GMAT API Tradeoff Study

Thinking Systems, Inc. 6441 N Camino Libby Tucson, Arizona 85718

Introduction

This document describes a brief trade study performed at Thinking Systems that evaluated approaches to creating an application programming interface (API) into GMAT from external systems. The focus of the study is exposing GMAT functionality. It does not address access issues from the other side of the interface -- that is, issues involved with calling the API from the side of the client side consumers.

Figure 1 shows an overview of the systems involved in the API approaches. The focus of this study is providing access to GMAT's internal class structure in order to expose both simple and complex units of functionality through a structure labeled in the figure as the "GMAT API." Interfaces to this functionality are examined for use from Python and Java, and to a lesser extent from C, in order to make these elements available to flight dynamics analysts at Goddard Space Flight Center (GSFC). Thinking Systems provided pseudocode illustrating how these calls could be made for four different approaches to the interface design so that the tradeoffs between approaches could be evaluated. This work was presented to the analysts in GSFC's Code 585 during a site visit from January 30-February 3, 2012. This document presents that work, along with a discussion of the tradeoffs for the approaches evaluated.

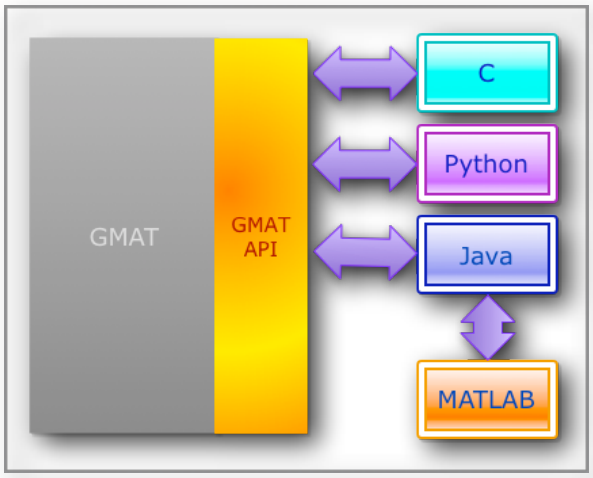

***Figure 1**: The GMAT API in Context*

Use Cases Discussed for Evaluation

Thinking Systems worked with Goddard to identify four basic use cases for evaluation during this study. Two of these cases were examined in some detail in order to develop the pseudocode illustrating the tradeoffs involved in each approach.

The use cases examined during this work are the following:

- Expose the GMAT time system converter and convert an epoch from A.1 modified Julian format to UTC modified Julian format.
- Expose the GMAT state converter and convert a Cartesian state into Keplerian elements.
- Expose the GMAT coordinate system converter and convert a state from Earth Centered Mean of J2000 Earth Equatorial Coordinates to Earth Body-Fixed Coordinates

- Expose the GMAT propagation subsystem and propagate a state from an initial input state for a single propagation step using a GMAT integrator

We examined the first and last of these use cases in some detail, including representative code for the former in Python, Java and MATLAB, and for the latter in Java. The initial version of the code for these is presented below.

API Design Approaches

We examined four approaches to meeting the needs of a robust, usable interface for the analysts. these approaches can be summarized as follows:

- Direct Native Class Access
- Class Access Extended Using the GMAT Engine
- Class Access Extended Using Customization of the GMAT Engine
- A Custom API Wrapper for GMAT Functionality

In more detail, these approaches consist of the following elements:

Direct Native Class Access

Using this approach, a user would create all of the building blocks needed for a piece of analysis. The user would connect these components together by hand, initialize the components, and use them as needed for analysis tasks.

Class Access Extended Using the GMAT Engine

This approach starts by creating and initializing the existing GMAT engine, which makes several core elements available for use by the analyst and constructs objects that are commonly used when working with GMAT user objects. Object interconnections and initialization are still performed by the user in user derived code.

Class Access Extended Using Customization of the GMAT Engine

GMAT's current Moderator/Configuration/Sandbox model is tailored to a system that runs as a multi-pass p-code compiler. If this approach is implemented, these core engine elements would be reworked to function as a p-code interpreter instead. In this mode, object interconnection and initialization would be performed "on the fly" in a single pass.

A Custom API Wrapper for GMAT Functionality

This final approach would hide some aspects of the current GMAT design behind a custom API designed to make GMAT functionality easy to learn and use.

A fifth approach was also discussed briefly: provide a set of native C wrappers for GMAT. This approach was not studied in depth during this work, but was prototyped in 2011, providing access to the GMAT force model for users of the MATLAB programming environment. This approach proved to be problematic because it relied on global objects created from inside of a shared library, the use of full GMAT scripting, including the execution of a mission control sequence prior to object access, limited (as currently implemented) exposure of object parameters, and rather crude control over the objects hidden behind the interface.

