source term fractions. The actual release fractions are handled by the Release.java class objects.

The Release.java class is a blueprint for all possible release objects. The Release.java class is constructed every time a new release is read-in from the user input. Release information is contained within the SRX file. There are typically ten release objects for each source object; one for each chemical group. All release objects are stored in the "releases" ArrayList in the Model.java class object. The Release.java class only contains four variables: the gap, early, ex-vessel, and late release times.

3.2.7 Nuclide.java

The Nucide.java class is a blueprint for all possible nuclide objects. The Nuclide.java class is constructed every time a new nuclide is read-in from the user input (ICX file). There is no limit to the number of nuclides that can be constructed in one RADTRAD simulation. All nuclide objects are stored in the "nuclides" ArrayList in the Model.java class object.

The Nuclide.java class contains various specific variables and arrays pertaining to the user provided nuclide data in the external data files. Primarily, the ICX file determines how many nuclides will be constructed for the RADTRAD simulation and the initial concentrations. Other nuclide variables such as chemical group and decay constant are determined by the XML and NIX files, respectively.

3.2.8 Simulation.java

The Simulation.java class is a blueprint for all possible simulation objects. The Simulation.java class is constructed at the beginning of the input processing of the user provided transport model. There can only be one simulation object per RADTRAD execution. This simulation object is stored in the Model.java class object.

The Simulation.java class contains various specific variables and arrays pertaining to the user provided simulation options such as the show plant and show scenario options which generate flags to print more detailed plant and scenario data at the time of output. Additionally, the simulation object controls timestep behavior type through the time step algorithm enumeration function that specifies default time stepping or adaptive time stepping. The simulation object also contains many functions to help manage calculation progression at key simulation times.

3.2.9 Filter.java

The Filter.java class is a blueprint for all possible filter objects. The Filter.java class is constructed every time a new filter table is read-in from the user provided transport model. As stated in Sections 3.2.2 and 3.2.3, filters can be added to compartments or pathways. This is where the objects are stored. In a compartment, the filter takes on the role of a recirculation filter. In a pathway, the filter defines the pathway as a filtered pathway. Only one filter can be added to any single compartment or pathway, but there is no limit to the size of a filter table.

The Filter.java class contains five arrays to form a filter table: flow rate, aerosol iodine efficiency, organic iodine efficiency, and elemental iodine efficiency, all vs. time. The Filter.java class also contains lookup methods that allow for filter efficiencies and flow rates to be retrieved at specific simulation times.

3.2.10 Spray.java

The Spray.java class is a blueprint for all possible spray objects. The Spray.java class is constructed every time a new spray table is read-in from the user provided transport model. Spray objects are stored in compartments. Only one spray object can be added to any single compartment for any single transport group, but there is no limit to the size of a spray table.

If the Powers model is selected for aerosol removal due to spray, the following inputs are required: a Powers percentile value, a Powers spray ratio table, and a Powers spray table (height and flux vs. time). Alternatively, user defined coefficients can be selected, in which case a removal coefficient vs. time table is required. The Spray.java class also contains lookup methods that allow table values to be retrieved at specific simulation times.

3.2.11 Deposition.java

The Deposition.java class is a blueprint for all possible deposition objects. The Deposition.java class is constructed every time a new deposition table is read-in from the user provided transport model. Deposition objects are stored in compartments. Only one deposition object can be added to any single compartment for any single transport group, but there is no limit to the size of a deposition table.

If the Powers model is selected for aerosol deposition, a Powers percentile value and a Powers accident type are required input. Additionally, if the Henry model is selected for aerosol deposition, a height and particle density vs. time table is required input. Alternatively, user defined coefficients can be selected, in which case a removal coefficient vs. time table is required. The Deposition.java class also contains lookup methods that allow table values to be retrieved at specific simulation times.

3.2.12 FlowPath.java

The FlowPath.java class is a blueprint for all possible flowpath objects. The Flowpath.java class is constructed every time a new flowpath table is read-in from the user provided transport model. Flowpath objects are stored in pathways. Only one flowpath object can be added to any single pathway for any single transport group, but there is no limit to the size of a flowpath table. A flowpath object defines the pathway it is stored in. A flowpath can either be a generic pathway or a leakage pathway.

If a leakage flowpath is selected for a pathway, only one leak rate vs. time table is required. However, if a generic flowpath is selected for a pathway, three tables of flow rate and decontamination factor vs. time must be provided – one for each iodine transport group – and one noble gas flow rate vs. time table is required. The FlowPath.java class also contains lookup methods that allow table values to be retrieved at specific simulation times.

3.2.13 Pipe.java

The Pipe.java class is a blueprint for all possible pipe objects. The Pipe.java class is constructed every time a new pipe table is read-in from the user provided transport model. Pipe objects are stored in pathways. Only one pipe object can be added to any single pathway for any single transport group, but there is no limit to the size of a pipe table. A pipe object defines the pathway it is stored in. The pipe described in this section is a generic pipe. Brockmann and Bixler pipes are described in Sections 3.2.14 and 3.2.15.

If a generic pipe is selected for a pathway, only one decontamination factor and flow rate vs. time table is required per transport group. The Pipe.java class also contains lookup methods that allow table values to be retrieved at specific simulation times.

3.2.14 BrockmannPipe.java

The BrockmannPipe.java class is a blueprint for all possible BrockmannPipe objects. The Brockmann pipe object is bound by the same rules as the pipe object described in Section 3.2.13. However, only the aerosol transport group can be modeled to transport through a Brockmann pipe. The following inputs are required for this type of pipe: pipe volume, pipe surface area, pipe angle, pipe temperature, pipe pressure, pipe aerosol settling velocity, and a pipe aerosol flow rate vs. time table.

