

# **An Object-Oriented Simple Climate Model**

**Bruce R. Barkstrom**  
**Atmospheric Sciences**  
**NASA Langley Research Center**  
**Hampton, VA**

## **Introduction – Objectives of This Paper**

In the classic approach to climate modeling, the continuous equations that govern the model's physical variables are discretized and placed in arrays that form the model's fundamental data structures. As the model runs, the model history is recorded in a sequence of these arrays. Understanding the model then proceeds by formulating hypotheses about causation – and searching through the stored history arrays to find support for those hypotheses.

In an object-oriented approach, the model is formulated in terms of objects that hide the internal structures of their variables. The model proceeds in time by allowing the objects to exchange messages that influence the internal state of the objects. As a result, diagnosing a model run is philosophically more akin to conducting an interrogation of the states and event sequences of the objects – which should make it easier to derive meaningful histories of the simulated events.

In this paper, we are not intent on deriving new numerical values for the future of Earth's climate. Rather, we want to describe how to design the computation in an object-oriented climate model. Because the computational framework is so different from the classic approach, we will simplify the physics to the point of dealing with a "Simple" climate model of the type used thirty to fifty years ago – a single-column model of the Earth as a whole.

We have three objectives:

1. To show how to derive and implement an Object-Oriented computer model from the physics that describe a simple, radiatively driven climate. The basic objects in this model consist of an atmosphere and a thermally-massive surface.
2. To compare data structures, model operation, and diagnosis for different implementations of the OO model.
3. To examine ways of designing the OO interfaces to make it easy to incorporate new physics into the model.

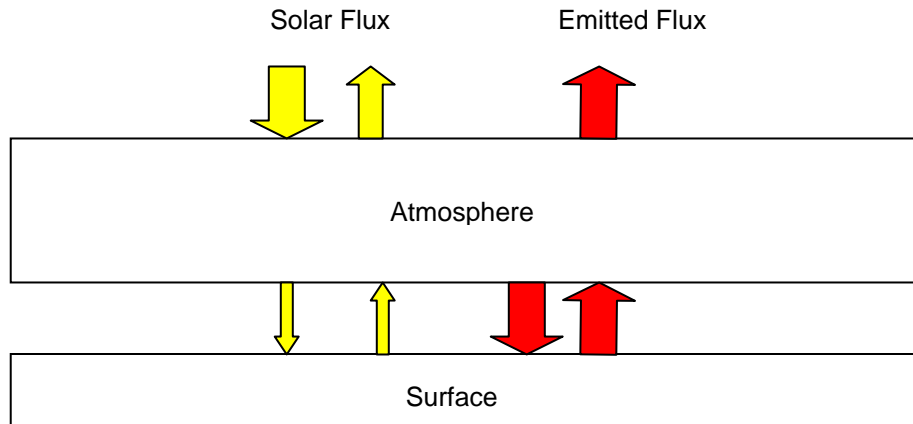
In the section that immediately follows, we will proceed as we might in the classic paradigm, deriving the equations that govern the model and then implementing them in a FORTRAN-like manner. Then, we will derive the OO form of the model.

## **Physical Basis of the Simple Climate Model**

Figure 1 shows the components of the model, which consists of an atmosphere and a surface. In the model the Earth's surface receives downwelling solar energy, reflects some of that energy and absorbs the remainder. The surface also receives downwelling infrared energy from the

atmosphere and emits a similar kind of energy back to the atmosphere. The surface has a temperature,  $T_s$ .

The atmosphere has a temperature,  $T_A$ . This part of the model is translucent to sunlight, both reflecting and transmitting this band of electromagnetic energy. In the longwave part of the spectrum, the atmosphere has an emissivity that allows the atmosphere to radiate to space and to the surface below.



**Figure 1. The Surface and the Atmosphere as fundamental objects in our model.** The arrows show the energy flows that govern the temperature of the system – the ones on the left are flows of solar energy; the ones on the right are flows of emitted energy.

### Equations for Solar Energy Transfer

In the shortwave part of the spectrum, where sunlight is the dominant energy source, the atmosphere can both reflect and transmit energy – although that flow is not directly related to the temperature of the atmosphere. Figure 2 provides labels for each of the solar energy flows in figure 1, allowing us to parameterize the flow in terms of the incident solar flux (averaged over a day),  $F_0$ . We assume that the atmosphere reflects a fraction,  $f$ , of the flux incident on either top or bottom. We also assume that it transmits a fraction,  $tr$ . At the surface, we assume that the fraction of incident flux that is reflected is  $a_s$ .

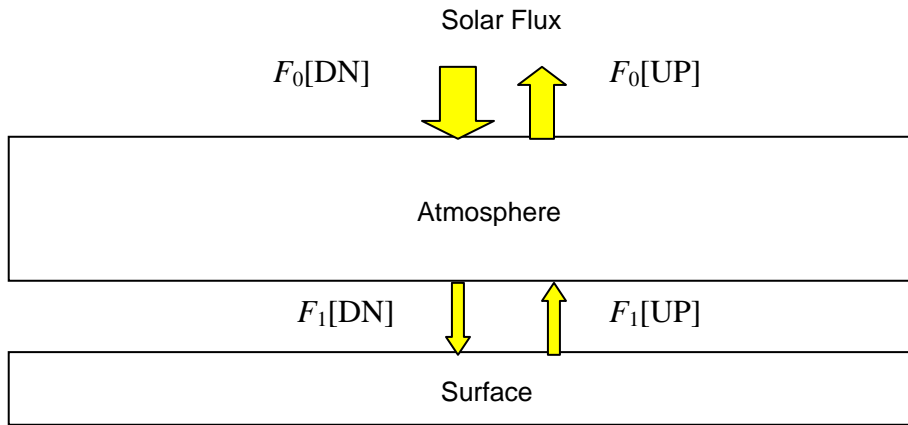
The governing equations are thus

$$F_0[\text{DN}] = \frac{1}{4} 1365 \text{ W/m}^2 \text{ [given]}$$

$$F_0[\text{UP}] = r F_0[\text{DN}] + tr F_1[\text{UP}]$$

$$F_1[\text{DN}] = tr F_0[\text{DN}] + r F_1[\text{UP}]$$

$$F_1[\text{UP}] = a_s F_1[\text{DN}]$$



**Figure 2. Solar Energy Flow Components.** Solar Flux coming down at the Earth's surface is transmitted through the atmosphere, where some of that flow is reflected. Solar flux coming out the top of the atmosphere consists of both flux reflected from the surface that is transmitted back through the atmosphere and flux reflected directly from the atmosphere.

We can use elementary algebra to solve these equations, finding that

$$F_1^S [\text{DN}] = (1 - r a_S)^{-1} tr F_0[\text{DN}]$$

$$F_1^S [\text{UP}] = a_S (1 - r a_S)^{-1} tr F_0[\text{DN}]$$

$$F_0^S [\text{UP}] = [r + tr a_S (1 - r a_S)^{-1} tr] F_0[\text{DN}]$$

### Equations for Emitted Energy Transfer

In the longwave part of the spectrum, the lack of reflection makes the equations somewhat simpler, as well as more directly coupled to the two temperatures of interest:

$$F_1^E [\text{DN}] = e \sigma T_A^4$$

$$F_1^E [\text{UP}] = \sigma T_S^4$$

$$F_0^E [\text{UP}] = (1 - e) \sigma T_S^4 + e \sigma T_A^4$$

$e$  is the emissivity of the atmosphere.

### Equations for the Atmosphere and Surface Temperature

We treat the surface and the atmosphere as lumped heat capacities. This means that the equations that govern the temperature change of the atmosphere and the surface are

$$C_S d T_S / dt = (F_1^E [\text{DN}] - F_1^E [\text{UP}]) + (F_1^S [\text{DN}] - F_1^S [\text{UP}])$$

and

$$C_A \frac{dT_A}{dt} = -F_0^E [\text{UP}] + (F_0^S [\text{DN}] - F_0^S [\text{UP}]) \\ - [(F_1^E [\text{DN}] - F_1^E [\text{UP}]) + (F_1^S [\text{DN}] - F_1^S [\text{UP}])]$$

Generally, we expect  $C_S \gg C_A$ .

## Standard Paradigm Method for Solving the Simple Climate Model

In the approach we identify as the “standard paradigm” method, we treat the two coupled equations we just derived as an ordinary differential equation, with known initial conditions. For small time steps, the assumption that the heating and cooling rates are nearly constant within the time step leads to a pair of equations:

$$T_S(t_i) = T_S(t_{i-1}) + \text{Net Flux [Sfc, } t_{i-1}] \Delta t / C_S$$

and

$$T_A(t_i) = T_A(t_{i-1}) + \text{Net Flux [Atm, } t_{i-1}] \Delta t / C_A$$

The author is aware that there are better finite difference solution methods for this problem – our concern in this paper is not so much accuracy as computational organization.

The solution algorithm is quite straightforward at this point. Listing 1 suggests a FORTRAN-like solution procedure in which the two temperatures are regarded as elements in a two-dimensional vector. The output is a one-dimensional array of these 2-vectors.

```
time      : real := 0.0;
TS, TS0  : real;
TA, TA0  : real;
-- Initialize temperatures
TS := TS0;
TA := TA0;
for time in 1 .. Final_Time loop
    -- Compute new values and output
    TS := TS + (SFC_Net_Flux/CS)*Deltat;
    TA := TA + (ATM_Net_Flux/CA)*Deltat;
    write (time, TS, TA);
end loop;
```

**Listing 1. Pseudo-code for FORTRAN-like Solution to the Simple Climate Model.**

## Moving to an Object-Oriented Approach for the Simple Climate Model

In this section, we want to outline the approach we have taken to implementing the Simple climate model in an object-oriented framework. We are particularly interested in this approach because it provides a systematic way of incorporating concurrency into the design. What we propose is not a unique physical decomposition of the problem (although it is difficult to think of a different partitioning for the simple physical problem we use as an example). Rather, it is a systematic approach to the systems engineering for designing a “real-time” infrastructure to help solve such models, drawing heavily on the work of Douglass [1998].