Additional Factors

There are several additional elements to be considered in this evaluation, which for the purposes of this document take the form of assumptions that need to be validated before fully committing to an approach. These additional factors are summarized here:

- MATLAB can act directly on Java objects. This capability is used to connect the Java based API into the MATLAB framework. In order to work correctly, this calling structure must be able to connect from the Java API to GMAT's shared libraries, which are coded in C++.
- There are tools available that simplify exposure of C++ code to both Java and Python. One such tool, the Simplified Wrapper and Interface Generator (SWIG), was selected as a viable candidate for this work. Before proceeding with that selection, SWIG should be used to generate Java and Python interfaces that are exercised on a small subset of the GMAT code to show that it works as advertised.
- GMAT's core code is currently built as a monolithic system contained in a single shared library named libGmatBase. In the sample code for use case 4 I break this library into separate components using a scheme defined when converting the Windows GCC build system into a set of Visual C++ 2010 solution and project files.
- The API deliverables resulting from this process will be built in a way that does not impact GMAT's system code. While there may be pieces of the system that are changed as a result of the API work, those pieces will be changed in a way consistent with the goals of the main GMAT system.
- Depending on the approach selected for this work, the GMAT API layer shown in Figure 1 may be small or large, and is built on top of core GMAT code, possibly refactored to be much more modular than is found in the current code base.

Use Case Examination

In this section, we examine the epoch conversion and propagation use cases in order to see how the user would interact with the API for each of the four design approaches described above.

Use Case 1: Epoch Conversion

The epoch conversion use case is the case examined in the most detail during this study. It provides both an example of the simplest type of functionality exposure possible and an illustration of the types of changes that occur with the different design approaches. On the other hand, much of the complexity of working with GMAT's internals is not apparent in this case because of the simplicity of the example. That complexity is best observed when we look at use case 4.

Thinking Systems constructed a shared library that exposed the minimum functionality needed by this use case, identified elements of the GMAT code that would benefit from minor refactoring to make the case simpler to use, and then proceeded to develop a rough draft of the coding that an analyst would need to write for each approach, working from Python, Java, or MATLAB.

In order for GMAT to expose epoch conversion to a user, 53 of GMAT's core code files need to be built into a shared library. Much of the code used for this library provides the minimal infrastructure that all GMAT classes require to function correctly: core definitions, default data values, exception and message handling facilities, and core data classes used throughout the system.

During the course of this evaluation, we noted that GMAT's epoch conversion code in inconsistent with the rest of the system in the following ways:

- The conversion code is written as function in a C++ namespace rather than static methods in a conversion class
- The code depends on two global data members that point to objects instantiated elsewhere.

These issues have been reported in GMAT's bug tracking system as defects that need to be addressed.

GMAT provides two core conversion functions (methods once the namespace is refactored, so I'll refer to them as methods in the following text) used to change epochs from one time system to another. Epoch conversions are made by calling the overloaded Convert methods with the signatures

```
Real Convert(const Real origValue, const Integer fromType,
             const Integer toType, Real refJd);
void Convert(const std::string &fromType, Real fromMjd, const std::string &fromStr,
             const std::string &toType, Real &toMjd, std::string &toStr,
             Integer format = 1);
```

For the purposes of this discussion, I'll use the latter method in the sample code because that method is used most frequently in GMAT when building epoch information for the graphical user interface (GUI).

The following sample pseudocode for use case 1 exercises that method following the same procedure:

1. Load the GMAT elements needed for the conversion
2. Define the variables needed for the call
3. Initialize as needed and call the method
4. Output the results

The following paragraphs present the code for this process using each of our four design approaches. Pseudocode is presented for Java, Python, and MATLAB.

Approach 1: Direct Native Class Access

This approach uses direct calls into code that mimics GMAT's internal classes using wrappers native to the language invoking the calls. In Java, the process of calling the epoch conversion method takes this form:

```
import GMAT.Foundation

double inTime = 21545.0;
double outTime;
string str;

LeapSecsFileReader lsfr = new LeapSecsFileReader("tai-utc.dat");
EopFile eop = new EopFile("eopc04.62-now");

TimeConverterUtil.SetEopFile(eop);
TimeConverterUtil.SetLeapSecsFileReader(lsfr);
TimeConverterUtil.Convert("A1ModJulian", inTime, "",
                          "UTCModJulian", outTime, str);

System.out.println(outTime);
```

Direct calls into the core classes requires that the user define two helper objects: a leap second file reader to retrieve the history of leap seconds applied to the data, and a file tracking the UT1 offsets arising from fluctuations in the Earth's rotation. Once these objects have been set on the converter, the Convert method can be called.