3.2.15 BixlerPipe.java

The BixlerPipe.java class is a blueprint for all possible BixlerPipe objects. The Bixler pipe object is bound by the same rules as the pipe object described in Section 3.2.13. However, only the elemental and organic transport groups can be modeled to transport through a Bixler pipe. The only input required for this type of pipe is the deposition settling velocity.

3.2.16 Breathing.java

The Breathing.java class is a blueprint for all possible breathing objects. The Breathing.java class is constructed every time a new breathing rate table is read-in from the user provided transport model. Breathing objects are stored in dose locations. Only one breathing object can be added to any single dose location, but there is no limit to the size of a breathing rate table.

The Breathing.java class contains two arrays that form a breathing rate vs. time table. The Breathing.java class also contains lookup methods that allow table values to be retrieved at specific simulation times.

3.2.17 Occupancy.java

The Occupancy.java class is a blueprint for all possible occupancy objects. The Occupancy.java class is constructed every time a new occupancy factor table is read-in from the user provided transport model. Occupancy objects are stored in dose locations. Only one occupancy object can be added to any single dose location, but there is no limit to the size of an occupancy factor table. Occupancy objects are only added to dose locations with the following compartment types: control room dose and normal dose.

The Occupancy.java class contains two arrays that form an occupancy factor rate vs. time table. The Occupancy.java class also contains lookup methods that allow table values to be retrieved at specific simulation times.

3.2.18 Description of Indexing for Arrays with Component Dimensions

This section briefly describes how RADTRAD components interact with the central computational arrays used in the calculation process.

3.2.18.1 How Array Dimensions are Set

In Java, the ArrayList data structure is dynamically allocated. This means, for example, that the “compartments” ArrayList – stored in the Model.java class object, as described in Section 3.2.1 – grows as new compartments are read in. Thus, because ArrayLists do not have fixed sizes, the size of the “compartments” ArrayList will vary with each new user provided transport model.

Unlike ArrayLists, RADTRAD computational arrays are not dynamically allocated. Thus, computational arrays with component dimensions are constructed only after all relevant components are read in from the user provided transport model. For example, if a computational array has a compartment dimension, the array is not constructed until all compartments have been processed, whereupon the array is sized by calling the size method on the "compartments" ArrayList. The size method returns the number of items in the ArrayList. This process results in the compartment dimension of the computational array being set to the exact number of compartments defined in the transport model. This process applies to all computational arrays with component dimensions.

3.2.18.2 How Arrays are Accessed

RADTRAD component objects typically possess an index variable. A component's index variable is an integer that is used to distinguish the component from others of the same type. Index variables also allow computational arrays to communicate with specific components.

Consider a one-dimensional array that stores nuclide activity for each compartment. Assuming there exists ten compartments in the transport model, each is sequentially read-in as discussed in Section 3.2.2 and assigned an index from zero to nine. The nuclide activity array elements correspond to the compartment index. For example, assuming the array name is "nuclideActivity," the first element of the array (nuclideActivity[0]) would store the nuclide activity for the compartment with an index of zero, the second element of the array (nuclideActivity[1]) would store the nuclide activity for the compartment with an index of one, and so on.

As discussed in Section 3.2.1, the ten compartment objects are stored in the "compartments" ArrayList in the model object. They are stored in sequential order by index number and can thus be retrieved by index number through the ArrayList get method. This allows for components to be easily accessed throughout RADTRAD. For example, assume it is necessary to retrieve the volume (a field within the compartment object) when calculating the nuclideActivity array discussed above. If attempting to calculate the value for nuclideActivity[4], the correct compartment is retrieved by calling the get(4) command on the compartments ArrayList. This will return the fifth element of the ArrayList which corresponds with the compartment with an index of four. Upon retrieving the compartment object, the volume field can be accessed.

3.2.18.3 Key Arrays

Table 3-1 describes some of the major computational arrays discussed in this section. Each of these arrays has at least one component dimension.

Table 3-1 Description of key computational arrays with component dimensions

Array name	Description	Number of component dimensions	Component dimension 1	Component dimension 2
source_cf	Source term contribution factors between each compartment by transport group	2	compartments	compartments
inventory_cf	Inventory contribution factors between each compartment by transport group	2	compartments	compartments
coeff	Coefficients for the transport and removal of the transport groups between each compartment	2	compartments	compartments
xn	Quantity of each nuclide at each compartment	2	nuclides	compartments
path	Quantity of each nuclide at each pathway	2	nuclides	pathways

xngrp	Quantity of each transport group at each compartment	1	compartments	-
pthgrp	Quantity of each transport group at each pathway	1	pathways	-
rrate	Each nuclide release rate for each source	2	nuclides	sources
grrate	Release rate into each compartment by transport group	1	compartments	-
dose_wbod	Whole body doses acquired for the time interval at each dose location	1	dose locations	-
dose_skin	Skin doses acquired for the time interval at each dose location	1	dose locations	-
dose_thyr	Thyroid doses acquired for the time interval at each dose location	1	dose locations	-
doses_tede	Total effective dose equivalent acquired for the time interval at each dose location	1	dose locations	-
accd_wbod	Accumulated whole body dose at each dose location	1	dose locations	-
accd_skin	Accumulated skin dose at each dose location	1	dose locations	-
accd_thyr	Accumulated thyroid dose at each dose location	1	dose locations	-
accd_tede	Accumulated total effective dose equivalent at each dose location	1	dose locations	-
transport_eff	Transport efficiency for each pathway by transport group	1	pathways	-

3.3 Package io

The io package is the central repository for classes associated with RADTRAD input/output processes. Figure 3-4 shows a list of every java class within the io package. Each of these classes within the io package is responsible for reading and processing a certain type of input data. During the input processing phase, the user provided inputs are mapped to RADTRAD components. The following sections describe the structure and role of each class in this package.

