Table of Contents

Intended Audience

Overview of the OpenSim API for Developers

Example

OpenSim API Architecture Overview

System and State

Component

Models are composed of Components

Property

Connector*

Inputs and Outputs*

Operator*

Source*

Reporter*

Frames*

Point, Station, and Marker*

Data Handling Classes*

Data Table

Data Adapter

Solvers*

Writing your own Component

* New Functionality in OpenSim 4.0

Intended Audience

This guide is recommended for those who desire in-depth knowledge of musculoskeletal modeling and simulation with OpenSim and wish to use the Application Programming Interface (API) to create or contribute novel models or algorithms not currently supported by the OpenSim application. The guide also expects that you are familiar with multibody dynamics and biomechanics.

The OpenSim software has a variety of interfaces: GUI, command line, XML, Python scripting, MATLAB scripting, and C++. This guide is designed for developers using the last 3 of these interfaces. Programming experience in C++ will be helpful in understanding the API architecture, but we have tried to make this guide accessible to those with only Matlab or Python experience, as well.
Overview of the OpenSim API for Developers

**OpenSim’s Purpose:**
The purpose of OpenSim is to provide anyone interested in human or animal movement with high-performance software to create computer models of the neuromusculoskeletal system, analyze experimental movement data, and simulate movement dynamics to gain insights about the anatomy, physiology, mechanics and control of movement. The software is used in a variety of applications, including neuroscience research, biomechanical analysis, surgical design, and ergonomic analysis, and bioengineering education, just to name a few. Our goal is to provide physically accurate models and tools that provide a solid foundation for scientific inquiry.

The OpenSim Application Programmer’s Interface (OpenSim API) serves both the OpenSim application, with its graphical user interface (GUI), and enables advanced users and developers to write their own programs to extend the capabilities of OpenSim, including customized analyses and workflows, Components, and OpenSim plug-ins.

**Design Philosophy:**
The ethos of the OpenSim API is that the code is modular, reusable, and easily extensible. To enable the assembly of neuromusculoskeletal systems from constituent biomechanical elements, OpenSim has adopted the composite design pattern¹ to systematically define, organize, and manage the elements (components) of a neuromusculoskeletal system.

**Composite Model Framework for Building a Computational System**
In OpenSim, a Component is a computational model element that describes some physical phenomenon. As a neuromusculoskeletal simulator, OpenSim components represent bodies, joints, muscles, and other physical structures, as well as sensors, controllers, and feedback circuits that describe human and animal dynamics. The purpose of a Component is to both capture the physical phenomenon (degrees-of-freedom, constraints, actuation, control) of interest and to enable the systematic composition of complex behaviors (dynamical models) from simpler components.

To illustrate this point, think of two biomechanists, Gary Gait and Norma Knee. Gary is interested in the actions of individual muscles during gait and other forms of locomotion. Norma wants to understand how loads are distributed within the knee and their effect on cartilage wear and health. For Gary, a model represents a patient’s musculoskeletal dynamics and is capable of muscle-driven gait that reproduces human-like performance. In Norma’s case, a model consists of a femur, a tibia and a patella, contact of articulating surfaces, ligaments, and muscles and she can compute contact forces. Gary and Norma’s “models” may have similar complexity but their concepts of a model are fundamentally different (i.e., whole body vs. a

¹ en.wikipedia.org/wiki/Composite_pattern
knee). Gary may model knees as pin joints and attain reasonable answers about the action of leg muscles during different types of gait. Norma cannot justify the same simplification. A more accurate knee model may result in more realistic muscle forces for Gary, and simulated muscle forces (that reproduce human gait) may provide better boundary conditions for Norma’s estimates of knee loads. However, for these researchers to benefit directly from each other’s work, their models must be interoperable. In order for their models to talk to one another and be systematically combined, both models must have constructs(s) in common. This modularity and interoperability is the main purpose of the Component API.