### Explanatory Notes on Objects

If we ask “what are objects”, Douglass [p. 15] suggests “An object is a cohesive entity that has attributes, behavior, and (optionally) state.” Then, he notes that “Attributes .. refer to the data encapsulated within an object.” [p.22] We can divide object behavior into two categories: passive and active. “Passive objects supply behaviors for other objects; that is, they provide services that other objects may request.” [p.23] “Active objects form the roots of threads and invoke the services (behaviors) of the passive objects.” [p. 23] “The logical interface between objects is done with the passing of messages. A message is an abstraction [and encapsulation] of data and/or control information passed from one object to another.” [p. 25]

Clearly, we need to capture the key objects and messages in the initial stages of an OO design. As part of this design work, we will group objects into associations. This is important because “Messages occur only between objects that have an association.” [p. 25] Later, we will have to specify how an object behaves when it receives a message. We will also have to design the protocols for each object’s interface – “the façade that the object presents to the world.” [p. 26]

We also want to identify object responsibilities. “The responsibilities of an object are the roles that it serves within the system.” [p. 27]

Objects are autonomous and concurrent – unless clearly spelled out differently. “**It is theoretically possible for each object to run on its own processor.**” [p. 27, emphasis added] Hardware limitations may prevent us from achieving this goal, of course. Usually we organize the object executions so that they work within threads, where “A thread is a set of operations executed in sequence.” [p. 28] Most commercial operating systems, including the varieties of UNIX, Linux, and Microsoft’s Windows NT and 2000 support threads.

Concurrent threads can run on separate processors, meaning that the relative speeds with which they process are uncoupled. On the same processor, we must rely on pseudo-concurrency provided by the underlying operating system, or write our own executive. These concurrency mechanisms allow the threads to progress more or less independently.

“Taken together, the characteristics of objects, such as attributes, behaviors, interfaces, encapsulation, and concurrency, allow each object to act as a separate entity – an autonomous

machine. This machine does not have to be very smart, but must own its responsibilities and collaborate with other machines to achieve some higher order goals.” [p. 28]

“This leads to a fundamental rule of object systems – **distributed intelligence**. Each object, however lowly and simple, has enough brains to manage its own resources and perform its own behaviors. This is different from functional decomposition in which it is common to have elaborately complex master subroutines that know everything about everybody. The truth of the matter is that it is far easier to construct dozens of semismart objects than a single really smart object that knows everything.” [p. 28]

### **An Outline of Object-Oriented Design for Real-Time Systems**

We do not expect to design climate models to run as embedded real-time systems of the type that control aircraft or automobiles (where the computers actually control mechanical systems and are distributed throughout the machine being controlled). However, there are several analogues between such machines and an Object-Oriented climate model – the desire for concurrency and the possibility of designing for distributed systems being foremost. Thus, we will apply the systematic design approach used for these complex machines as the framework for designing an OO climate model. Douglass [1998, 2000] provides a readable exposition of this design approach. Table 1 summarizes the design steps we recommend following.

It is also important to allow for design iterations. As we recount below, it is often difficult to get the design correct the first time through the process – particularly with respect to the design control architecture. On the first try, it is likely that the model will be conceived as having a “Master Controller” that governs “Slave Processors”. That danger is particularly present in modeling coming from the classic paradigm, where the model proceeds to move along in lock-step with the master control object synchronizing the processing in the slave objects.

As we will see, it is usually preferable to distribute control throughout the objects, so that the “span of control” of object associations is more or less uniformly partitioned over the full range of active objects. From a performance point of view, this partitioning is related to the need for global data stores and for message passing bandwidth. A centralized control scheme is likely to be the result of thinking in terms of large matrices and will require large bandwidth in order to communicate with this global structure – even if the individual processes are dealing with limited portions of the global structure.

In addition, visualizing the problem as having a single “Master Controller” that synchronizes all of the “Slave Processors” limits the design’s ability to operate with different process rates in different objects. This is likely to be an important feature of an object-oriented approach to modeling because it allows rapid local processes to be handled more appropriately than will be the case if we force modeling at the slower speeds of a larger object – or force the large scale object to keep up with the rapid pace of the smaller object.

In the material that follows, we illustrate the design process for our Simple climate model. We will follow the outline in Table 1, using the Unified Modeling Language (UML) to document what we are doing and to communicate the design to users.

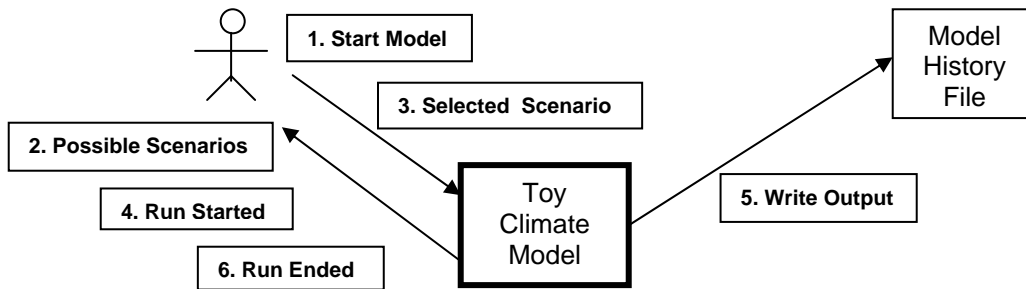
**Table 1. Suggested Object Oriented Design Steps**

1. Create a system context diagram, showing how the users of the model will interact with it.
2. From the diagram, identify the external objects, such as The user who initiates the model run, monitors it, and examines the results Files that receive output Files that log activities Files for diagnostic work and validation GUI for activity monitoring GUI for output GUI for displaying diagnostic and intermediate results
3. Identify Context Level Message Flows and Properties
4. Specify External Events Create an Event List Design External Timing Requirements (to allow creation of a performance budget)
5. Build Use Cases and Scenarios Use Case Diagram Identify Use Cases Build Scenarios Create Sequence Diagrams Create Collaboration Diagrams Provide a User Manual Outline
5. Identify Objects in the model (“underline nouns”)
6. Describe Message Flows between internal objects
7. Provide Object Attributes
8. Derive Equations for Object Methods
9. Define Class Relationships and Associations – Architectural Design –“Master Controller” Architecture vs –“Distributed” Architecture
10. Mechanistic Design (partitioning of computation and I/O)
11. Detailed Design

## Placing the Model in the Context of Its Environment

In real-time system design, it is important to understand which things belong to the system and which do not. For a model of physical processes, the problem does not seem quite as confusing at the outset. We usually visualize a model as having a single user operating on a single computer (even if that computer is a cluster of workstations) and having a relatively simple collection of devices for storing results and presenting output. Of course if we allow our horizon to expand to include models that run in a geographically distributed configuration, the context is not quite as simple.

The first tool we use in our design is the **System Context Diagram**. This diagram should show the key actors and logical devices that interact with the system. It also needs to show the messages that must flow from the actors and devices into the system and from the system back out. We also want to identify important characteristics of the message patterns, such as synchronism and arrival patterns. Figure 3 and Table 2 provide the appropriate documentation of our initial understanding of the way we want the model to work.



**Figure 3. Context Diagram for Climate Model with Single User.** The stick figure identifies the user. The central box indicates the model. The box on the right indicates the file that records the history of a model run. The numbered boxes indicate the messages that need to flow into and out of the model. The numbers in the box indicate the approximate order in which the message will be sent. The properties of these messages are described in Table 2.

**Table 2. Context Level Event List – No Exceptions**

Event ID	Event	System Response	Direction Pattern	Arrival Pattern	Synchronization	Required Response
1	Start Model	Request possible model scenarios from the Model	IN	One-time	Asynchronous	0.01s
2	Possible Scenarios	Send user message identifying possible model scenarios	OUT	One-time	Asynchronous	0.5s
3	Selected Scenario	Initialize objects	IN	One-time	Asynchronous	0.01s
4	Run Started	Set objects running; Open Model History File	OUT	One-time	Asynchronous	0.01s
5	Write Output	Append new value in Model History File	OUT	Episodic	Asynchronous	0.01s
6	Run Ended	Close Model History File	OUT	One-time	Asynchronous	0.1s

Figure 3 is intended to identify the major actors that interact with the model. The Simple Climate Model is an extremely simple system. If we were trying to design a distributed version built to run interactively on computers housed in geographically dispersed locations, the Context Diagram could become considerably more complex. We would still only describe the communications between the model and its environment. However, in that more complex situation, the outbound messages could include model-monitoring messages that would allow different classes of users to see which objects were running in the various system locations.

The table that describes the context level messages is also needed to record and keep straight the various categories of messages the system interchanges with the outside world. Each message arises in connection with a particular event and evokes a particular response from the system. We need to record the direction in which the message is going, whether it is a single event or one that happens periodically. We also need to know if that message needs to be synchronized with other messages to or from the system. The final column is probably less important for models than it would be for designing the avionics system for an airplane. However, the information in this column is needed to establish the performance budget for the system. This column will be much more important when we need to deal with geographically dispersed systems where network latency could be a critical component of system response.

Note also that we do not incorporate events that generate exceptions in this initial design phase. Exception handling is very important to an object-oriented system, particularly one that may have to deal with network transmission glitches or other faults. We will need to build a list of exceptions and identify the proper response to them. However, specifying exceptions belongs to a much later part of the system design.