When implemented in Python, this process takes the form

```
import GMAT_Foundation

inTime = 21545.0
outTime = 0.0
Str = ""
```

```
lsfr = LeapSecsFileReader("tai-utc.dat")
eop = EopFile("eopc04.62-now")

TimeConverterUtil.SetEopFile(eop)
TimeConverterUtil.SetLeapSecsFileReader(lsfr)
TimeConverterUtil.Convert("A1ModJulian", inTime, "",
      "UTCModJulian",outTime, str)

print(outTime)
```

Aside from language peculiarities, the code looks identical. Similarly, when calling from MATLAB, the code takes a this form:

```
>> javaaddpathfullfile(gmatjarRoot,'foundation','GmatFoundation.jar')
>> inTime = 21545.0;
>> outTime = 0.0;
>> Str = '';
>> lsfr = LeapSecsFileReader("tai-utc.dat");
>> eop = EopFile("eopc04.62-now");
>> TimeConverterUtil.SetEopFile(eop);
>> TimeConverterUtil.SetLeapSecsFileReader(lsfr);
>> TimeConverterUtil.Convert("A1ModJulian", inTime, "", ...
   "UTCModJulian",outTime, str);
>> outTime

outTime =

     2.154499962923169e+04
```

(The output here, like the input, is a mock-up of expected results.)

Approach 2: Class Access Extended Using the GMAT Engine

When the GMAT engine is available for use, the user has access to the object creation and initialization interfaces used by processes that connect to the core code. This access includes basic object setup, as is illustrated in the Java code for use case 1:

```
import GMAT.Foundation
Moderator.Instance().Initialize();

double inTime = 21545.0;
double outTime;
string str;

TimeConverterUtil.Convert("A1ModJulian", inTime, "",
                          "UTCModJulian", outTime, str);

System.out.println(outTime);
```

GMAT creates the system executive, called the Moderator, as a singleton instance in the second line of this example. On the same line, that instance is initialized. Part of the initialization process is the construction of the helper classes used in the epoch conversion code. Working through the Moderator, the user no longer needs to be concerned with that aspect of the initialization process.

The Python code for this approach shows the same features:

```
import GMAT_Foundation
Moderator.Instance().Initialize()

inTime = 21545.0
outTime = 0.0
Str = ""
TimeConverterUtil.Convert("A1ModJulian", inTime, "",
      "UTCModJulian",outTime, str)

print(outTime)
```

as does the MATLAB code:

```
>> javaaddpathfullfile(gmatjarRoot,'foundation','GmatFoundation.jar')
>> Moderator.Instance().Initialize();
>> inTime = 21545.0;
>> outTime = 0.0;
>> Str = '';
>> TimeConverterUtil.Convert("A1ModJulian", inTime, "", ...
   "UTCModJulian",outTime, str);
>> outTime

outTime =

     2.154499962923169e+04
```

Approach 3: Class Access Extended Using Customization of the GMAT Engine

The key feature of using a customized Moderator for the engine is the ability to handle initialization in a mode more suitable to running as an interpreted p-code system rather than as a compiled system. That aspect is not apparent in this simple example, as can be seen when we look at the Java code that implements epoch conversion:

```
import GMAT.Foundation
Moderator.Instance().Initialize();

double inTime = 21545.0;
double outTime;
string str;

TimeConverterUtil.Convert("A1ModJulian", inTime, "",
                          "UTCModJulian", outTime, str);

System.out.println(outTime);
```

This code is identical to that shown in Approach 2, as is the Python code:

```
import GMAT_Foundation
Moderator.Instance().Initialize()

inTime = 21545.0
outTime = 0.0
Str = ""

TimeConverterUtil.Convert("A1ModJulian", inTime, "",
      "UTCModJulian",outTime, str)

print(outTime)
```

and the MATLAB implementation:

```
>> javaaddpathfullfile(gmatjarRoot,'foundation','GmatFoundation.jar')
>> inTime = 21545.0;
>> outTime = 0.0;
>> Str = '';
>> TimeConverterUtil.Convert("A1ModJulian", inTime, "", ...
   "UTCModJulian",outTime, str);
>> outTime

outTime =

     2.154499962923169e+04
```

Approach 4: A Custom API Wrapper for GMAT Functionality

Implementation using a custom wrapper allows us to hide additional pieces of GMAT from the user, and thus provides an opportunity to make the GMAT calls as simple as possible. An example of the simplification that could be achieved is the hiding of the GMAT engine components behind the interface. For this use case, that would separate the user from the need to work directly with a Moderator, as can be seem in this Java mock-up of one API approach:

```
import GMAT.Foundation
GmatInterface.Initialize();

double inTime = 21545.0;
double outTime;

TimeConverterUtil.Convert("A1ModJulian", inTime, "UTCModJulian",
                          outTime);

System.out.println(outTime);
```