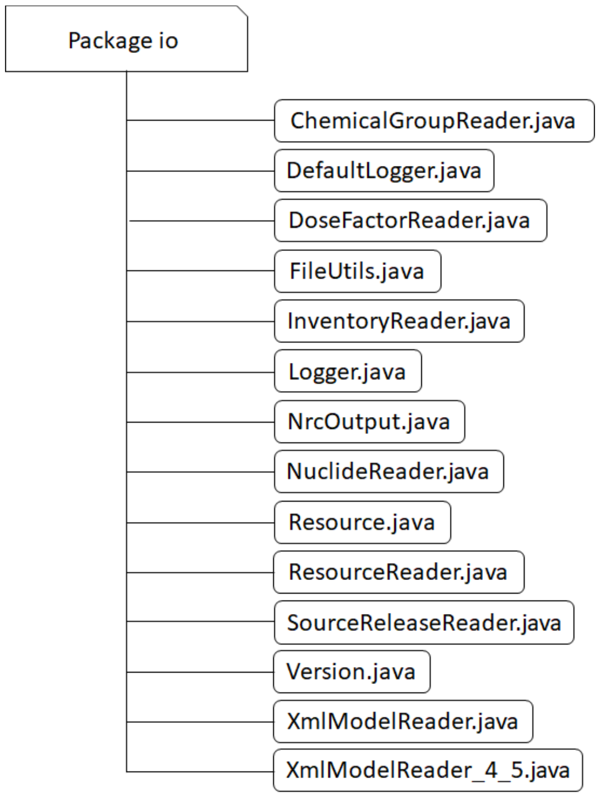


Figure 3-4 Illustration of the java files contained in the io package

3.3.1 XmlModelReader.java and XmlModelReader_4_5.java

The XmlModelReader.java class is the parent to the XmlModelReader_4_5.java class. The latter class holds the bulk of the methods that are called to process the main RADTRAD file (PSF file) which contains all non-inventory data in the user provided transport model. Each major method in this class is described in Section 2.1.2. At the beginning of a RADTRAD execution, the XmlModelReader.java and XmlModelReader_4_5.java classes are constructed once, and there is one instance of each class per execution.

3.3.2 Resource.java

The Resource.java class is instantiated every time a new external input file is processed. The file types that are translated as resource objects are the external data files described in Section 2.1.3: the ICX, NIX, DFX, XML, and SRC files. Each resource object has a name and pathway field.

3.3.3 ResourceReader.java

The ResourceReader.java class is the parent class to the four classes described in the subsections below. These classes contain the methods responsible for reading each resource object described in Section 3.3.2. Common methods for reading each resource are stored in this parent class. Among these is the read method which is the primary method for reading each resource object. As described in Section 2.1.3, the read method is overridden in each of the daughter classes in order to process the specific type of resource object.

3.3.3.1 ChemicalGroupReader.java

The ChemicalGroupReader.java class contains the methods required to read the XML file resource object and map the data to each nuclide object's group field. The primary method that performs this function is described in Section 2.1.3.

3.3.3.2 DoseFactorReader.java

The DoseFactorReader.java class contains the methods required to read the DFX file resource object and map the data to each nuclide object's dose factor field. The primary method that performs this function is described in Section 2.1.3.

3.3.3.3 InventoryReader.java

The InventoryReader.java class contains the methods required to read the ICX file resource object and map the data to each nuclide object's inventory fields, such as the insert rate and concentration. This also contains the method that is responsible for adding nuclide objects to the transport model. The primary method that performs these functions are described in Section 2.1.3.

3.3.3.4 NuclideReader.java

The NuclideReader.java class contains the methods required to read the NIX file resource object and map the data to each nuclide object's nuclide property fields, such as decay constant and atomic mass. The primary method that performs this function is described in Section 2.1.3.

3.3.3.5 SourceReleaseReader.java

The SourceReleaseReader.java class contains the methods required to read the SRX file resource object. Release components are added to the RADTRAD transport model and all release data is mapped to these components. The primary method that performs this function is described in Section 2.1.3.

3.3.4 Logger.java

The Logger.java class is an abstract class that defines the lowest level printing methods that handle all RADTRAD output generation. The primary method in this class is the addMessage method which is called by the log method. The log method is called frequently in RADTRAD by external printing methods. The addMessage method is not implemented in this class. Instead, it is implemented by its daughter class, DefaultLogger.java, described in the following subsection.

3.3.4.1 DefaultLogger.java

The DefaultLogger.java class contains the implementation for the addMessage method, which is the lowest level printing method in RADTRAD. This class is constructed once at the beginning of a RADTRAD simulation. In the DefaultLogger.java constructor, each output file is defined. This allows the addMessage method implementation to print messages to different output files based upon the call arguments. The RADTRAD print output process is described in greater detail in Section 2.4.

3.3.5 Version.java

The Version.java class contains methods that read in the RADTRAD version number and allow the number to be compared to other versions. This class is instantiated once at the beginning of the main input file processing.

3.3.6 NrcOutput.java

The NrcOutput.java class is a large class that is the primary repository for the printing methods related to output information commonly desired by the NRC. These methods are described in Section 2.4. At the beginning of a RADTRAD execution, the NrcOutput.java class is constructed once, and there is one instance of the NrcOutput.java class per execution.

3.3.7 FileUtils.java

The FileUtils.java class contains a collection of file utility methods that allow file properties to be retrieved. All methods in this class are static and thus, this class is never instantiated.

3.4 Package mnemonics

The mnemonics package is the central repository for all mnemonic classes in RADTRAD. All variables in each class within the mnemonics package are static and final. Therefore, though they are referenced frequently, no classes in this package are instantiated in the RADTRAD code. When another class needs access to the variables within a mnemonics class, the mnemonic class is imported and its variables are referenced statically. Figure 3-5 shows a list of every java class within the mnemonics package. The classes are clearly titled to describe the type of indexes contained within them.