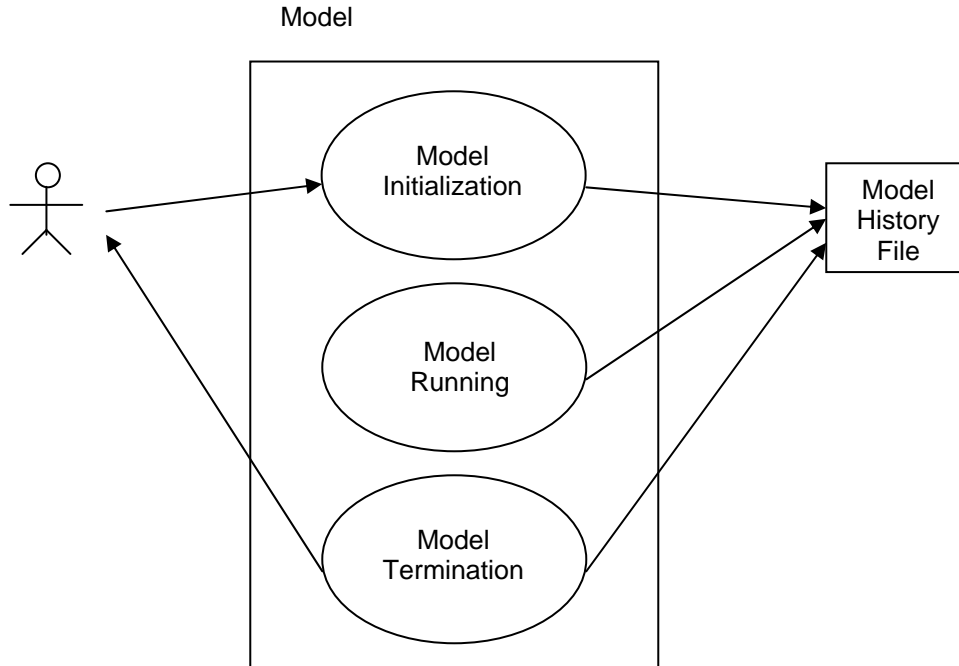
## **Building Use Case Scenarios and Outlining the User Manual**

A “Use Case” is an example of a model activity that is intended to elucidate how the system operates. In the classic model paradigm, use cases are probably superfluous because the way the system works is very simple:

- Initialize
- Compute
- End

In an object-oriented model, however, there may be a number of different phases, each of which has its own collection of objects. For these different phases, we need to record how the various system components interact. This kind of specification gives us an initial clue about what kinds of messages we might expect and serves as a systematic method of developing tests that we can apply to the model as a whole system. The classic modeling paradigm concentrates primarily on ensuring correct numbers emerge from the model. In an object-oriented model, we also need to make sure that the model interacts with its environment in accord with its design.

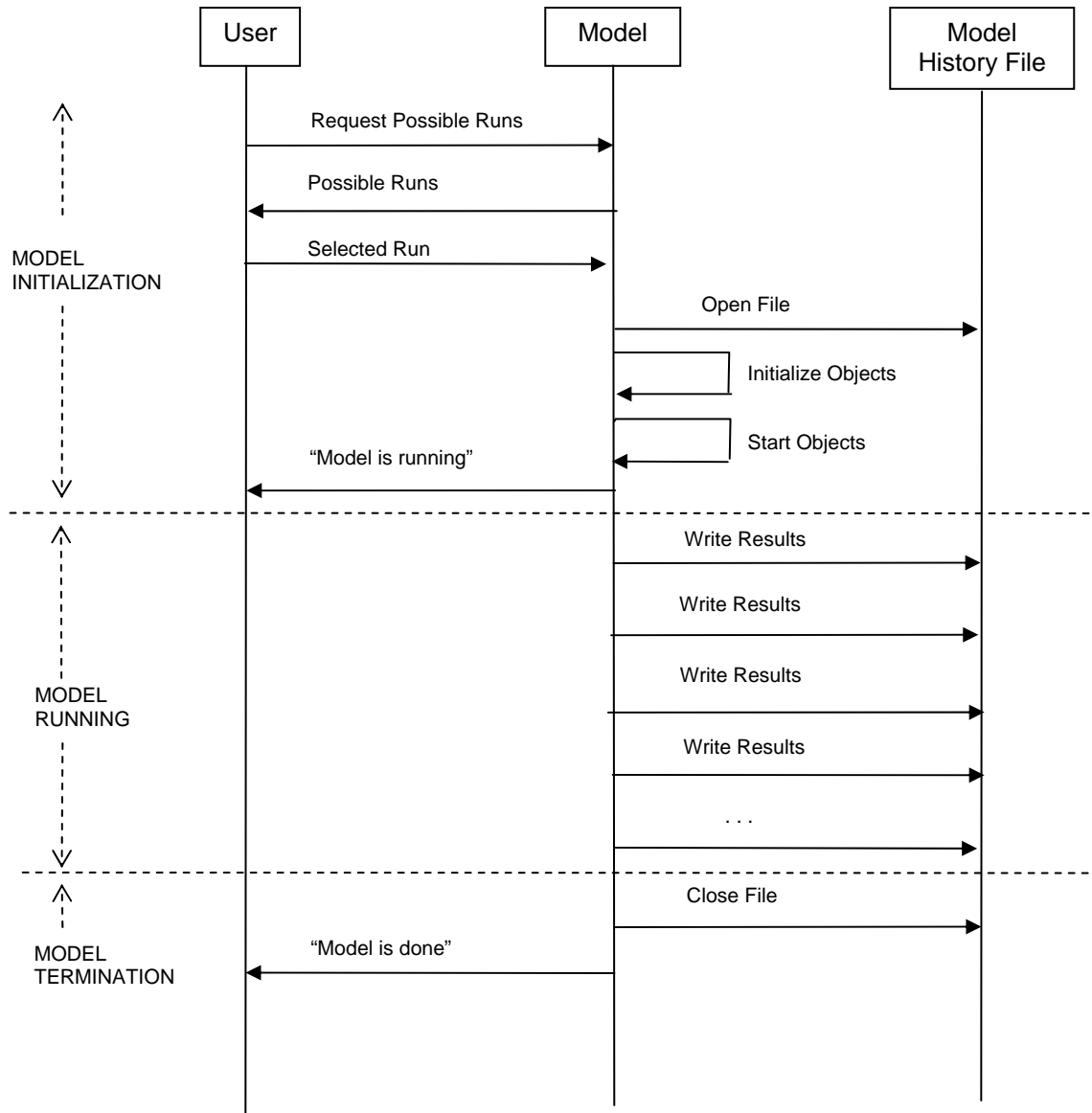
To document the use cases for the model, we are allowed to look at a simple version of the contents of the model. We want to identify the actors in each use case (which may include some of the objects inside the model). We also want to identify the actions that we expect them to perform. In UML, there is a standard diagram for recording the actors, the **Use Case Diagram**. Figure 4 shows this diagram for our simple model. There is also a corresponding **Use Case Scenario** table that provides more detail on the expected actions. Table 3 provides this information for our model. As we will see shortly, this table will allow us to write out a **Sequence Diagram** that shows the order in which the scenario’s actions should be performed. Figure 5 provides this system-level diagram for our model.



**Figure 4. Use Case Diagram for the Simple Model.** The Use Case appears in the circle within the model. The user appears on the left as an external actor, as does the Model History File. The arrows indicate the direction of message flow during the scenario.

**Table 3. Use Case Scenarios**

Use Case	Use Case Description	External Actors	Actions
Model Initialization	User initiates model, so Model initializes objects and opens Model History File	User	<ol style="list-style-type: none"> <li>1. User requests possible model run.</li> <li>2. User receives list of possible model runs.</li> <li>3. User chooses a model run.</li> <li>4. Model opens Model History File.</li> <li>5. Model initializes internal objects and sets them into motion.</li> <li>6. Model notifies User that model is running.</li> </ol>
Model Running	Model writes results of intermediate calculations to Model History File	Model History File	<ol style="list-style-type: none"> <li>1. System appends results to Model History File.</li> </ol>
Model Termination	Model closes Model History File and notifies User that the model run is complete	User Model History File	<ol style="list-style-type: none"> <li>1. Model closes Model History File.</li> <li>2. Model Notifies User of end of model run.</li> </ol>



**Figure 5. Top Level Sequence Diagram.** In this diagram, time moves downward along the vertical lines. The horizontal arrows show the messages that pass to and from each of the actors in the system. We do not show absolute timings for these messages, but the sequence needs to occur in the order shown.

Although it may seem strange, we are now in a position to outline the User’s Manual for the model. This is more beneficial than it might seem on the surface because it will help us identify actions we want to include in the model’s objects. For example, if we want the model to let us follow the time history of temperature as a result of perturbing the solar “constant” or of the atmosphere’s emissivity, we can incorporate the user control of these perturbations into the model scenarios we allow the user to run. Simply noting these options as possibilities now will allow us to make sure we incorporate those perturbations into the model as we move downstream with the full design. Figure 6 shows the outline for the model as we might conceive it now.

## Preliminary Outline of a User Manual for the Simple Model

### To Start the Model

1. Start the executable
2. Request possible model runs
  - a. Show history from specified initial atmospheric and surface temperatures.
  - b. Show history from a specified perturbation to the solar constant (using a perturbation that rises to its full value with an exponential increase).
  - c. Show history of a specified perturbation to the atmospheric opacity (using a perturbation that rises to its full value with an exponential increase).
  - d. Show a history of atmospheric and surface temperatures in response to a joint perturbation of solar constant and atmospheric opacity.
3. When the model parameter selection dialog box appears, fill in appropriate values for initial temperatures, solar constant, atmospheric reflection and transmission, and surface albedo. Also choose proper file and path for the Model History File. Set model running.
4. Verify that model is set to run by clicking on “Run” button.