Notice that the user no longer needs to know about the existence of a Moderator in this code, and the API has been further tailored to omit two of the helper variables that were required but unused in the call to convert the data. These features are also seen in the Python pseudocode:

```
import GMAT_Foundation
GmatInterface.Initialize

inTime = 21545.0
outTime = 0.0

TimeConverterUtil.Convert("A1ModJulian", inTime, "UTCModJulian",
                          outTime)

print(outTime)
```

and the MATLAB:

```
>> javaaddpathfullfile(gmatjarRoot,'foundation','GmatFoundation.jar')
>> inTime = 21545.0;
>> outTime = 0.0;
>> TimeConverterUtil.Convert("A1ModJulian", inTime,...
   "UTCModJulian",outTime);
>> outTime

outTime =
```

```
2.154499962923169e+04
```

Use case 1: Summary

The following table summarized the features of these approaches to epoch conversion:

	Init LOC	Call LOC	Visible Classes	Extra Vars	Call Params	Notes
Approach 1: Export Raw Classes	3	3	3	3	6	
Approach 2: Use Current Moderator	2	1	1	3	6	Requires a startup file
Approach 3: Use New Moderator	2	1	1	3	6	
Approach 4: Class Interface	2	1	1	2	4	

One feature that is not readily apparent in the discussion of approach 2 above is noted in this table: in order for a user to work with the current GMAT engine, a GMAT startup file is required. This requirement can be handled in approaches 3 and 4 through the application of default settings.

This study does not address use cases 2 and 3 at this time. The state conversion requirements are similar to those found for epoch conversion: three additional files need to be included in the shared library, the input state needs to be a six element vector plus an epoch, and the output state using native calls is packed into a GMAT Rvector6 object, so that object needs to be exposed for use in the interface. Coordinate system conversion as called for in use case 3 requires access to many of GMAT's classes, and places a large burden of configuration and management on the user. These same requirements are placed on the user when working with use case 4, as will be seen in the next section.

Use Case 4: Propagation

The propagation use case is a fairly complex set of objects acting together to take a single propagation step for a single spacecraft orbiting Mars. The use case examined here consists of the following elements:

- Force Model:
  - Mars full field 4x4 potential, using the Mars 50c model
  - MarsGRAM 2005 atmosphere model
  - Solar Radiation Pressure
  - Sun and Jupiter point mass perturbations
- Prince-Dormand 7(8) Integrator
- Spacecraft state defined in Mars MJ2000 coordinates
- State elements specified as the Keplerian state [3500 0.01 69 0 0 0]
- Default values used whenever possible

Note that this use case requires both an orbital element converter and a coordinate system converter, as called for in use cases 2 and 3, respectively. The focus of use case 4 is the configuration and use of the propagation subsystem. Some of the details needed for the conversion elements may be missing because of that focus.

This use case requires interactions with several classes that are not directly referenced in the problem statement above. In particular, the user needs to work with

- the SolarSystem class
- the CoordinateSystem class
- the AxisSystem class
- the PropSetup class

In the code samples for this use case, we also make the following assumptions:

- The ODEModel class sets solar system structures needed when the ODEModel is initialized
- The plugin infrastructure needed for the current MarsGRAM atmosphere model is handled outside of the code shown here
- The default epoch (A.1 modified Julian date 21545.0 - A.1 Gregorian epoch January 1, 2000 12:00:00.000) is sufficient for demonstration purposes here.