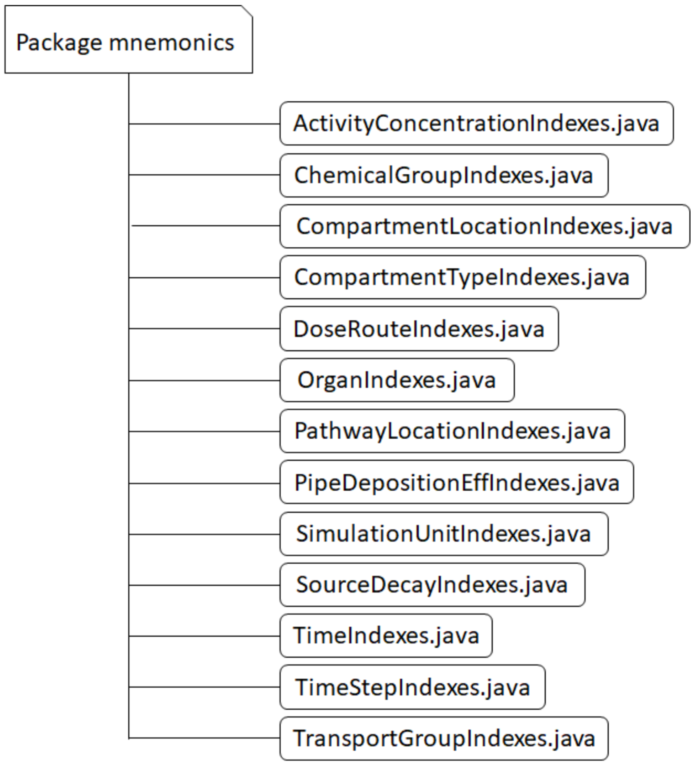

Package mnemonics
ActivityConcentrationIndexes.java
ChemicalGroupIndexes.java
CompartmentLocationIndexes.java
CompartmentTypeIndexes.java
DoseRouteIndexes.java
OrganIndexes.java
PathwayLocationIndexes.java
PipeDepositionEffIndexes.java
SimulationUnitIndexes.java
SourceDecayIndexes.java
TimeIndexes.java
TimeStepIndexes.java
TransportGroupIndexes.java

Figure 3-5 Illustration of the java files contained in the mnemonics package

3.4.1 ActivityConcentrationIndexes.java

The ActivityConcentrationIndexes.java class contains three static final variables that are used throughout the RADTRAD code to distinguish between three activity concentration types: Curies per megawatt, Curies per cubic centimeter, and Curies per hour. These variables are set to integer values of one through three, respectively.

3.4.2 ChemicalGroupIndexes.java

The ChemicalGroupIndexes.java class contains eleven static final variables that are used throughout the RADTRAD code to distinguish between the ten different chemical groups that nuclides can be classified under: noble gases, halogens, alkalis, telluriums, alkaline earths, noble metals, ceriums, lanthanides, others, and nonradioactive aerosols. The final mnemonic value is the chemical group count. These variables are set to integer values of zero through ten, respectively.

3.4.3 CompartmentLocationIndexes.java

The CompartmentLocationIndexes.java class contains six static final variables that are used throughout the RADTRAD code to distinguish between the five different compartment locations: atmosphere, containment sump, not defined, deposition surfaces, and filter. The final mnemonic value is the compartment location count. These variables are set to integer values of zero through five, respectively.

3.4.4 CompartmentTypeIndexes.java

The CompartmentTypeIndexes.java class contains six static final variables that are used throughout the RADTRAD code to distinguish between the six different compartment types: unused, control room dose, environment, normal, normal dose, and control room. These variables are set to integer values of zero through five, respectively.

3.4.5 DoseRouteIndexes.java

The DoseRouteIndexes.java class contains four static final variables that are used throughout the RADTRAD code to distinguish between the three different dose routes: inhalation, cloud shine, and skin. The final mnemonic value is the dose route count. These variables are set to integer values of zero through four, respectively.

3.4.6 OrganIndexes.java

The OrganIndexes.java class contains four static final variables that are used throughout the RADTRAD code to distinguish between the three different organs relevant to RADTRAD's dose calculations: thyroid, whole body, and skin. The final mnemonic value is the organ count. These variables are set to integer values of five, seven, eight, and nine, respectively. The numbering is as such because there were nine specified organs in previous versions of the RADTRAD code.

3.4.7 PathwayLocationIndexes.java

The PathwayLocationIndexes.java class contains six static final variables that are used throughout the RADTRAD code to distinguish between the five different pathway locations: pipe wall, filter, generic, no accumulation location, and not defined. The final mnemonic value is the pathway location count. These variables are set to integer values of zero through five, respectively.

3.4.8 PipeDepositionEffIndexes.java

The PipeDepositionEffIndex.java class contains five static final variables that are used throughout the RADTRAD code to distinguish between the five different iodine pipe deposition efficiency types: aerosol decontamination factor, elemental efficiency, elemental decontamination factor, organic efficiency, and organic decontamination factor. These variables are set to integer values of four through eight, respectively.

3.4.9 SimulationUnitIndexes.java

The SimulationUnitIndexes.java class contains three static final variables that are used throughout the RADTRAD code to distinguish between three simulation units: grams, Curies, and megabecquerels. These variables are set to integer values of zero through two, respectively.

3.4.10 SourceDecayIndexes.java

The SourceDecayIndexes.java class contains five static final variables that are used throughout the RADTRAD code to distinguish between five source decay types: no decay or daughtering, no decay, active decay, no daughtering, and daughtering. These variables are set to integer values zero, one, two, zero, one, respectively.

3.4.11 TimeIndexes.java

The TimeIndexes.java class contains four static final variables that are used throughout the RADTRAD code to distinguish between four time index types: time index, minimum delta-t index, maximum delta-t index, and maximum error index. These variables are set to integer values of zero through three, respectively.

3.4.12 TimeStepIndexes.java

The TimeStepIndexes.java class contains four static final variables that are used throughout the RADTRAD code to distinguish between three time step identifiers: reference step, calculation step, and second calculation step. The final mnemonic value is the organ count. These variables are set to integer values of zero, one, two, and two respectively.