Figure 1. Overview of the OpenSim Modeling Framework. Aspects of the musculoskeletal system (Physical System) such as bones, joints, and muscles, and its neurocontrol composed of spinal circuits, muscle spindle and Golgi tendon organs are represented as Components in the Modeling Framework. The specification of the types and arrangement of Components in a Model reflect the physical system being modeled and it is the main task of the Modeler. From the assembly of Components (the Model) the corresponding computational System (system of equations readily solved by a computer) is automatically created and ready for numerical simulation and analysis. All System unknowns (variables) and their values are in the State.
Example

What exactly does the OpenSim API look like? Here is an example of C++ code for building a model of an arm with a single muscle and running a simulation. In the remainder of the guide, we will provide a detailed description of the key classes and methods of the OpenSim API architecture that enable this example.

Simple arm and actuator

```cpp
#include <OpenSim/OpenSim.h>
using namespace SimTK;
using namespace OpenSim;
int main() {
    Model model;
    model.setUseVisualizer(true);

    // Create two links, each with a mass of 1 kg, center of mass at the body's
    // origin, and moments and products of inertia of zero.
    OpenSim::Body* humerus = new OpenSim::Body("humerus", 1, Vec3(0), Inertia(0));
    OpenSim::Body* radius = new OpenSim::Body("radius", 1, Vec3(0), Inertia(0));

    // Connect the bodies with pin joints. Assume each body is 1 m long.
    PinJoint* shoulder = new PinJoint("shoulder",
        // Parent body, location in parent, orientation in parent.
        model.getGround(), Vec3(0), Vec3(0),
        // Child body, location in child, orientation in child.
        *humerus, Vec3(0, 1, 0), Vec3(0));
    PinJoint* elbow = new PinJoint("elbow",
        *humerus, Vec3(0), Vec3(0), *radius, Vec3(0, 1, 0), Vec3(0));

    // Add a muscle that flexes the elbow.
    Millard2012EquilibriumMuscle* biceps = new
        Millard2012EquilibriumMuscle("biceps", 200, 0.6, 0.55, 0);
    biceps->addNewPathPoint("origin", *humerus, Vec3(0, 0.8, 0));
    biceps->addNewPathPoint("insertion", *radius, Vec3(0, 0.7, 0));

    // Add a controller that specifies the excitation of the muscle.
    PrescribedController* brain = new PrescribedController();
    brain->addActuator(*biceps);
    // Muscle excitation is 0.3 for the first 0.5 seconds, then increases to 1.
    brain->prescribeControlForActuator("biceps",
        new StepFunction(0.5, 3, 0.3, 1));
```
// Add components to the model.
model.addBody(humerus);  model.addBody(radius);
model.addJoint(shoulder); model.addJoint(elbow);
model.addForce(biceps);
model.addController(brain);

// Add a table reporter to store the muscle fiber force over time.
TableReporter* tblReporter = new TableReporter();
tblReporter->set_report_time_interval(1.0);
tblReporter->upInput("inputs").connect(biceps->getOutput("fiber_force"));
model.addComponent(tblReporter);

// Configure the model.
State& state = model.initSystem();
// Fix the hip at its default angle and begin with the knee flexed.
model.updCoordinateSet()[0].setLocked(state, true);
model.updCoordinateSet()[1].setValue(state, 0.5 * Pi);
model.equilibratemuscles(state);

// Add display geometry.
model.updMatterSubsystem().setShowDefaultGeometry(true);
Visualizer& viz = model.updVisualizer().updSimbodyVisualizer();
viz.setBackgroundColor(White);
// Ellipsoids: 0.5 m radius along y-axis, centered 0.5 m up along y-axis.
DecorativeEllipsoid geom(Vec3(0.1, 0.5, 0.1)); Vec3 center(0, 0.5, 0);
viz.addDecoration(humerus->getMobilizedBodyIndex(), Transform(center), geom);
viz.addDecoration(radius->getMobilizedBodyIndex(), Transform(center), geom);

// Simulate.
RungeKuttaMersonIntegrator integrator(model.getSystem());
Manager manager(model, integrator);
manager.setInitialTime(0); manager.setFinalTime(10.0);
manager.integrate(state);

// Write the muscle fiber force table to file.
auto table = tblReporter->getReport();
CSVFileAdapter::write(table, "bicepsFiberForce.csv");
}
OpenSim API Architecture Overview

The following sections provide an overview of the key abstractions and definitions used in the OpenSim API. OpenSim 4.0 includes lots of new functionality, which we have indicated with an asterisk when first defined.