### At the End of the Model Run

1. Verify that the model has completed and note location of Model History File.
2. Examine data contained in the Model History File.

**Figure 6. Preliminary Outline of a User Manual for the Model.** Note that step 2 of starting the model helps to specify the properties of objects we need to specify. Also note that this description assumes that the control of the model is done through a GUI environment, rather than simply dealing with ASCII file input. If we also wanted to provide interactive graphical displays of the model history, the preliminary outline of the User Manual can help identify key features of the graphical interface we would want to include.

## Object Design for the Model

The next step in the procedure is to identify the objects that belong in the model. Douglass [1998, 2000] provides a number of strategies to help in this process. For our purposes, we suggest starting with his “Underlining the Noun” strategies to identify the objects that may seem passive at the start of the design. In the material that follows, we follow a natural psychological sequence that creates a single “Master Controller” to run the model. While this is probably an inappropriate final design strategy, it is the author’s expectation that readers will benefit from seeing the evolution of the model design when we correct this mistake. As we have already commented, a mature object-oriented design is probably better conceived as ensuring a collection of autonomous machines than as a centrally controlled process. However, it will not always be obvious how to achieve the appropriate degree of autonomy when the design starts. The result is that we suggest an iterative design process, in which the design starts with an architecture that may have many passive objects and a few controllers. As the iteration proceeds, the designer should begin to consider how to spread intelligence through the objects, so that there are more active (and autonomous) ones and fewer passive ones.

In more advanced stages of design, it should be possible to introduce objects that have a transient life in the model. For these objects, we expect one class of objects to detect the need for one or more of the transient objects. The initiating class requests instantiation of the appropriate number of transient class objects, which are spawned off to perform their work and communicate to the appropriate elements of the model. Of course, the transient objects can end their life and disappear when they complete their part of the calculation. In many ways, this approach to modeling resembles computer games that have avatars who have an appropriate amount of intelligence in reacting to their environment. In the course of the game, an avatar is called into being endowed with certain attributes. As the game progresses, the avatar has experiences, interacts with its environment (which may change the avatar's attribute values and change the environment), and may die. Thinking of the objects in the model as characters in the model's history is probably a powerful metaphor for putting the model designer in the "appropriate frame of mind" for this work.

### **An Inappropriately Centralized Initial Object Design**

As we commented above, we start with an inappropriate initial design that contains many passive objects controlled by a single "Master Controller". We expect this approach will be common for inexperienced object designers. Almost by nature, we start by thinking of objects as passive, "noun-like" entities. Only after we look at the design in some detail do we realize that we can give the objects life and let them assume more responsibility.

The design items we need for this work include the following:

- List of passive objects
- List of active objects and their functions
- Object Associations and corresponding Message Flows
- Active Object Sequence Diagram (including Internal Objects)
- Passive Objects and Their Attributes (Object Diagrams)
- Active Objects
- Class Diagrams
- Class Relationships and Association Diagrams
- Active Object (Task) Procedure Descriptions

We present this list in the order in which we will use these items in producing a design. While this list may seem overly complex, the author can assure potential model designers that it saves work in the long run because each using each item is actually a small step that encourages systematic thinking about the model and how it works. In addition, by recording the design work as it proceeds, the designer can more easily revise the design as understanding improves.

It is a relatively small step to go from the last item in this list to working code for the model. Before writing out the code, however, we suggest stepping back from the design and considering architectural revisions that will improve its overall operation. In this document, we will therefore move to the point of providing pseudo-code for the Active Object Procedure Descriptions. However, we will not go all the way to code because the revision process will suggest a way to reduce the number of objects and to increase their autonomy.

### *List of passive objects*

Although embedded, real-time systems can present difficult problems in determining the appropriate objects to include in the system, the model we are building presents an easier target. We can examine the equations we derived for the “FORTRAN-like” version of the model and extract the important variables we can think of as objects. Basically, this approach produces a list of nouns. For the model so far, the list of passive objects would appear to be

- Sun
- Atmosphere
- Surface
- Atmospheric Optical Properties
- Surface Optical Properties (basically albedo)
- Radiative Flux Profiles
- Space (which receives reflected and emitted energy from the atmosphere and the surface)
- Model History File

### *List of active objects and their functions*

If we adopt the FORTRAN-like design of the model, we only need one active object, the “Master Controller”. This controller is charged with actually moving the computation along. It queries the passive objects for their current attributes, computes new values for those attributes, and then asks the passive objects to update their attributes to correspond to the new values. Of course, the controller may also receive messages from the user and needs to control writing to the Model History File. We can list the functions of the Master Controller object as

- Master Controller
  - Receives messages from the user to control the model
  - Sends messages to the user about the state of the model and its progress
  - Opens, Writes, and Closes the Model History File
  - Initializes the Passive Objects
  - Receives Passive Object Attributes
  - Computes new attribute values
  - Coordinates requests for services between all of the internal objects

This does seem like a lot of items to control. If we look back at the passive objects, we can see that there are three objects that change as the model proceeds: the Atmosphere (and its temperature), the Surface (and its temperature), and the Radiative Flux Profiles. This suggests that we extract the “Computes new attribute values” function from the Master Controller and create three additional active objects:

- Atmospheric Temperature Update Controller
  - Receives current atmospheric temperature
  - Receives current atmospheric flux divergence (difference between net flux at the top of the atmosphere and net flux at the bottom)
  - Computes a new atmospheric temperature
  - Sends new atmospheric temperature to update the Atmosphere object

- Surface Temperature Update Controller
  - Receives current surface temperature
  - Receives current surface energy balance flux (net flux at the surface)
  - Computes a new surface temperature
  - Sends new surface temperature to update the Surface object
  
- Radiative Flux Update Controller
  - Receives current atmospheric temperature
  - Receives current surface temperature
  - Receives current solar flux
  - Receives current atmospheric optical properties
  - Receives current surface optical properties
  - Computes current atmospheric and surface fluxes and flux divergences
  - Sends new flux profile to update the Radiative Flux Profiles object

When we add these controllers, we need to revise the structure of the Master Controller to recognize these new objects. As long as we are operating with the FORTRAN-like approach to synchronized time steps, we are likely to be inclined to give the responsibility for maintaining time synchronization to the Master Controller. The revised function list for this object should recognize this, as the following revision shows:

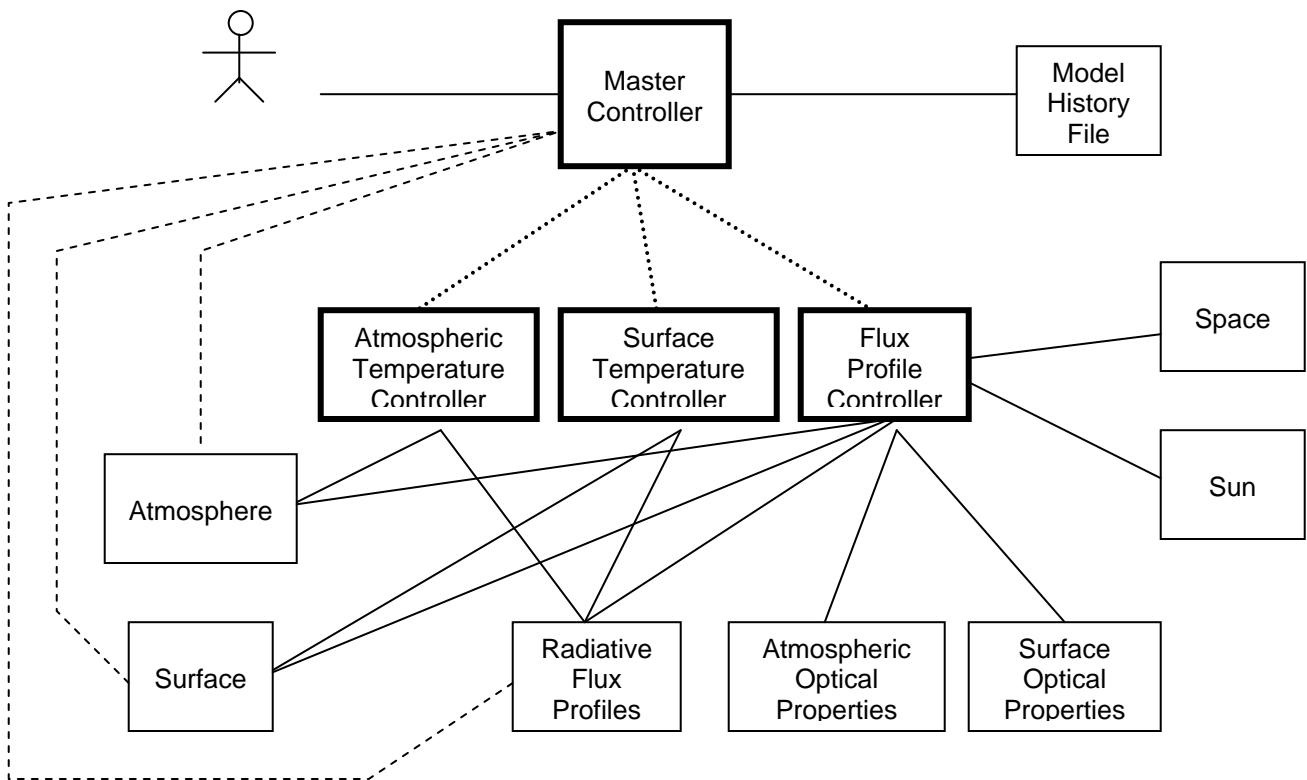
- Master Controller (Revision 1)
  - Receives messages from the user to control the model
  - Sends messages to the user about the state of the model and its progress
  - Opens, Writes, and Closes the Model History File
  - Initializes the Passive Objects
  - Receives Passive Object Attributes
  - Updates model time
  - Requests Flux Profile Update from the Radiative Flux Update Controller
  - Requests Surface Temperature Update from the Surface Temperature Update Controller
  - Requests Atmospheric Temperature Update from the Atmospheric Temperature Update Controller

Note that we no longer need to list the coordination function separately, since the updates are what the coordination needed to accomplish.