3.4.13 TransportGroupIndexes.java

The TransportGroupIndexes.java class contains six static final variables that are used throughout the RADTRAD code to distinguish between the five different transport groups: noble gases, elemental iodine, organic iodine, aerosol iodine, and general aerosols. The final mnemonic value is the transport group count. These variables are set to integer values of zero through five, respectively.

3.5 Package timestep

The timestep package is the central repository for the classes and interfaces associated with the RADTRAD timestepping process. The RADTRAD timestepping procedure varies depending on many factors. The different procedures are broken up into different classes and interfaces. These are shown in Figure 3-6. Ultimately, these classes and interfaces contain the methods that determine each timestep for a RADTRAD simulation. The following sections describe the structure and role of each major class and interface in this package.

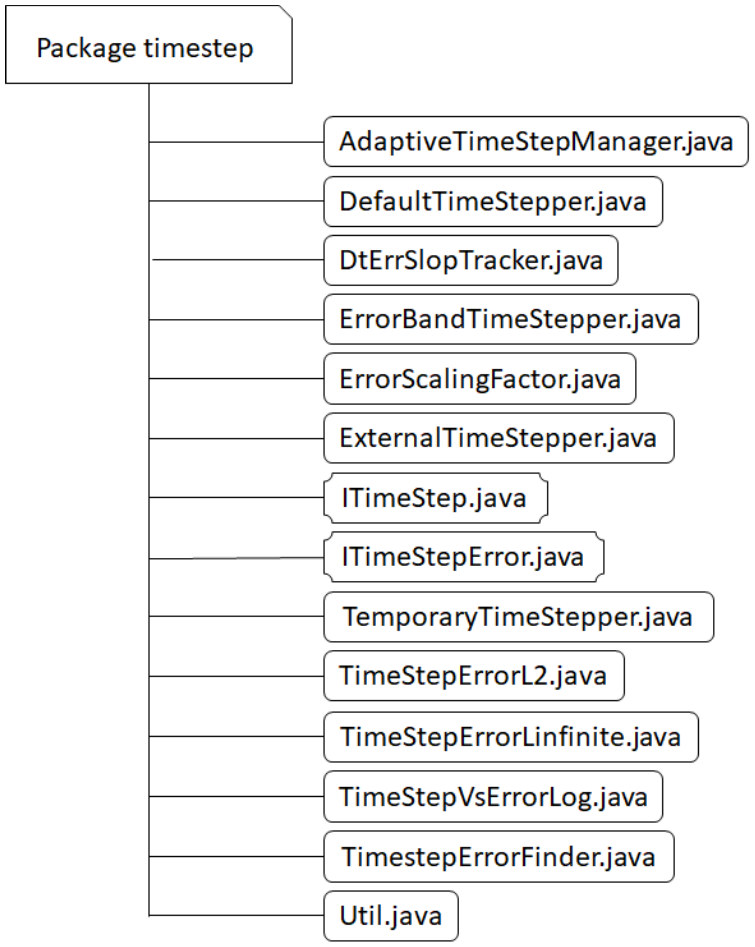


Figure 3-6 Illustration of the java files contained in the timestep package

3.5.1 AdaptiveTimeStepManager.java

The AdaptiveTimeStepManager.java class contains the method that determines which timestep handler to use and swaps the handler as needed during the progression of a RADTRAD simulation. This class is instantiated once at the beginning of the radcalc method only when the adaptive timestepping option is turned on by the user. This class implements the ITimeStep.java interface which contains only the dt_size method, as described in Section 3.5.7. The dt_size method returns the timestep size to be used for a single timestep. In this implementation, different timestep sizes are returned depending on various timestep conditions, such as error and simulation time. Based upon these conditions, other implementations of the dt_size method are called.

3.5.2 DefaultTimeStepper.java

The DefaultTimeStepper.java class contains the method that automatically adjusts time step size and limits the size based upon simulation time. This class is instantiated once at the beginning of the radcalc method. This class implements the ITimeStep.java interface which contains only the dt_size method, as described in Section 3.5.7. The dt_size method returns the timestep size to be used for a single timestep. In the default timestep scenario, no error measure is considered. As stated above, the timestep size is adjusted automatically based upon simulation time and simulation parameters, such as whether the daughter production option is turned on.

3.5.3 DtErrSlopTracker.java

The DtErrSlopTracker.java class contains the methods that are used to compare delta-t vs. error points to determine if decreasing the timestep size improves the error sufficiently. This class is instantiated in the ErrorBandTimeStepper.java class where this algorithm is used.

3.5.4 ErrorBandTimeStepper.java

The ErrorBandTimeStepper.java class contains the method that uses timestep halving or doubling to keep the timestep within an error band. The class is instantiated in the AdaptiveTimeStepManager.java class constructor. This class implements the ITimeStep.java interface which contains only the dt_size method, as described in Section 3.5.7. The dt_size method returns the timestep size to be used for a single timestep. This class's implementation of the method requires a region where the error decreases with a decrease in timestep size. If it detects a region where the error exceeds the error limit and is increasing as the timestep decreases over a few timesteps, it will return null, indicating that a new algorithm should be applied for some time. This implementation is called from the implementation of the dt_size method in the AdaptiveTimeStepManager.java class (Section 3.5.1).

3.5.5 ErrorScalingFactor.java

The ErrorScalingFactor.java class contains the methods that calculate an error scaling factor for error that decreases its importance when the change in dose is small relative to the overall simulation change in dose. This class is instantiated in the TimeStepErrorL2.java and TimeStepErrorLinfinite.java classes (Sections 3.5.10 and 3.5.11) where error is calculated.