System and State

A System is the computational (mathematical) system of equations that represents the model dynamics, in a way that a computer can solve. In OpenSim, the model is formed by rigid bodies (e.g. bones) and connected by joints. A State is a set of values for all the unknowns (variables) of the System’s equations. The State contains all values necessary to fully evaluate the system of equations. The State includes time and the generalized coordinates ($Q$; the joint angles and displacements) and the generalized speeds ($U$) of the multibody system. Any Component can add to the System (and State). For example, Muscles add their activation and fiber-length variables to the State with their corresponding differential equations in the System.

A StatesTrajectory$^*$ is a sequence of States that satisfy the dynamical system of equations through time.

```cpp
SimTK::State state;
int nq = state.getNQ(); // Number of generalized coordinates.
SimTK::Vector q = state.getQ(); // Values of generalized // coordinates
SimTK::Vector qdot = state.getQDot(); // Derivatives of the // generalized coordinates.
```

Any method in OpenSim that performs a calculation that doesn’t simply depend on constants takes a SimTK::State object as the first argument.

```cpp
model.getMassCenter(state); // Need to know pose of model.
```
A Component is the basic “unit” of modelling. For example, a muscle can be modeled as a single Component. A Component has the following user-facing attributes:

1. **Properties**: Constant parameters (e.g., a Body’s mass).
2. **Connectors** to other components that it depends on (e.g., a Joint connects two Frames).
3. **Input** quantities that the component needs to do its job (e.g., a metabolics calculator component could have a “muscle power” input).
4. **Output** quantities that component can compute, and that can be used to satisfy other components’ **Inputs**.
5. **Subcomponents**: A Component can rely on other smaller components to do its job (e.g., a Muscle component could have subcomponents for activation dynamics and for fiber dynamics).

A Component has the following computational attributes:

1. Continuous state variables (usually referred to as “state variables”) are the variables in the dynamical equations contributed by the Component (e.g. activation and fiber-length state variables of a Muscle).
2. Discrete state variables are system unknowns that may or may not be governed by differential or algebraic equations, and include external inputs and controls. For example, the control to an Actuator is a discrete state variable.
3. Cache variables allow saving (state-dependent) calculations that may be used multiple times, for computational efficiency. For example, computing the path length of a muscle with several intermediate points and wrapping over obstacles is expensive. Therefore, a path component itself caches its length so that the muscle (or any other component) that needs the length in its computations does not reevaluate it unnecessarily. The validity of the path length cache variable is dependent on the generalized coordinates--as long as these generalized coordinates do not change, the path length is valid. OpenSim employs categories of dependency known as realization Stages (see Simbody User Guide, Section 2.4 for details) to manage the validity of cache variables. Any changes to the generalized coordinates invalidate
(and wipe out) any cache variable associated with the Position Stage and all other subsequent (Velocity, Dynamics, Acceleration) stages. OpenSim automatically invalidates cache variables to support the correctness of simulations. Manual cache management is highly prone to errors leading to “stale” variables and fundamentally incorrect physics. Sadly, while results may appear plausible they are completely invalid and sometimes impossible to detect and debug. If you need to cache for performance, use OpenSim’s facilities.

Models are composed of Components

A Component encapsulates (all or part of the) system dynamics (of a model or element of a model) and provides (computes) values of interest. A component adds its dynamics and allocates necessary resources (state, cache, and other variables). It defines parameters (see Properties below) that specify its behavior/function and defines dependencies on any other Components (see Connectors below) via its Inputs and Outputs.

Components of a Model form a rooted directed tree topology of ownership (Components own and know about their subcomponents not the other way around). A root component (the top level Model) contains all the necessary Components to define the System. Components in the tree are uniquely identified by their full path name from the root (like a file path)

A submodel (e.g., a subassembly in CAD) forms its own branch of a Model. This might be a Device, a Leg model or even a FullBody model.