Approach 1: Direct Native Class Access

This approach places the largest burden on the user because all of the object creation and use has to be hand coded. The pseudocode needed to implement a single propagation step takes this form:

```
// Load modules -- this might be a single entity
import Gmat.Foundation
import Gmat.Spacecraft
import Gmat.Environment     // For SolarSystem
import Gmat.Propagation
import Gmat.MarsGram

// We need planets, Sun, etc, all created in the SolarSystem
SolarSystem ss = new SolarSystem();

// Define Mars Coordinate System
CoordinateSystem marsMJ2k = new
   CoordinateSystem("CoordinateSystem","marsMJ2k");

// Set J2000Body and Origin
marsMJ2k.SetStringParameter("J2000Body", "Earth");
marsMJ2k.SetStringParameter("Origin", "Mars");
CelestialBody earth = ss.GetBody("Earth");
marsMJ2k.SetRefObject(earth, Gmat::SPACE_POINT, "Earth");
CelestialBody mars = ss.GetBody("Mars");
marsMJ2k.SetRefObject(mars, Gmat::SPACE_POINT, "Mars");

// create MJ2000Eq AxisSystem
AxisSystem axis = new AxisSystem("MJ2000Eq", "MJ2000Eq_Mars");
marsMJ2k.SetRefObject(axis, Gmat.AXIS_SYSTEM, axis.GetName());

// Define the internal Coordinate System
CoordinateSystem earthMJ2k = new
   CoordinateSystem("CoordinateSystem","earthMJ2k");

// Set J2000Body and Origin
earthMJ2k.SetStringParameter("J2000Body", "Earth");
earthMJ2k.SetStringParameter("Origin", "Earth");
CelestialBody earth = ss.GetBody("Earth");
```

```
earthMJ2k.SetRefObject(earth, Gmat::SPACE_POINT, "Earth");

// create MJ2000Eq AxisSystem
AxisSystem intAxis = new AxisSystem("MJ2000Eq", "MJ2000Eq_Earth");
earthMJ2k.SetRefObject(intAxis, Gmat.AXIS_SYSTEM, axis.GetName());

// Axis system cloned in the CS, so free the memory
//delete axis;  // In Java, auto garbage collection does this

// Complete initialization
marsMJ2k.SetSolarSystem(ss);
marsMJ2k.Initialize();

// Setup the spacecraft
Spacecraft marsOrb = new Spacecraft("MarsOrb");

marsOrb.SetStringParameter("CoordinateSystem", "marsMJ2k");
marsOrb.SetStringParameter("DisplayStateType", "Keplerian");
marsOrb.SetRealParameter("SMA", 3500.0));
marsOrb.SetRealParameter("ECC ", 0.01);
marsOrb.SetRealParameter("INC ", 69.0);
marsOrb.SetRealParameter("RAAN ", 0.0);
marsOrb.SetRealParameter("AOP ", 0.0);
marsOrb.SetRealParameter("TA", 0.0);

// Set up s/c attitude here if needed -- it looks like the current
// implementation requires it.

marsOrb.SetReferenceObject(marsMJ2k, Gmat.COORDINATE_SYSTEM, “MarsMJ2k”);
marsOrb.SetReferenceObject(earthMJ2k, Gmat.COORDINATE_SYSTEM, “MarsMJ2k”);
marsOrb.Initialize();

// Create force model
ForceModel fm = new ForceModel("fm");

// Define forces
fm.TakeAction("ClearDefaultForce");
fm.SetStringParameter("CentralBody", "Mars");

GravityField gf = new GravityField("MarsPot", "Mars", 4, 4);
gf.SetStringParameter("PotentialFile", "Mars50c.cof");
fm.AddForce(gf);

PointMassForce jove = new PointMassForce("JovePot");
jove.SetBooleanParameter("PrimaryBody", false);
jove.SetRealParameter("GravConst",
   GmatSolarSystemDefaults.PLANET_MU[GmatSolarSystemDefaults.JUPITER]);
fm.AddForce(jove);

PointMassForce sol = new PointMassForce("SunPot");
sol.SetBooleanParameter("PrimaryBody", false);
sol.SetRealParameter("GravConst", STAR_MU);
fm.AddForce(sol);

DragForce mdrag = new DragForce("MarsDrag");
AtmosphereModel atm = new MarsGRAM2005("MarsGRAM2005", "MarsGRAM");
mdrag.SetInternalAtmosphereModel(atm);
fm.AddForce(atm);
```

```
SolarRadiationPressure srp = new SolarRadiationPressure("SRP");
srp.SetRealParameter("BodyRadius",
   GmatSolarSystemDefaults.PLANET_EQUATORIAL_RADIUS[
   GmatSolarSystemDefaults.MARS]);
fm.AddForce(srp);

PropSetup prop = new PropSetup("prop");
PrinceDormand78 pd78 = new PrinceDormand78("");
prop.SetPropagator(pd78);
prop.SetODEModel(fm);

// Set Spacecraft
PropagationStateManager psm = prop.GetPropStateManager();
psm.SetObject(marsOrb);

prop.Initialize();

// Take one propagation step
prop.Step();
```

It is important to note that in the code above, order is important - the coordinate system must be built after the solar system and before the spacecraft; initialization of the spacecraft depends on initialization of the objects that it depends on, and so forth. This ordering will be necessary for any approach we take, but becomes more complicated - and in need of cleaner documentation - when the access methods are made to lower level objects.