3.5.6 ExternalTimeStepper.java

The ExternalTimeStepper.java class contains the method that is used to run two instance of RADTRAD concurrently with one being a timestep server and the other a timestep client. This class is instantiated once at the beginning of the radcalc method if the timestep server port option is turned on by the user. This class implements the ITimeStep.java interface which contains only the dt_size method, as described in Section 3.5.7. The dt_size method returns the timestep size to be used for a single timestep. This implementation is used for testing purposes to determine whether two instances of RADTRAD obtain identical results when the same time step size is used. For example, it has been used to test whether an instance of RADTRAD run in standard mode produces identical results as the adaptive time stepping mode when timesteps of the same size are used.

3.5.7 ITimeStep.java

The ITimeStep.java interface contains only the dt_size method, which returns the timestep size to be used for a single timestep. This interface is implemented by the following classes: AdaptiveTimeStepMananger.java, DefaultTimeStepper.java, ErrorBandTimeStepper.java, ExternalTimeStepper.java, TemporaryTimeStepper.java, and TimeStepVsErrorLog.java. These classes are described in Sections 3.5.1, 3.5.2, 3.5.4, 3.5.6, 3.5.9, and 3.5.12, respectively.

3.5.8 ITimeStepError.java

The ITimeStepError.java interface contains the calculateError method, which calculates the timestep error for an individual array. This interface is implemented by the following classes: TimeStepErrorL2.java and TimeStepErrorLinfinite.java. These classes are described in Sections 3.5.10 and 3.5.11, respectively.

3.5.9 TemporaryTimeStepper.java

The TemporaryTimeStepper.java class contains the method that steps forward using a given ITimeStep implementation for a specified number of steps, then returns null to indicate that it is time to attempt to switch to a new time stepper. The class is instantiated in the AdaptiveTimeStepManager.java class constructor. This class implements the ITimeStep.java interface which contains only the dt_size method, as described in Section 3.5.7. The dt_size method returns the timestep size to be used for a single timestep. Currently, this algorithm only uses the DefaultTimeStepper.java implementation. It is called when other adaptive timestep algorithms cannot be used.

3.5.10 TimeStepErrorL2.java

The TimeStepErrorL2.java class contains the method that calculates the error for an array of dose values based upon an L2 norm of the array values. This class is instantiated in the TimeStepErrorFinder.java class constructor if the L2 norm is to be used to calculate the timestep error. This class implements the ITimeStepError.java interface which contains the calculateError method, as described in Section 3.5.8. The calculateError method calculates the timestep error for an individual array. In this implementation, the error is calculated based upon an L2 norm as described in the RADTRAD Adaptive Time Step Algorithm report[1].

3.5.11 TimeStepErrorLinfinite.java

The TimeStepErrorLinfinite.java class contains the method that calculates the error for an array of dose values based upon an L-infinite norm of the array values. This class is instantiated in the TimeStepErrorFinder.java class constructor if the L-infinite norm is to be used to calculate the timestep error. This class implements the ITimeStepError.java interface which contains the calculateError method, as described in Section 3.5.8. The calculateError method calculates the timestep error for an individual array. In this implementation, the error is calculated based upon an L-infinite norm as described in the RADTRAD Adaptive Time Step Algorithm report[1].

3.5.12 TimeStepVsErrorLog.java

The TimeStepVsErrorLog.java class contains the methods that check for a timestep where a table of timestep size vs. error size is printed out. The class is instantiated in the Radtrad.java class and is used during the autodt method (described in Section 2.4). This class implements the ITimeStep.java interface which contains only the dt_size method, as described in Section 3.5.7. The dt_size method returns the timestep size to be used for a single timestep.

3.5.13 TimestepErrorFinder.java

The TimeStepErrorFinder.java class contains the methods that determine the timestep error. Depending upon various factors, the L2 or L-infinite errors are calculated. The classes containing the algorithms for these calculations are described in Sections 3.5.10 and 3.5.11, respectively. This class is instantiated once in the Radtrad.java class if timestep error or adaptive timestepping is turned on by the user. This class object stores the timestep error and information about the timestep error.

3.5.14 Util.java

The Util.java class is structured to contain utility functions for the time stepper classes. Currently, this class contains only the trimTime method which is a static method rounds the timestep to the nearest 180 seconds for simulation time. This method is called in the AdaptiveTimeStepManager.java and DefaultTimeStepper.java classes (Sections 3.5.1 and 3.5.2) after the untrimmed timestep has been calculated.

3.6 Package cmdline

The cmdline package is a small package containing one class that is used to parse the RADTRAD command line options. This package is illustrated in Figure 3-7.

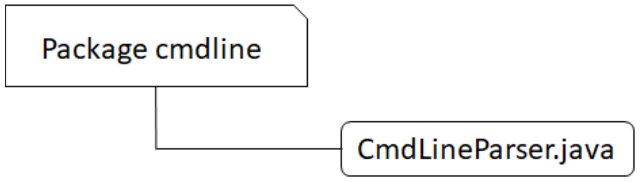


Figure 3-7 Illustration of the cmdline package and its one constituent class

The CmdLineParser.java class contains all the methods that parse the command line options. The class is instantiated once at the beginning of the Radtrad.java main method. In the Radtrad.java main method, all command line options are read in using the CmdLineParser.java object.

3.7 Package cmdserver

The cmdserver package is a small package containing two classes that can be used to run two RADTRAD simulations simultaneously. These classes are illustrated in Figure 3-8 and described below.