### *Object Associations and corresponding Message Flows*

We now have twelve objects in the model. The next design tool we invoke is the **Object Association Diagram** that identifies the message flows needed between the various objects. This diagram is the key to controlling the complexity of the model. It allows us to produce a testable design and gives us a tool for quantifying the message traffic within the system. In the next design step, we will look at the Active Object Sequence Diagram that will show the various message flows in a schematic form. By capturing the size of each message and the rate of message initiation, we can identify the I/O traffic between the objects. This will give us the first approximation to the performance properties of the overall system.

Figure 7 is the Object Association Diagram for our first try at the Simple Model object design. The complexity of the connections is one indication that we may want to repartition the passive and active object responsibilities.



**Figure 7. Object Association Diagram for the Simple Model.** This diagram shows the twelve objects we have identified in our first try at an object-oriented design for the Simple climate model. The heavily outlined objects are the active ones; the lighter outlined objects are passive. The lines between the object boxes represent the connections between the objects that need to communicate information of one sort or another. The solid lines indicate data streams that must operate to perform the model computations. The dashed lines on the left indicate the message flows that are required in order to provide the Master Controller data that it can dispatch to the Model History File. The lines consisting of '+' characters are control flows that request the appropriate controllers to update the values that they control.

Figure 7 is interesting for what it tells us about the control structure and data flow of this program. The solid lines indicate the data flows that the model needs in order to do its calculations. The dashed lines indicate the data flows the Master Controller needs in order to write values to the Model History File. These paths arise because the temperatures that are the key data for that history reside within the Atmosphere and Surface objects. They must be queried for the values of these temperatures when the Master Controller decides to send this information to the History File.

From this figure, we can derive the Sequence Diagram for the active objects. Figure 8 shows the results. Note that although figure 8 is similar to figure 5, the new figure shows how the active objects interact with each other – structure that is not available in the earlier figure. With the lock-step approach adopted by the use of the Master Controller, we see the individual time steps in this figure.

### *Active Object Sequence Diagram (including Internal Objects)*

Figure 8 shows the **Active Object Sequence Diagram**. We abbreviate some of the object names in this figure:

Atmospheric Temperature Controller (ATC)

Surface Temperature Controller (STC)

Flux Profile Controller (FPC)

We also divide the model operation into three major phases: Model Initialization, Time Stepping, and Model Termination. The control and data flow in the middle phase is repeated as long as the model needs to keep running. This figure also shows a distinction between control messages and data messages. The Master Controller sends control messages to the various objects that need commanding. The data messages can be sent from objects and are important in keeping the numerical calculations on track. Of course, the model could not operate without the command messages.

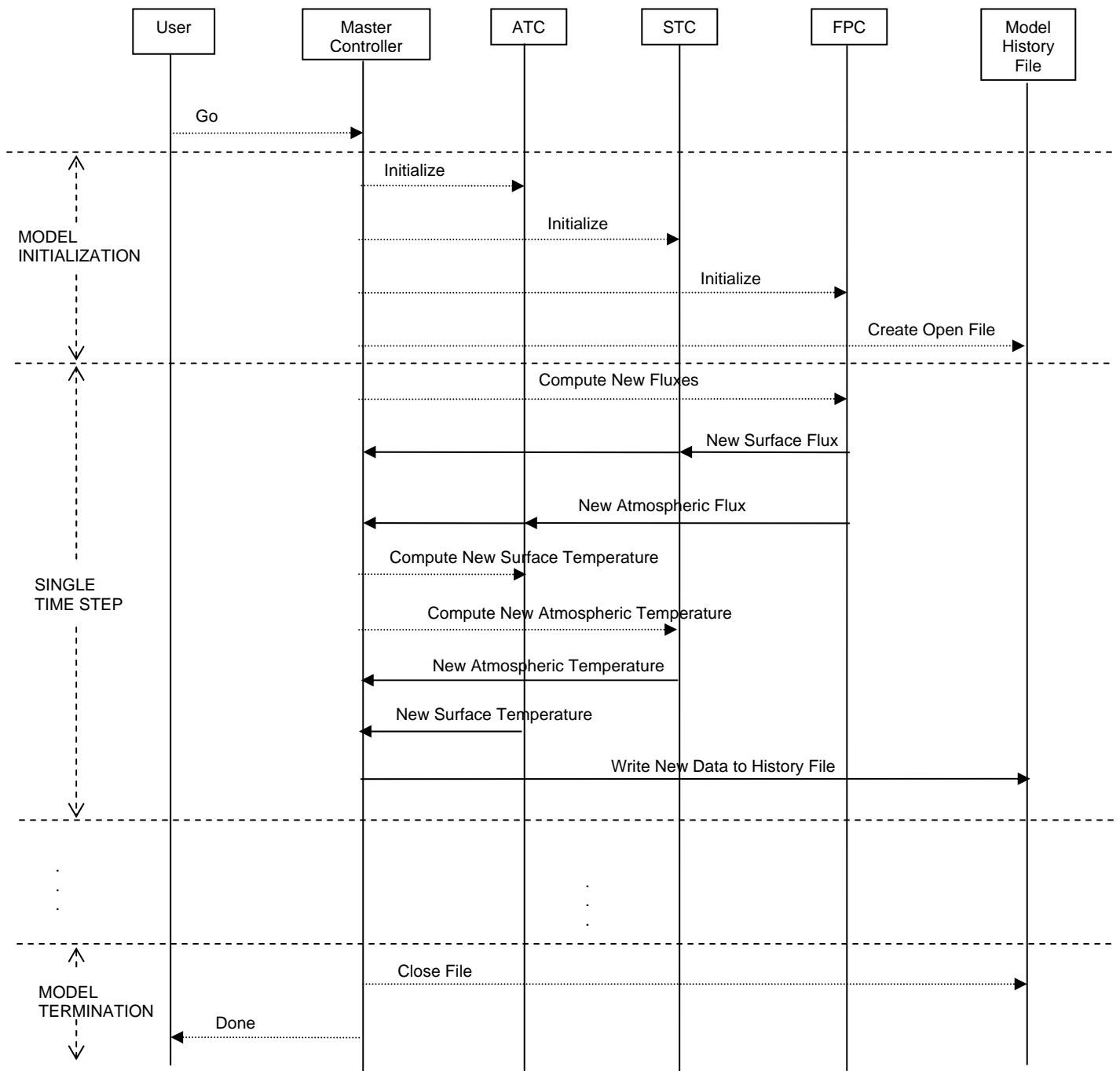
This sequence diagram is important for quantifying model performance because we can assign a data size to each message in the diagram. By adding up the data flows between the different objects, we can estimate the I/O rates required to handle the message passing. A first approximation to the time it takes the model to run through a time step is the sum of message passing time and computation time. As we will see later, this quantification is useful in checking how the model will perform against its performance budget.

### *Objects and Their Attributes (Object Diagrams)*

We are now in a position to document the object attributes (we'll wait until we get to the active objects before we deal seriously with the object methods, i.e. the functions each object has). Figure 9 (on the page after Figure 8) shows the objects and their attributes. Note that an object may have several attributes, so that one way of representing these components is to put them in a `record` (for Ada) or a `struct` (for C or C++).

The convention for documenting object attributes should be clear from Figure 9. The object is a box divided into three horizontal partitions. The top partition contains the object name. The middle partition contains the object's attributes. The bottom partition (which we don't fill in in this figure) contains the object's methods (or functions).

After we finish documenting the objects, we will provide an initial class diagram that identifies the message flows between the objects. This diagram will also provide some suggestions about how we might repartition our model to reduce the number of objects and make them more autonomous.



**Figure 8. Active Object Sequence Diagram.** This diagram shows the three basic phases of the Simple model's operation: Model Initialization, Time Stepping, and Model Termination. The dotted arrows indicate control flow, in which the initiating object sends a command to the receiving object(s). The solid arrows indicate data flow, in which the initiating object sends data to the receiving object(s). Note that the Time Stepping phase is repeated, with the Master Controller ensuring synchronization.

<b>Sun</b>
S0 : real; -- [W/m**2] -- Average Solar Irradiance -- ~ 365 W/m**2

<b>Atmosphere</b>
TA : real; -- [K] -- Atmospheric Temperature

<b>Surface</b>
TS : real; -- [K] -- Surface Temperature

<b>Atmospheric Optical Properties</b>
r : real; -- Atmosheric Reflectivity tr : real; -- Atmospheric Transmissivity ea : real; -- Atmospheric Emissivity

<b>Surface Optical Properties</b>
a : real; -- Surface Albedo

<b>Atmospheric Temperature Controller</b>
Time_at_Start_of_Interval : real; Time_Interval : real;

<b>Surface Temperature Controller</b>
Time_at_Start_of_Interval : real; Time_Interval : real;

**Figure 9. Model Objects and Their Attributes.** The attributes for each object are defined (using Ada syntax). ‘real’ variables are equivalent to C ‘long float’ or FORTRAN ‘double’. Although the attributes here are simple variables, we could include more complex structures, such as arrays, lists, etc. For these more complex types however, good design practice suggests making the attributes private, so that these types would only be visible through the object methods (or access functions).