This Component architecture means that:
- A Component can be composed of Components
- A Model is a Component.
- A Model is composed of Components
- A Model can be composed of Models*

You can add a subcomponent to another Component like so:

```java
CoordinateActuator* act = new CoordinateActuator();
act->setName("motor");
device.addComponent(act);
```

You can access existing subcomponents by name:
You can also iterate through all subcomponents of a component, recursively (e.g., each subcomponent’s subcomponents):

```cpp
for (const auto& c : device.getComponentList()) {
    c.getFullPathName();
}
```

```cpp
// Only iterate through Bodies
double totalMass = 0;
for (const Body& b : device.getComponentList<Body>()) {
    totalMass = totalMass + b.getMass();
}
```

### Property

A Property is a fixed parameter of the model that does not change as a simulation evolves over time (i.e., is not part of or dependent on the State). A Property is a serializable (as XML) container of common data types (e.g., doubles and strings) as well as Objects (e.g., Functions) and even other Components which include properties themselves. A serialized model is the XML counterpart of the Model and its Components (as Properties).

For each Property, there are a number of methods for accessing or modifying the value of the Property. For example, a Body’s mass property has the following related methods:

```cpp
double m = body.get_mass();
body.set_mass(3.5);
```

The properties available in any class are listed on Doxygen, the GUI (Help > XML Browser), and the command line tool (opensim info Model).
A Connector defines the dependency of a Component on another Component (the “connectee”). The type of the connectee must be specified type (e.g., Body, Joint). The dependency is specified by the connectee’s relative or full path name. A Connector automatically finds and connects to the dependency when connect() is called. The Connector provides the status of the connection (connected or not) and a reference to to the “connectee” when connected. All of a class’ Connectors are listed on the Doxygen page for that class.

You can access a Component for the number of Connectors it has, as well as for their names:

```cpp
int numConn = joint.getNumConnectors();
auto connNames = joint.getConnectorNames();
```

You can get a specific Connector of a Component using its name:

```cpp
AbstractConnector& parent = joint.updConnector("parent_frame");
```

Once you have a Connector, you can see what types of Components it can be connected to, and if its connection is satisfied:

```cpp
parent.getConnecteeTypeName(); // result: "PhysicalOffsetFrame"
bool satisfied = parent.isConnected();
```

You can specify which Component the Connector should connect to. Here, we tell a Joint that its parent frame is Ground:

```cpp
parent.connect(model.getGround());
```

To access the connectee itself, you can call the following method on the Component:

```cpp
auto ground = joint.getConnectee<PhysicalFrame>("parent_frame");
```
Inputs and Outputs

Inputs and Outputs specify data flow. An Output is any Component “computed” value as a function of the state. An Output is used to pass results of computation from one Component to any other, provide fast access to Component calculations (same as a member function call), and collect results without needing to know the details of the source Component and/or its methods.

An Input is a “slot” that accepts an Output of a specified type. The Input verifies that the Output can be evaluated/consumed as an Input. Inputs allow for a Component to have a dependency on an Output (that can come from any Component or user-specified Data) rather than a specific Component type.

An Output is a value of interest generated by a Component (e.g., Muscle has Outputs for its fiber-length and its tension). Outputs are accessed by name and evaluated given a State. Outputs are typed by their value type (e.g., double, Vec3, SpatialVec, Vector).

The Input provides its status as being connected (to an Output) or not. Thus if an Input is connected to an Output, the Input triggers the evaluation of the Output by the Component to which the Output belongs. If an Output is not needed by any Input it is never evaluated.

(1) Single Value Output: A single channel of data. Can be any data type (Vec3, Mat33, double, etc)
(2) List Output: Multiple channels of data. Each channel must be of the same data type
(3) Single Value Input: An input channel that reads one channel of data.
(4) List Input: A list input that reads multiple channels of data of the same type

All of a class’s Inputs and Outputs are listed on the Doxygen page for the class.