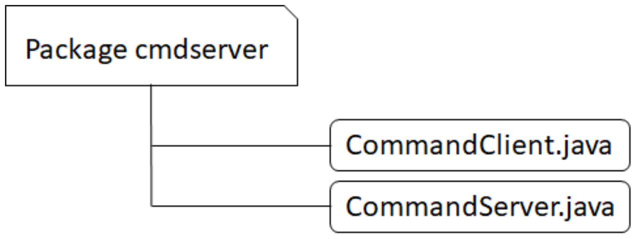


Figure 3-8 Illustration of the java files contained in the cmdserver package

The CommandServer.java class is a blueprint for a socket that connects to a command client which passes commands. The CommandClient.java class is its complement: a blueprint for a client that passes and receives commands from the server. The commands are requests for data. Currently, these classes are used to pass timestep information and run two instances of RADTRAD concurrently, as described in Section 3.5.6. The process can be thought of as follows: the server object contains timestep data from an original ongoing RADTRAD simulation. The client object requests timestep data from the server object and uses these timesteps to run a simultaneous simulation with the received timestep information. This process only occurs if the user enables the behavior.

3.8 Package restore

The restore package is the central repository for the classes and interfaces associated with the RADTRAD save and restore functionality. These are shown in Figure 3-9. The save and restore functionality is required in order to implement the adaptive timestepping and error calculation logic. In order to operate with an optimal timestep size, RADTRAD must be able to move forward in time, calculate timestep error, and then restore back to the original time state if timestep error is determined to be too high. The classes and interfaces that contains this functionality are described in the following sections.

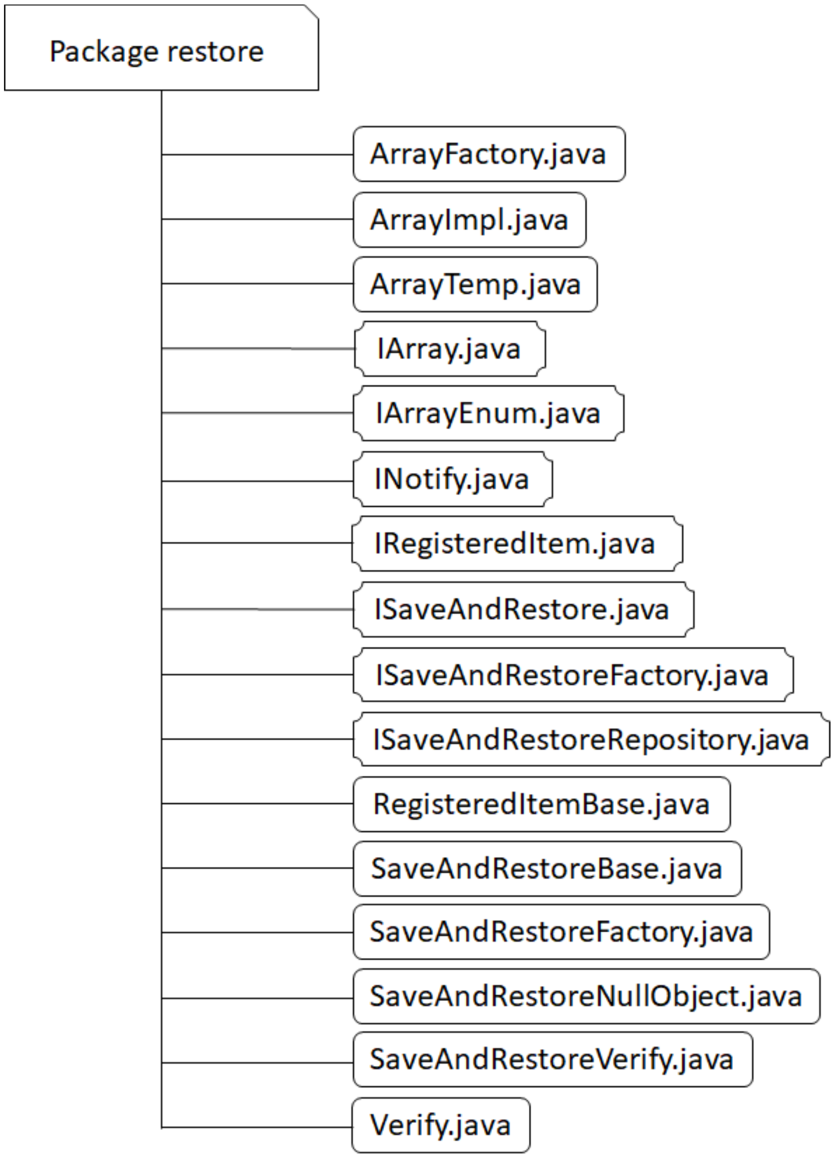


Figure 3-9 Illustration of the key java files contained in the restore package

3.8.1 ArrayFactory.java

The ArrayFactory.java class contains methods that are used in the SaveAndRestoreBase.java (Section 3.8.11) class to create IArray objects.

3.8.2 ArrayImpl.java and ArrayTemp.java

The ArrayImpl.java and ArrayTemp.java classes exists primarily to implement the IArray.java interface, discussed in Section 3.8.3. These classes also implement the ISaveAndRestore.java interface, described in Section 3.8.7, but these implementations are currently used in the code.

3.8.3 IArray.java

The IArray.java interface defines the get methods that allow each type of array to be accessed in the radcalc function. Arrays that need to be saved and restored in the RADTRAD simulation process are defined as type IArray in the radcalc method. An IArray contains a reference array containing the solution from the previous timestep, and an interim calculation array containing the current timestep solution. This interface is implemented by the ArrayImpl.java and ArrayTemp.java classes, described in Section 3.8.2.

3.8.4 IArrayEnum.java

The IArrayEnum.java interface defines a set of functions that are used to iterate through the different types of registered array items defined by the classes located in the SaveAndRestoreBase.java file listed in Table 3-2. These classes contain a method called values that returns the values of a registered item which are used for timestep error calculation.

3.8.5 INotify.java

The INotify.java interface defines the notify method. The notify method is used in RADTRAD to notify a listener that something is done. For example, notify is implemented in SaveAndRestoreBase.java to print a requested comparison of values when the comparison has been completed. This interface is implemented by SaveAndRestoreBase.java, SaveAndRestoreVerify.java, and Verify.java, described in Sections 3.8.11, 3.8.14, and 3.8.15 respectively. It is also implemented by the class SaveAndRestoreVerifyFactory contained in SaveAndRestoreFactory.java described in Section 3.8.12.