<b>Flux Profile Controller</b>
Time_at_Start_of_Interval : real; Time_Interval : real;

<b>Master Controller</b>
Start_Time : real; -- [s] Max_Time_Interval : real; -- [s] File_ID : String;

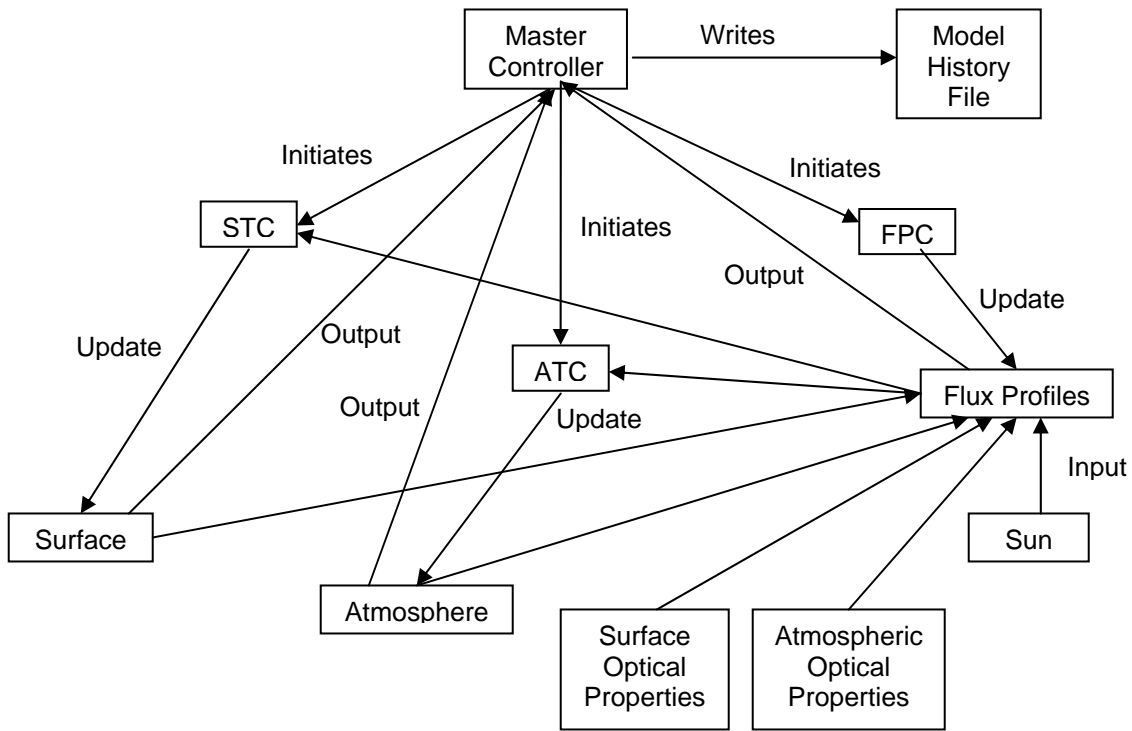
**Figure 9 (Cont'd). Model Objects and Their Attributes.**

### *Class Diagrams*

Now that we have a much clearer idea about the objects we are working with, we are in a position to refine our notion of the actions we want them to undertake. The next design tool we use is the Class Diagram, which helps us define the flow of control. We want to identify the initiating object and the recipient of the action. By using this new diagram, we will be better able to group the objects together into workable packages. Figure 10 contains the Class Diagram for the current partitioning of the Simple Model design.

In the next subsection, we will break down the overall class diagram into pieces that give us object associations. From these, we will be able to obtain a much cleaner model design with many fewer pieces.

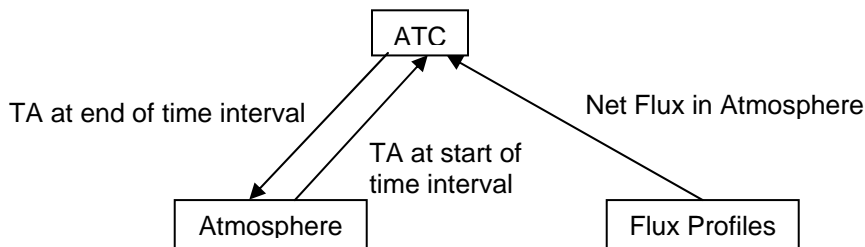
First, we need to look carefully at figure 10. We observe that there seem to be three major groupings of objects: an Atmospheric Package, a Surface Package, and a Flux Package. Each of these packages has a controller and each interacts with objects that contain attributes appropriate for the portion of the domain that we are trying to model. The only difficulty is that both the Atmospheric Controller and the Surface Controller need to obtain data from the Flux Profile, although that object belongs with the Flux Package. With this perspective, we can proceed further in identifying the Class Relationships and Associations that begin to identify the messages that need to be passed from one object to another.



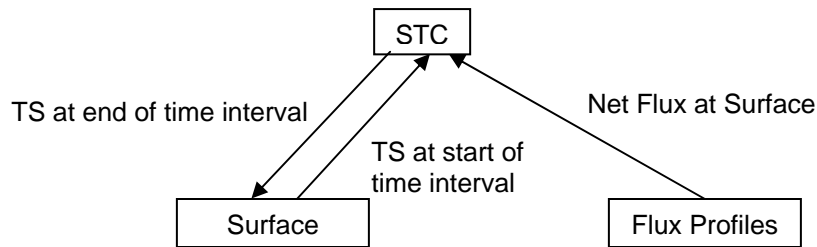
**Figure 10. Class Diagram.** In this diagram, we capture the classes of objects and their large-scale associations.

*Class Relationships and Association Diagrams*

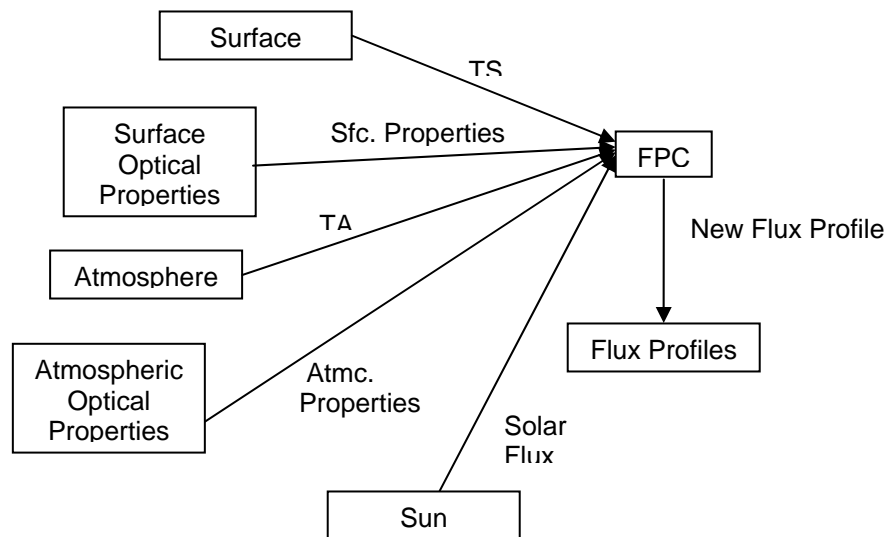
Figure 11a shows the objects we include in the Atmosphere Package, together with the messages that we expect this association needs to support. Figure 11b provides the equivalent grouping for the Surface Package. Figure 11c shows the somewhat more complex associations for the Flux Package.



**Figure 11a. Atmosphere Package Associations.** Owing to the limitations of the drawing package for Ms word, we use arrows on the linkages, although standard UML class and association diagrams would indicate the direction of message flow by the use of small arrowheads.



**Figure 11b. Surface Package Associations.**



**Figure 11c. Flux Package Associations.** Note that because the radiative fluxes depend only on the instantaneous state of the atmosphere, there is no need to update the flux profile. We only need the current state to do the computation. This makes the relationships on the right-hand side of this diagram unidirectional, rather than bi-directional.

### *Active Object (Task) Procedure Descriptions*

Douglass suggests that at this point one would begin to think about how associations of the type we've just described in Figure 11 would work. We can begin with the association in Figure 11a. For our purposes, we want to think of the controller as being engaged in a continuous updating task, or unending loop. In other words, we follow a procedure similar to the one outlined in Listing 2. We sketch the task for ATC; we provide somewhat more detail for Atmosphere. The code identifies `Atmosphere.Update_Temperature_Value` as a message, encapsulating the data defined as the 'real' number, TA.

```

ATC:
  loop
    Decide time for new temperature;
    Call Atmosphere to get TA now;
    Call Flux_Profile to get fluxes now;
    Compute new TA;
    Send new TA to Atmosphere;
    -- Rendezvous using function in the
    -- Atmosphere package identified as
    -- Atmosphere.Update_Temperature_Value(
    --   New_TA : in      real);
  end loop;
end ATC;

Atmosphere:
  TA : real;
begin
  loop
    accept Update_Temperature_Value(
      New_TA : in      real)
    do
      TA := New_TA;
    end Update_Temperature_Value;
  end loop;
  . . .
end Atmosphere;

```

**Listing 2. Pseudo-Code illustrating rendezvous between the Atmospheric Temperature Controller Package and the Atmosphere Package when both packages are viewed as (Ada) tasks.**

The procedures for the other controllers and objects are very similar. The task (or package) that needs to do something builds a function that accepts a message with encapsulated data structures. The initiating task is then responsible for sending the appropriate and valid message to the task that will carry out the work. Listing 3 provides a suggestion on how to design the FPC so that it interacts properly with the Flux object.

```

FPC:
  loop
    Get TS from Surface;
    Get Surface Optical Properties from
      Surface_Optical_Properties;
    Get TA from Atmosphere;
    Get Atmosphere Optical Properties from
      Atmospheric_Optical_Properties;
    Get Solar Constant from Sun;
    Call Flux_Profile to get fluxes now;
    Compute new Flux Profile;
    Send new Flux Profile to Flux_Profile;
    -- Rendezvous using function in the
    -- Flux_Profile package identified as
    -- Flux_Profile.Update_Flux_Profile
  end loop;
end ATC;

Flux_Profile:
begin
  loop
    accept Update_Flux_Profile
    do
      . . . ;
    end Update_Flux_Profile;
  end loop;
  . . .
end Flux_Profile;

```

**Listing 3. Pseudo-Code illustrating rendezvous between the Flux Controller Package and the Flux Package when both packages are viewed as (Ada) tasks.**

Note that this description of the FPC package does not contain any information about the model's time value. This means that this package will operate "blindly", continually looping to compute a new value of flux. As we have been intimating, this suggests we are still enmeshed in the notion that the Master Controller is in charge of time and will keep the whole model moving in lock-step with its desires. We still retain the hierarchical control structure that is so visible in Figure 7, where the Master Controller is in charge of ordering each of the subsidiary controllers to do particular things.