You can access the number of inputs a Component has, as well as their names:

```cpp
int numIn = metabolics.getNumInputs();
```
auto names = metabolics.getInputNames();

Once you have their names, you can access a specific Input:

AbstractInput& input = metabolics.updInput("activation");

You can ask a specific Input for its data type and whether or not it is connected:

string type = input.getConnecteeTypeName(); // result: double
bool satisfied = input.isConnected();

You can connect an Input to an Output like so:

input.connect(soleus.getOutput("activation"));

If an Input is connected to an Output, you can ask for its value:

double a = metabolics.getInputValue<double>("activation");

There are similar methods for Outputs:

int numOut = metabolics.getNumOutputs();
auto names = metabolics.getOutputNames();
AbstractOutput& output = metabolics.getOutput("heat_rate");
string type = output.getTypeName();
string value = output.getValueAsString();
double hr = metabolics.getOutputValue<double>("heat_rate");

List Outputs may have multiple Channels:

bool isList = output.isListOutput();

If so, then you could connect individual Channels to an Input:

input.connect(tablesource.getOutput("column").getChannel("col1"));

If you have a List Input, then you can wire it to multiple Channels (from both one-value and list Outputs):
There are cases where an Input might need to "rename" an Output. For example, an InverseKinematics Solver could have a List Input for experimental marker locations, and could require its connectees to be named after markers in the Model. But say the experimental marker that corresponds to the Model’s “toe” marker is called “foot.” One can “rename” an output using “annotations”:

```java
input.connect(tablesource.getOutput("column").getChannel("col1"));
input.connect(soleus.getOutput("activation"));
```

---

**Operator***

An Operator is a purely functional/mathematical Component. An Operator defines a set of one or more Inputs, computes on them, and send the results to a set of one or more Outputs. An Operator has no dependency on other Components and does not require a Model to operate (just Inputs). Some examples are arithmetic operations (+,-,*,/), delays, and de/multiplexers.

---

**Source***

A Source is a category of components that serves as a source of Outputs (signals) to satisfy model Inputs and that does not itself have any Inputs.

TableSource is one such Component which holds a TimeSeriesTable and exposes a List Output which contains a Channel for each column of the table.
A Reporter reports the results of Model computations. It can take any Outputs from a Model (and its Components) as its Inputs. Whether it reports to the terminal, file, or a port is dependent on the concrete Reporter type. Reporters are templated on the datatype of the reported Output<T> type (i.e., a single Reporter can only output one type of data), but there is no limit to the number of Reporters a Model can have.

TableReporter is one such Component which can populate a table with values it receives from its List Input. Every channel from the List Input results in a column in the table.

ConsoleReporter is another such Component which receives values from its List Input and prints them to the console/terminal.

Frame is an OpenSim representation of a reference frame. It consists of a right-handed set of three orthogonal axes and an origin point. Frames provide a convenient way to locate physical structures, such as joints and muscle attachments. Frames also provide a convenient basis for performing spatial calculations. For example, if your system includes contact, you might define a Frame that is aligned with the normal direction of a contact surface and whose origin is at the center-of-pressure. You can then easily report the contact forces normal to the contact surface and the location of the center-of-pressure, without repeatedly performing the transformations in your main or analysis code.

There are several types of Frames:

(1) **PhysicalFrame**: supports physical connections (e.g., Joints, Constraints) and is the Frame type to which forces can be applied. A concrete example of a PhysicalFrame is a Body. PhysicalFrame is an abstract class.
(2) **PhysicalOffsetFrame**: a type of Physical Frame whose transform is specified as a constant offset from another Physical Frame. For example, PhysicalOffsetFrames can be used to specify the location of a Joint or Constraint on a Body.

(3) **Body**: a PhysicalFrame with inertia. A Body is specified by its mass, a center-of-mass located in the PhysicalFrame, and its moment of inertia tensor about the center-of-mass.

(4) **Ground**: an inertial reference frame in which the motion of all Frames and Points may conveniently and efficiently be expressed. As a PhysicalFrame, Ground supports physical connections by joints, constraints and forces can be applied to it.

The following diagram illustrates how each type of PhysicalFrame might appear in a model.

![Diagram of PhysicalFrames](image)

Every Frame is capable of providing its Transform (translation of the origin and the orientation of its axes) in the Ground frame as a function of the State.