Approach 2: Class Access Extended Using the GMAT Engine

Once the GMAT engine components are available for use, the user has access to both the methods of the Moderator and the Interpreter subsystem. These components make it simpler to use the interface for users that already know GMAT scripting, because the user can pass in script lines through the API and expect GMAT to interpret those lines correctly. This approach is used extensively in the following pseudocode:

```
// Load modules -- this might be a single entity
import Gmat.*
Moderator mod = Moderator.Instance();
mod.Initialize();
ScriptInterpreter interpreter = ScriptInterpreter.Instance();
// Mod: Add an InterpretLine method to ScriptInterpreter

// Build the needed CS
CoordinateSystem marsMJ2k = mod.CreateCoordinateSystem("marsMJ2k");
interpreter.InterpretLine("marsMJ2k.Origin = Mars");
interpreter.InterpretLine("marsMJ2k.Axes = MJ2000Eq");

// Build the Spacecraft
Spacecraft sat = mod.CreateSpacecraft("MarsOrb");
interpreter.InterpretLine("MarsOrb.CoordinateSystem = marsMJ2k");
interpreter.InterpretLine("MarsOrb.DisplayStateType = Keplerian");
interpreter.InterpretLine("MarsOrb.SMA  = 3500.0");
interpreter.InterpretLine("MarsOrb.ECC  = 0.01");
interpreter.InterpretLine("MarsOrb.INC  = 69.0");
interpreter.InterpretLine("MarsOrb.RAAN = 0.0");
interpreter.InterpretLine("MarsOrb.AOP  = 0.0");
interpreter.InterpretLine("MarsOrb.TA   = 0.0");

// Create force model
```

```
ForceModel fm = mod.CreateODEModel("fm");

interpreter.InterpretLine("fm.CentralBody = Mars");
interpreter.InterpretLine("fm.PrimaryBodies = {Mars}");
interpreter.InterpretLine("fm.PointMasses = {Jupiter, Sun}");
interpreter.InterpretLine("fm.Drag = MarsGRAM2005");
interpreter.InterpretLine("fm.SRP = On");
interpreter.InterpretLine(
   "fm.GravityField.Mars.PotentialFile = 'Mars50c.cof'");

PropSetup Prop = mod.CreatePropSetup("Prop");
interpreter.InterpretLine("Prop.FM = DefaultProp_ForceModel");
interpreter.InterpretLine("Prop.Type = PrinceDormand78");

interpreter.FinalPass();

//Build object maps (This is all pseudocode!!!)
ObjectArray ObjectMap = new ObjectArray;
ObjectArray GlobalMap = new ObjectArray;

ObjectMap.Add(marsMJ2k);
ObjectMap.Add(sat);
ObjectMap.Add(fm);
ObjectMap.Add(Prop);
SolarSystem solarSys = mod.GetSolarSystemInUse();
CoordinateSystem iCS = mod.GetInternalCoordinateSystem();

// Initialize all of the stuff that we use
objInit = new ObjectInitializer(solarSys,ObjectMap,GlobalMap,iCS);

try {
      objInit->InitializeObjects();
}
catch (BaseException &be) {
...}

// Set Spacecraft
PropagationStateManager psm = Prop.GetPropStateManager();
psm.SetObject(marsOrb);

// Take one propagation step
prop.Step();
```

In GMAT, objects created through the Moderator with names are stored in a local vector of objects called the configuration. The objects in the configuration are managed by a component called the ConfigurationManager. Those objects are preserved in their scripted state, and cloned into a GMAT Sandbox when a mission is run. In the approach show above, we are sidestepping the Sandbox cloning step and working directly with the objects in the configuration. Those objects still need to be initialized, though. That initialization occurs beginning when the ScriptInterpreter makes its final pass through the configured objects and proceeding through calls to GMAT's object initializer in these lines of code:

```
interpreter.FinalPass();

//Build object maps (This is all pseudocode!!!)
ObjectArray ObjectMap = new ObjectArray;
ObjectArray GlobalMap = new ObjectArray;

ObjectMap.Add(marsMJ2k);
```

```
ObjectMap.Add(sat);
ObjectMap.Add(fm);
ObjectMap.Add(Prop);
SolarSystem solarSys = mod.GetSolarSystemInUse();
CoordinateSystem iCS = mod.GetInternalCoordinateSystem();

// Initialize all of the stuff that we use
objInit = new ObjectInitializer(solarSys,ObjectMap,GlobalMap,iCS);

try {
      objInit->InitializeObjects();
}
catch (BaseException &be) {
...}
```

It is this set of processes that would be simplified using an approach that replaces the engine code with an engine built for p-code interpreted use rather than compiled use, as proposed in approach 3.