Listing 4 suggests (again in pseudo-code) what it looks like the Master Controller has to do when it is actually controlling the model computations. This loop is equivalent to the control loop in the FORTRAN-like version of the model. The tight ties to time stepping suggest that we may not have designed the subsidiary objects as autonomously as we should have.

```

Master_Controller:
  Loop
    Compute new time; -- New_Time := Time + dt;
    STC.Compute_New_TS(New_Time);
    ATC.Compute_New_TA(New_Time);
    FPC.Compute_New_Fluxes(New_Time);
  end loop;
end Master_Controller;

```

**Listing 4. Pseudo-Code illustrating how the Master\_Controller marches in lock-step through model time.** This pseudo-code suggests that the three packages (tasks) being called have appropriate rendezvous points that are part of their definition.

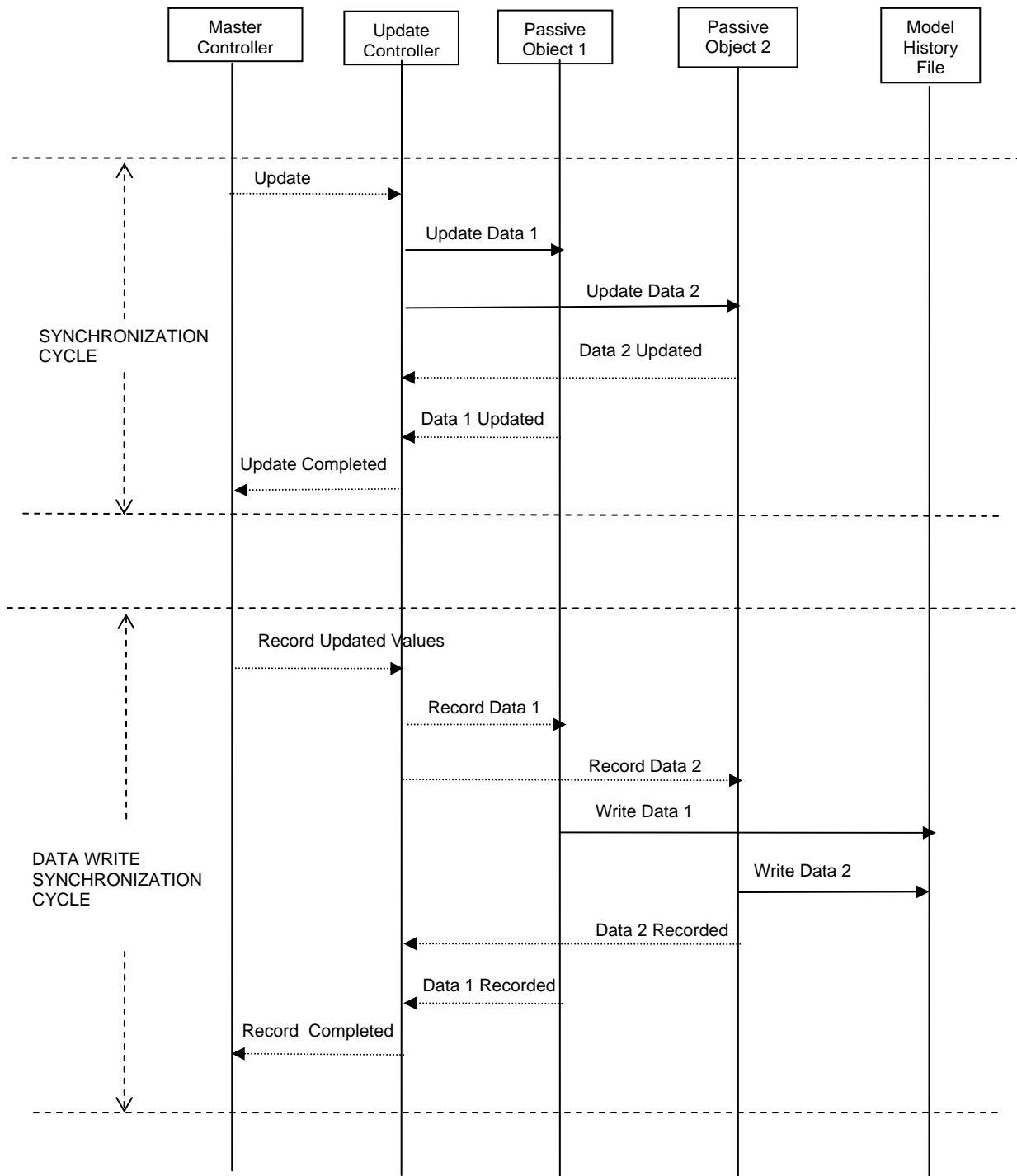
Of course, Listing 4 does not provide a complete specification of all the things the Master Controller must do. This package (or task) needs to get appropriate values for the current temperatures and fluxes to write to the Model History File. Also, we should note that just because the Master Controller requests computations from the subsidiary tasks that doesn't mean that they are completed by the time the Master Controller requests another update. If we are only running on a single CPU, perhaps lack of notification doesn't bother us. However, if we visualize this approach in a multi-site model, we may need a more careful control approach.

## Design Critique and Examination of Alternative Designs

The design we have derived is highly centralized. The subsidiary controllers have little freedom – simply perform their computations when directed to do so. However, as we have discovered, there may still be problems ensuring task completion unless these “lower-level” controllers receive notification that the appropriate messages have been received and that the other objects have finished their part of the work.

To provide a simple example of the way in which a process may need to receive appropriate information, consider a system that consists of a Master Controller, an Update Controller, two passive objects whose attributes the Update Controller intermittently adjusts, and a history file. Figure 12 shows sequence diagram where the Update Controller needs to ensure verification of message data changes from the passive objects before signaling that the process is completed.

In this alternative design, the “passive” objects that contain data assume responsibility for writing the data to the file and then notifying the Update Controller of their success. Likewise, the Update Controller sends relatively few messages regarding successful completion to the Master Controller. The design in Figure 12 does suffer from the need to synchronize the write operations. Because the object-oriented design requires us to specify whether we need a particular order to operations, the two writes indicated in this figure cannot be guaranteed to write in a particular order. In other words, we cannot guarantee that the file will be written so that Data 1 precedes Data 2. To provide such a guarantee, we need to introduce a “History Controller” or “Output Controller”.



**Figure 12. Sequence Diagram for a More Decentralized Control Approach.** In contrast with Figure 7, this design devolves responsibility for writing into the “passive” objects that contain the data (attributes) that need to be recorded. Also note that the Update Controller receives verification before signaling completion of a command.

In a general sense, we appear to be moving toward a more decentralized control approach than we started with. To the extent that we are dealing with a temporal hierarchy of objects, we can begin to view each object as having a sequence of states:

- Request and receive “constant” data from slowly varying objects in the environment
- Request and receive “summary” data from rapidly varying objects in the environment
- Compute new values for this object’s attributes
- Send new values to history controller
- Send new values for this object’s contribution to the environment of “faster” objects
- Send new values for this object’s contribution to the statistical state of “slower” objects

As a second design comment, it appears that a good distributed-control design will keep the data and its updates within the control of the object that contains them. The design we pursued so far appears to visualize data as a “passive” quantity that is acted upon by functions. In a better design, the data and the functions that work on it would be bundled together. This makes the design more cohesive and reduces the amount of message passing associated with the computations.