The Frame class provides convenience methods for re-expressing vectors from one Frame to another.

It is perhaps less evident that Frames can be extremely useful for linking a multitude of reference frames together to form chains and trees. For example, a Frame to specify muscle attachments (M) and a Frame to specify a joint location (J) could themselves be specified in an anatomical Frame (A) defined by bony landmarks identified by surface markers or tagged on CT or MRI images. The body (B), to which the anatomical frame (A) is attached, can be thought of as a "Base" frame or a root of a tree from which a set of descendant frames arise. In particular, a Base frame and all its descendants have
the property that they share the same angular velocity, since they are affixed to the same underlying Frame (in this case a Body).

\[
\begin{align*}
M \text{---muscle points} \\
/ \\
B \text{---A} \\
\backslash \\
J \text{---joint axes}
\end{align*}
\]

Given this tree, both the muscle attachment points (in M) and the joint axes, J, change when the anatomical frame, A, changes with respect to the base, B, without requiring muscle attachments and joint axes to be manually adjusted. Consequently, a useful concept is that of a Base frame, and a Frame can always provide a Base frame. If a Frame is not affixed to another frame, its Base frame is itself.

Point, Station, and Marker*

A Point is an OpenSim representation of any location in space. Points can be used to define and compute the location of physical structures (such as points of constraints and points of muscle attachments). Points can also embody the results of spatial calculations. For example, if your system involves contact, you can define a Point that describes the location of the center-of-pressure as one element rolls over another.

A Point provides its location, velocity, and acceleration in the Ground frame as a function of the Model’s state, as long as the state has been realized to the appropriate stage (Position, Velocity, or Acceleration)

(1) **Point**: an abstraction for any location in space
(2) **Station**: a Point fixed to and located on a Physical Frame, which can be a Body, Ground, or any Physical Offset Frame.
(3) **Marker**: implementation of a Mocap Marker.

OpenSim Stations are Points defined in Euclidian space. They are defined with a three element column vector (Vec3), relative to their parent Frame. Stations are thus analogous to PhysicalOffsetFrames in that constraints and forces can be attached and/or applied to them.
Data Handling Classes*

Data Table

DataTable is an in-memory storage container for data with support for holding metadata. DataTables provide a single, unified method for storing and accessing data, such as results generated by Components. They provide fast access to numerical elements and to complete rows and columns of data, by index and by column names. By providing in memory storage for Components (e.g., Reporters and Sources described above), DataTable separates the data itself from file formats and isolates file handling to Adapters (see below).

DataTables contain an independent column and a set of dependent columns. The independent column can be any scalar data-type (e.g., int, float, double). The dependent columns can be of any data type (double, Vec3, Mat33, SpatialVec, etc.). All of the dependent columns must be of the same data type. Each independent and dependent column, as well as the entire table, can contain metadata.

Data Adapter

DataAdapter is an abstract class that defines an interface for reading and writing the contents of a DataTable.

The DataAdapter separates use of a DataTable from the various sources of data, including streams, files, databases, and devices. A DataAdapter is how you read from or write to a DataTable to/from different sources of data. Concrete classes handle the particular interface and format of a given data source.

FileAdapter offers an interface to read and write files through the following methods.

```cpp
FileAdapter::readFile(string fileName)
FileAdapter::writeFile(string fileName)
```
Based on the extension in filename, these methods invoke one of the following concrete FileAdapters to perform the read/write. It is also possible to invoke the following concrete FileAdapters directly.

(1) **TRCFileAdapter**: reads and writes TRC files.
(2) **STOFileAdapter**: reads and writes STO files.
(3) **CSVFileAdapter**: reads and writes CSV files.
(4) **C3DFileAdapter**: reads C3D files.

---

**Solvers**

A Solver is an algorithm to compute System unknowns of a Model. For example, the InverseKinematicsSolver operates on the System underlying a Model to determine the generalized coordinate (state variable) values that satisfy external measurements (markers). The MomentArmSolver evaluates a muscle path to determine the effectiveness of a muscle to generate a generalized force about or along a coordinate.