Approach 3: Class Access Extended Using Customization of the GMAT Engine

Reworking GMAT's engine for use in an interpreted environment requires that the Moderator be rewritten to function in interpreted mode, that the interpreters address issues currently handled in the final pass through the p-code compiler, and that the object initializer work in a piecemeal fashion, so that object initialization is performed on a line-by-line basis rather than all at once following the final pass through the interpreters. The object initializer would need to work hand-in-hand with the interpreter, and would not be visible to the user.

This approach is illustrated in the following pseudocode:

```
// Load modules -- this might be a single entity
import Gmat.*
Moderator mod = Moderator.Instance();
mod.Initialize();
ScriptInterpreter interpreter = ScriptInterpreter.Instance();
// Mod: Add an InterpretLine method to ScriptInterpreter

// Build the needed CS
CoordinateSystem marsMJ2k = mod.CreateCoordinateSystem("marsMJ2k");
interpreter.InterpretLine("marsMJ2k.Origin = Mars");
interpreter.InterpretLine("marsMJ2k.Axes = MJ2000Eq");

// Build the Spacecraft
Spacecraft sat = mod.CreateSpacecraft("MarsOrb");
interpreter.InterpretLine("MarsOrb.CoordinateSystem = marsMJ2k");
interpreter.InterpretLine("MarsOrb.DisplayStateType = Keplerian");
interpreter.InterpretLine("MarsOrb.SMA  = 3500.0");
interpreter.InterpretLine("MarsOrb.ECC  = 0.01");
interpreter.InterpretLine("MarsOrb.INC  = 69.0");
interpreter.InterpretLine("MarsOrb.RAAN = 0.0");
interpreter.InterpretLine("MarsOrb.AOP  = 0.0");
interpreter.InterpretLine("MarsOrb.TA   = 0.0");

// Create force model
ForceModel fm = mod.CreateODEModel("fm");

interpreter.InterpretLine("fm.CentralBody = Mars");
interpreter.InterpretLine("fm.PrimaryBodies = {Mars}");
interpreter.InterpretLine("fm.PointMasses = {Jupiter, Sun}");
```

```
interpreter.InterpretLine("fm.Drag = MarsGRAM2005");
interpreter.InterpretLine("fm.SRP = On");
interpreter.InterpretLine(
   "fm.GravityField.Mars.PotentialFile = 'Mars50c.cof'");

PropSetup Prop = mod.CreatePropSetup("Prop");
interpreter.InterpretLine("Prop.FM = DefaultProp_ForceModel");
interpreter.InterpretLine("Prop.Type = PrinceDormand78");

// Set Spacecraft
PropagationStateManager psm = Prop.GetPropStateManager();
psm.SetObject(marsOrb);

// Take one propagation step
prop.Step();
```

As can be seen by the code above, this approach closely mimics GMAT scripting. Some of the object member access can be removed from the approach using a custom API, as is shown below for approach 4.

Approach 4: A Custom API Wrapper for GMAT Functionality

Once again a custom interface can be used to wrap the calls into GMAT's object model, hiding some of the details of the engine code and consolidating features that require separate lines. A full design for such an API is beyond the scope of this document, but a starting draft on such an interface is not. The resulting pseudocode would look something like this:

```
// Load modules -- this might be a single entity
import Gmat.*
GmatInterface.Initialize();

// Build the needed CS
CoordinateSystem marsMJ2k = CreateCoordinateSystem("marsMJ2k");
InterpretLine("marsMJ2k.Origin = Mars");
InterpretLine("marsMJ2k.Axes = MJ2000Eq");

// Build the Spacecraft
Spacecraft sat = CreateSpacecraft("MarsOrb");
InterpretLine("MarsOrb.CoordinateSystem = marsMJ2k");
InterpretLine("MarsOrb.DisplayStateType = Keplerian");
InterpretLine("MarsOrb.State  = [3500.0 0.01 69.0 0.0 0.0 0.0]");

// Create force model
ForceModel fm = CreateODEModel("fm");

InterpretLine("fm.CentralBody = Mars");
InterpretLine("fm.PrimaryBodies = {Mars}");
InterpretLine("fm.PointMasses = {Jupiter, Sun}");
InterpretLine("fm.Drag = MarsGRAM2005");
InterpretLine("fm.SRP = On");
InterpretLine("fm.GravityField.Mars.PotentialFile = 'Mars50c.cof'");

Propagator Prop = CreatePropagator("Prop");
InterpretLine("Prop.FM = DefaultProp_ForceModel");
InterpretLine("Prop.Type = PrinceDormand78");

// Set Spacecraft
Prop.SetObject(marsOrb);

// Take one propagation step
```

```
Prop.Step();
```

Note that the engine components -- the Moderator and the ScriptInterpreter -- along with some of the class names and features, like the PropagationStateManager inside of the PropSetup can be hidden from the user if this approach is taken.