3.8.6 IRegisteredItem.java

The IRegisteredItem.java interface is a type that represents an item that has been registered for save and restore. It defines a method that returns an object that is used to iterate over the values in a saved array. This interface is implemented by the RegisteredItemBase.java class (Section 3.8.10) and the following classes: BooleanArrayRestore, DoubleArray2Restore, DoubleArray3Restore, DoubleArray4Restore, DoubleArrayRestore, and IntArrayRestore. These other classes are contained in SaveAndRestoreBase.java (Section 3.8.11). This interface extends the ISaveAndRestore.java interface.

3.8.7 ISaveAndRestore.java

The ISaveAndRestore.java interface defines the save and restore methods that are called to save current states and restore saved states for necessary IArrays in the radcalc method. This interface is implemented by the following classes described in Sections 3.8.2, 3.8.10, 3.8.11, 3.8.13, and 3.8.14 respectively: ArrayImpl.java, ArrayTemp.java, RegisteredItemBase.java, SaveAndRestoreBase.java, SaveAndRestoreNullObject.java, SaveAndRestoreVerify.java. The following classes also implement this interface: BooleanArrayRestore, DoubleArray2Restore, DoubleArray3Restore, DoubleArray4Restore, DoubleArrayRestore, and IntArrayRestore. These classes are contained in SaveAndRestoreBase.java. This interface is extended by the IRegisteredItem.java and ISaveAndRestoreRepository.java interfaces.

3.8.8 ISaveAndRestoreFactory.java

The ISaveAndRestoreFactory.java interface defines a method that creates an ISaveAndRestoreRepository.java object type. This interface is implemented by the DefaultSaveAndRestoreFactory, NullSaveAndRestoreFactory, and SaveAndRestoreVerifyFactory classes. These are static nested classes contained in the SaveAndRestoreFactory.java class.

3.8.9 ISaveAndRestoreRepository.java

The ISaveAndRestoreRepository.java interface defines the methods that allow each IArray type to be registered for save and restore purposes. This interface is implemented by the SaveAndRestoreBase.java, SaveAndRestoreNullObject.java, and SaveAndRestoreVerify.java classes.

3.8.10 RegisteredItemBase.java

The RegisteredItemBase.java class implements the IRegisteredItem.java interface. This abstract class is extended by the following classes: BooleanArrayRestore, DoubleArray2Restore, DoubleArray3Restore, DoubleArray4Restore, DoubleArrayRestore, and IntArrayRestore. These other classes are contained in SaveAndRestoreBase.java (Section 3.8.11).

3.8.11 SaveAndRestoreBase.java

The SaveAndRestoreBase.java class is the base class that implements the ability to register arrays and perform save and restore operations. This class implements the ISaveAndRestoreRepository.java interface. This java file contains a class for each different type of array that may need to be saved and restored in RADTRAD. Each of these classes extend RegisteredItemBase.java and contain save and restore operations. Table 3-2 describes the array type that each of these classes are designed to deal with.

Table 3-2 Description of the registered item types used for the classes in SaveAndRestoreBase.java

Class name	Array type	Number of dimensions
BooleanArrayRestore	Boolean	1
IntArrayRestore	integer	1
DoubleArrayRestore	double	1
DoubleArray2Restore	double	2
DoubleArray3Restore	double	3
DoubleArray4Restore	double	4

3.8.12 SaveAndRestoreFactory.java

The SaveAndRestoreFactory.java class contains three static nested classes that implement the ISaveAndRestoreFactory.java interface (Section 3.8.8). These nested classes are named the following: DefaultSaveAndRestoreFactory, NullSaveAndRestoreFactory, and SaveAndRestoreVerifyFactory. Each of these classes implement the create method which constructs and returns a new ISaveAndRestoreRepository.java object. The DefaultSaveAndRestoreFactory is constructed if the save and restore functionality is needed in the RADTRAD simulation. This is true if adaptive timestepping, timestep testing, or timestep error calculation is turn on. If not, the NullSaveAndRestoreFactory is constructed. The SaveAndRestoreVerifyFactory is used for RADTRAD testing purposes and also implements the INotify.java interface.

3.8.13 SaveAndRestoreNullObject.java

The SaveAndRestoreNullObject.java class is constructed when save and restore functionality is not needed. This class implements the ISaveAndRestoreRepository.java interface.

3.8.14 SaveAndRestoreVerify.java

The SaveAndRestoreVerify.java class is used to verify that the save and restore feature works correctly. It can be used to check that the inputs and outputs for a function are the same if restore is called and a timestep of the same size is repeated. Thus, if restore is called but results differ, it can be determined that the restore was not implemented correctly. This class extends the SaveAndRestoreBase.java class. This class also implements the INotify.java interface (Section 3.8.5) for comparison purposes.

3.8.15 Verify.java

The Verify.java class is used to detect differences in calculations in subsequent calls to a function when the same parameters are passed in after a restore is called. Thus, if the function results differ, it can be determined that hidden state variables are being used by the function that need to be saved before a restore can take place. This class implements the INotify.java interface (Section 3.8.5) for comparison purposes.

4 Bibliography

[1] S. L. Humphreys, T. J. Heames, L. A. Miller, D. K. Monroe, RADTRAD: A Simplified Model for RADionuclide Transport and Removal And Dose Estimation, U.S. Nuclear Regulatory Commission, Washington D.C., 1997.

[2] W. C. Arcieri, D. L. Mlynarczyk, L. Larsen, SNAP/RADTRAD 4.0: Description of Models and Methods, Information Systems Laboratories, Inc., Rockville, MD, 2016.

[3] L. Larsen, RADTRAD 4.5 & 4.6: Adaptive Time Step Algorithm and Results, Information Systems Laboratories, Inc., Idaho Falls, ID, 2019.