The Manager currently serves as a “ForwardDynamics” Solver which integrates the System dynamics forwards in time. We intend to provide a ForwardDynamicsSolver that returns a StatesTrajectory. In the meantime, you can generate a StatesTrajectory by implementing your own simulation function, like this:

```cpp
StatesTrajectory simulate(const Model& model, const State& initialState, double finalTime) {
    StatesTrajectory states;
    SimTK::RungeKuttaMersonIntegrator integrator(model.getSystem());
    SimTK::TimeStepper ts(model.getSystem(), integrator);
    ts.initialize(initialState);
    ts.setReportAllSignificantStates(true);
    integrator.setReturnEveryInternalStep(true);
    while (ts.getState().getTime() < finalTime) {
        ts.stepTo(finalTime);
        // StatesTrajectory API for appending states:
        states.append(ts.getState());
    }
    return states;
}
```
Writing your own Component

The main task for Components (as part of a Model) is to generate a System. When the underlying System is built all the variables in the system equations appear in the State. A Component, therefore, is responsible for adding state variables to the system and providing access to those variables. The system can (and typically does) obtain default values for state variables from the Component's properties.

The responsibility of the Component builder is four fold:

I. **construct** the **structural attributes** of the component: its properties (serializable attributes), connectors (structural dependencies), inputs (signals it needs) and outputs (signals/values, results) that it will produce.

II. **finalize** data members that are dependent on your properties and add relevant checks to ensure your component is **connected** properly. With a Component finalized and connected, it is time to translate your Component into equations of the System and its variables into the corresponding State. Once a system is built, changes to any Component properties and connections will invalidate the System and the System must be rebuilt.

III. **add** the necessary multibody elements to the system, and any other dynamics described as a system of first order ODEs. This involves allocating and **adding** state variables and specifying their time derivatives.

IV. **initialize** the state variables allocated by your component, which generally involves using properties either to hold initial values or to compute them. The inverse operation of **converting a state into properties** for use another time is also helpful.

I. **Constructing your Component's Attributes**

1. Derive your component from the base component type you want to extend. You may derive from that class or its parent. If your component is completely new, you can derive directly from Component or ModelComponent, but that is generally a bit more work, since the base classes for the different types do a lot of the hookups (that follow) for you.

2. Construct your component with the properties (attributes that appear in XML) you want, you must first define the properties you want to expose by calling one of the "OpenSim_DECLARE_PROPERTY" macros, which you use to specify the name, type and description of the property. NOTE: your class automatically has get/set_<property_name>() methods when you apply the macro and that that properties and corresponding methods are inherited from its parent class.

```cpp
OpenSim_DECLARE_PROPERTY(gain, double, "Controller gain.");
OpenSim_DECLARE_LIST_PROPERTY(functions, Function,
    "Functions are used to compute the force applied to bodies.");
```
3. Invoke the individual `constructProperty_<property_name>()` methods for each property you have declared and assign them default values in each constructor. If you have multiple properties and constructors it is convenient to implement `void constructProperties()` as a private method that can be called from each constructor.

```cpp
public:
    MyController() { constructProperties(); }
private:
    void constructProperties() {
        constructProperty_gain(10);
        constructProperty_functions();
    }
```

4. If you have dependencies on other components you should specify those dependencies by using the “OpenSim_DECLARE_CONNECTOR” macro with the Connector name and type of the dependency. For example, a Joint has two connectors of type PhysicalFrame and their names are “parent_frame” and “child_frame” respectively. Connectors automatically generate properties with connector type and name as part of the XML tag. The expected XML value is the full path name of the dependency component. OpenSim can now check the existence of the component that is your dependency and of the required type and provide meaningful messages.

```
OpenSim_DECLARE_CONNECTOR(muscle, Muscle,
    "The muscle for which to compute metabolic rate.");
```