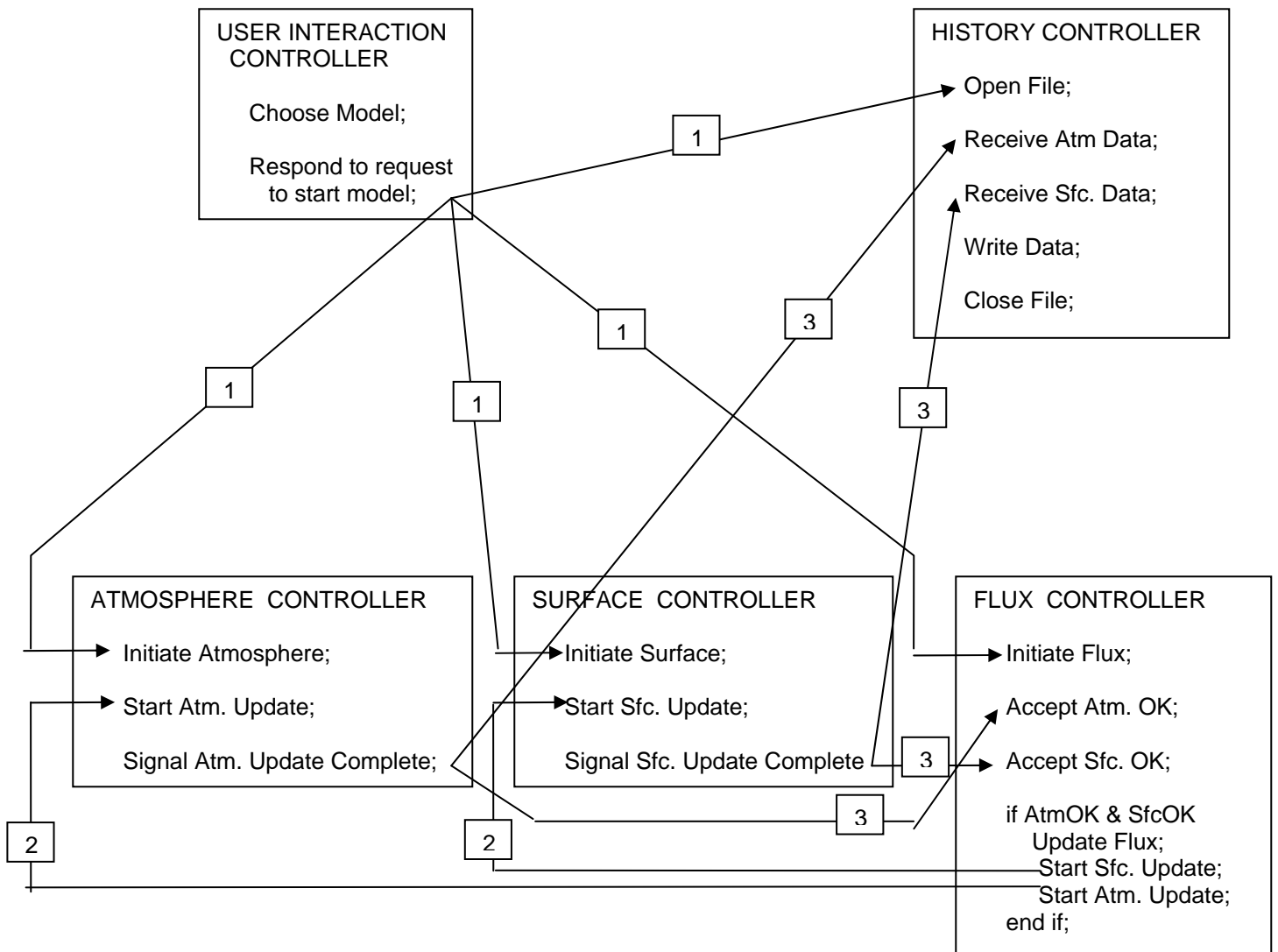
Figure 13 suggests an alternative design with about half as many objects as the previous design:

- *A User Interaction Controller*, intended primarily for sending messages to and receiving messages from the User Interface
- *An Atmosphere Controller*, dealing with the atmosphere’s temperature and optical properties
- *A Surface Controller*, dealing with the surface’s temperature and optical properties
- *A Flux Controller*, dealing with the radiative flux profiles
- *A History Controller*, dealing with writing data to the Model History File

For practical computation, we can instantiate these objects as Ada tasks. Figure 13 identifies the task entry points that serve as rendezvous locations for the messages through the system. The listings that follow this figure illustrate the way in which this concurrent approach encapsulates the object attributes with the objects. To a substantial degree, this encapsulation also avoids global data constructs that need to be visible to all of the objects in the system.

By following the messages labeled 1, 2, and 3, the reader can follow the basic looping sequence through the model computations. There is, of course, a substantial similarity to the loop through the standard paradigm model in the FORTRAN-like code. One way of looking at the new design is that it allows the loop to proceed even in the presence of indeterminacy about the time at which a particular computation will complete. The Flux Controller shows the kind of logic we need in order to accept a message to the effect that a new temperature is available, but to wait until all of the data needed for the computation is assembled before actually computing the new fluxes.

When we write the code to implement this design (in Ada, at least), we create one package for each of the objects in Figure 13. As in other object-oriented languages, this means that there is a package specification, which presents items that are publicly visible by other objects, and a package body, which is not visible and which encodes the procedures used in the computation.



**Figure 13. Alternative Design for the Simple Model.** This diagram shows the basic rendezvous points within the objects. The numbered lines indicate the sequence of message passing the model uses as it proceeds. Lines with the same number may be regarded as simultaneous. Note that the message sequence 2, then 3, then 2, and then 3 again provides the equivalent mechanism to the looping structure in the standard paradigm approach.

### Example Implementation of the Alternative Design

Listing 5 provides a package specification of the Atmosphere Controller object in Figure 13. Note that the atmospheric temperature is visible to other packages. Listing 6 provides the equivalent package body.

```

Package Atmosphere_Controller is
    TA : real := 200.0;  -- [K]

    task Control_Atmosphere is

        Initiate_Atmosphere(Initial_TA : in    real);

        Update_Atmosphere(
            F0_DNS : in    real;  -- Solar Flux at TOA
            F1_DNS : in    real;  -- Solar Flux at Sfc
            F1_UPS : in    real;
            F0_UPS : in    real;  -- Reflected Solar, TOA
            F1_DNL : in    real;  -- Emitted Flux at Sfc
            F1_UPL : in    real;
            F0_UPL : in    real;  -- Emitted Flux, TOA
            Time   : in    real);

    end Control_Atmosphere;

```

**Listing 5. Ada Code for the the Atmosphere Package Specification.** Note that this package makes the atmospheric temperature visible to other packages. It also provides a task with two entry points that serve as message receivers.

Listing 6 is interesting because it shows how the new design combines message passing entry points with computation. In Ada, the task is started as soon as the package is elaborated. This means that the loop is effectively ready to receive any of the message rendezvous points defined in the code. The loop contains control structures that distinguish between the uninitialized state of this task and its state after initialization.

All of the objects are designed and implemented in the same fashion. The equations we derived early in this paper appear in the appropriate task bodies. The same is true of the parameters that contain the computed values from the model runs.

```

package body Atmosphere_Controller is

    task body Control_Atmosphere is
        Initiated : Boolean := false;
        Time : real := 0.0;
    begin
        loop
            select
                when not Initiated =>
                    accept Initiate_Atmosphere(
                        Initial_TA);
                    TA := Initial_TA;
                    Initiated := true;

                Or

                When Initiated =>
                    Accept Update_Atmosphere(
                        F0_DNS,
                        F1_DNS,
                        F1_UPS,
                        F0_UPS,
                        F1_DNL,
                        F1_UPL,
                        F0_UPL,
                        New_Time);
                    TA := TA + ((F0_DNS - F0_UPS)
                        + (0.0 - F0_UPL)
                        - (F1_DNS - F1_UPS)
                        - (F1_DNL - F1_UPL))
                        *(New_Time - Time)/CA;
                    Time := New_Time;
                    Flux_Controller.Control_Flux(TA);
            end select;
        end loop;
    end Control_Atmosphere;

```

**Listing 6. Ada Code for the the Atmosphere Package Specification.** Note that this package makes the atmospheric temperature visible to other packages. It also provides a task with two entry points that serve as message receivers. [This code derived from memory – will be updated in revision.]

## Concluding Comments on Object-Oriented Models

It should be clear that work on object-oriented models for Earth science applications has barely begun. In many ways, modeling work proceeds as it has for the last several centuries. Formulate a description of the physics that we believe describe a phenomenon we want to model. Then, reduce the formulation to a computable form. Finally, run the model and compare with available observations.

As we have seen, designing models using an object-oriented approach frees us to consider new ways of describing physical phenomena – perhaps in a way that is more friendly to human psychology. It is easier to think of object life histories and to describe these histories to others than it is to describe an ever swirling continuum. The freedom to adopt concurrent computations in a systematic fashion is also likely to prove a liberating force for this approach to designing computations.

The freedom of the object-oriented approach comes at a price. It is clear that we know how to formulate and solve efficiently the fluid-flow problems that are essential for much of our realistic work in the Earth sciences. It is not so clear how to adopt the old mathematics to the new framework. While the author suspects that the most sensible approach is likely to evolve by modifying current quasi-Lagrangian flow schemes, in which the fluid parcels are allowed to move and change shape, and are then coaxed back into an acceptable grid structure to start the next step. What would be needed to extend this approach to an object-oriented framework is to identify the parcels that belong to the objects and then put them into the objects. In this case, the fluid flow calculations would become the source of messages between objects, in which the contents of the messages are the data on the fluid properties.

The approach we have described in this paper does not include methods of estimating quasi-optimal object partitionings. In the past, it has been common to use a fixed grid point partitioning, in which a number of grid points are selected in advance. In the kind of object-oriented model we describe in this paper, the number of points (as well as their geometry) become dynamic – the objects could change shape and size over the course of the computation. This means that we need reasonable algorithms for making reasonably efficient choices for partitioning the problem's geometry, including both message passing cost and computation cost as part of the choice. The sequence diagrams we have used provide a useful first step in enumerating the data flow associated with message passing. However, this first step is far from providing a reliable method for engineering dynamic partitioning of a model's problem space.

The problem of dealing with concurrency is similar. The object-oriented approach offers modelers the freedom to use multi-time scale models. Indeed, we have suggested a philosophy of using a time hierarchy, in which big, slow objects seem constant with respect to smaller, faster objects. The big, slow objects have so much inertia that they can only respond to statistical averages of the smaller, faster ones.

This philosophy might even be extended to allow smaller, faster objects to be treated in a statistical fashion. In this philosophy would assume that the objects are drawn from a spectrum

of different sized objects and only do the computations for selected points in the spectrum. When the computations were completed, the model would interpolate between the properties of the objects that had been selected as representatives.