5. Specify your component’s Outputs by employing the “OpenSim_DECLARE_OUTPUT” macro with the name of the output, its value type, and the member function of your component that produces the value of the output. Similarly, use the “OpenSim_DECLARE_INPUT” macro to create a named slot for the Output of another component to plug in.

```
OpenSim_DECLARE_INPUT(desired_angle, double, SimTK::Stage::Position,
    "This controller will try to minimize the error from this angle (radians.).");
OpenSim_DECLARE_OUTPUT(heat_rate, double, SimTK::Stage::Dynamics,
    calcHeatRate);
```

II. Finalize and Connect the Component

1. Override `extendFinalizeFromProperties()` to perform any data loading or conversions that the Component will need during a simulation. For example, you have six numbers from properties that are the moments and products of inertia, then within `extendFinalizeFromProperties()` you can convert those 6 numbers to a proper `SimTK::Inertia`, which will throw pertinent exceptions about the validity of those numbers to form a physical inertia tensor. NOTE, the base Component marks the component as being up-to-date with its properties at the end of `finalizeFromProperties()`.
If be the that them indices = SimTK::State & component. However, you can implemented the System so void extendAddToSystem (SimTK::MultibodySystem & system) const override {
  // Implementation of component dynamics
}

III. Adding your Component’s Dynamics to the System

1. Override void extendAddToSystem(SimTK::MultibodySystem& system) const and add Simbody elements to the System. It is important to note that you must maintain the underlying indices so your component can access them later if necessary. Joint, Constraint, Force, provide methods to help you create new types of those Components and manage the indices for you. Ask for help if you want to expose state variables allocated by underlying Simbody elements.

2. If you have component dynamics that you will model as ODEs you must also add the corresponding state variable(s) to the system’s state also in extendAddToSystem() by invoking addStateVariable(). If you can express your component dynamics as a function of the state: zdot = F(state), then z is the state variable added by your component.

3. The function (e.g. F(state) above) that determines the time derivative of your state must be implemented in void computeStateVariableDerivatives(const SimTK::State& s) const, which you override and use setStateVariableDerivative() to set the derivative value for a specified state variable.

4. NOTE: Any value that your dynamics depends on that isn’t a constant, must itself be a state variable or an Input. If the values are provided externally (e.g. user supplied) the values must be held as discrete state variables and they are similarly allocated by addDiscreteVariable() and must be set before the dynamics of the Component are realized.

void extendAddToSystem(SimTK::MultibodySystem& sys) const override {
  // Implementation of component dynamics
}
IV. Initializing and Recalling the State

1. It is good practice to initialize your component’s state variables, so that the component and the Model as a whole can simulate without user input. For example, Coordinates maintain “default_value” and “default_speed” properties for this purpose. The translation from properties to state variable values is implemented by

```cpp
void extendInitStateFromProperties(SimTK::State& state) const;
```

which you can override to initialize all the state variables your component has added from properties you have defined.

```cpp
void extendInitStateFromProperties(SimTK::State& s) const override {
    Super::extendInitStateFromProperties(s);
    setStateVariableValue(s, "activation", get_default_activation());
}
```

2. Similarly, overriding

```cpp
void extendSetPropertiesFromState(const SimTK::State& state);
```

enables you to perform the inverse operation and to update the property (for example the default_value) from the value in the State, for instance after a forward simulation. Ideally, your component should enable a simulation to be stopped, the state values held as properties, the model saved, then reloaded, initialized from its properties and the simulation resumed, such that concatenation of the two simulations is identical to performing a single simulation of longer duration.

```cpp
void extendSetPropertiesFromState(const SimTK::State& s) const override {
    Super::extendSetPropertiesFromState(s);
    set_default_activation(getStateVariableValue(s, "activation"));
}
```

While testing the dynamics of your Component and its results is best left to you, the Component builder, OpenSim provides a test harness for testing the structure and integrity of your Component. The test program in `testComponents.cpp` instantiates and performs some standard checks on each Component in its queue, to which you can include your Component, to test:

1. serialization and deserialization and their equivalence
2. cloning (copying) with its equivalence to the original
3. memory increases due to copying
4. adding the component to a Model
5. the use of Connectors to define and satisfy dependencies
6. initializing the System (calling model.initSystem()) and its impact on memory
7. Evaluating Outputs at the Stage indicated by the the Output.