Use case 4: Summary

The following table summarized the features of these approaches to propagation:

	Total LOC	Object Definition	Initialization	Execution	Notes
Approach 1: Export Raw Classes	66	38	27	1	Slightly inflated; shows 5 lines of imports where others use 1
Approach 2: Use Current Moderator	44	25	16	3	Requires a startup file
Approach 3: Use New Moderator	29	25	1	3	
Approach 4: Class Interface	21	18	1	2	Slightly deflated; the state is set in a single call rather than 6

The reader is cautioned here to remember that the pseudocode presented in these examples has not been built into a system, so there are features that are likely missing in all examples, and particularly for approaches 3 and 4, which are based on a current best guess at what can be achieved starting from GMAT's core structures. Nevertheless, these cases can be used to make a first cut estimate at the tradeoff between selecting an approach and the resulting level of effort involved. That tradeoff is presented in the following section.

Design Tradeoffs

This study has examined four approaches to exposing GMAT functionality to mission analysts. In this summary, these approaches are assessed in terms of the level of effort required to make each work, the level of refactoring projected for the GMAT code base, the ease of use for the resulting system when used by analysts familiar with GMAT, and additional notes and comments. These findings are summarized in a table at the end of this section.

Approach 1: Direct Native Class Access

Approach 1 is the simplest approach examined in this work. The table below identifies advantages and disadvantaged to this approach.

Advantages	Disadvantages
• Simplest to implement • Users know exactly how things are interconnected	• Most verbose to use • Requires extensive documentation or intimate knowledge of GMAT internals • Users must do the object to object interconnection and initialization by hand in the right order

Approach 2: Class Access Using the GMAT Engine

The second approach uses much of the GMAT engine and existing classes to generate the API. The user effort is reduced, at the expense of additional development work required to expose the GMAT engine.

Advantages	Disadvantages
• Closely maps to GMAT scripting • Can be implemented as a natural progression from approach 1 • Significant portion of testing is identical to GMAT testing	• Complex features require performing operations that mimic the Sandbox • Requires a startup file and settings that match GMAT • Works in a p-code compiled mode: Objects are defined and configured, then initialized, then used

Approach 3: Class Access Using a Custom GMAT Engine

Approach 3 adds the ability to interact with GMAT objects from a command line tool -- for example, the MATLAB user interface for Java/MATLAB users, or the IDLE environment for Python users.

Advantages	Disadvantages
• Users can interact with objects dynamically • Can be implemented as a natural progression from approach 1 • Startup issues can be addressed using default settings	• High level of effort to implement this approach • Testing is interactive when running in interpreter mode

Approach 4: Access Using a Custom API

The final approach provides a programming interface customized to the needs of the user community. All of GMAT's object infrastructure is hidden from user view, so that the user only defines and uses the components directly related to the analysis problem space.

Advantages	Disadvantages
• Tailored to the user community • Most closely resembles scripting	• Large development cost to implement • Testing costs are likely to be high

Summary of the Approaches

The data presented in this section is extremely preliminary, and should be further evaluated prior to use in formal estimations. Level of efforts may be off by as much as a factor of 4 or so, depending on the complexity of the selected approach and the features that are targeted for exposure through the interface.

The following table evaluates the approaches using the following criteria (Note that the scales are intended to indicate that larger numbers require more effort. Thus a usability of 1 is much better than a 10, while a level of effort of 1 is also better than a 10):

- **Complexity**: A rating of the approach complexity on a scale of 1 to 10, where 1.0 means using GMAT as it currently exists and 10.0 means completely encasing GMAT inside of a custom interface that has no resemblance to the existing system.
- **Implementation Preparations**: An estimate of the amount of work required prior to development in order to proceed with the selected approach, where 1.0 means no preparation is required and 10.0 means that a full architectural specification is required.
- **Development level of effort (LOE)**: The amount of effort that would need to be expended to implement the approach, where a level of effort of 1.0 corresponds to the effort needed to access GMAT functionality from a C++ program linking directly to the GMAT base code
- **Usability**: The amount of effort projected for GMAT savvy user to perform analysis using the API from one of the platforms examined, rated on a scale from 1 to 10 with 1 being the simplest to use, and 10 being nearly unusable.
- **Documentation Effort**: A factor estimating the amount of training materials that would be needed to teach a GMAT savvy user to use the API, where a rating of 1 means that the existing documentation (including the Doxygen generated code documents) is sufficient, and a rating of 10 means that extensive documentation would be required.
- **External dependencies**: This category lists factors that need to be included when considering the approach tradeoffs. For example, all of these approached require some form os C++ to Java and Python wrapper code, so they depend on a tool that provides such code.

Approach	Complexity	Prep	LOE	Usability	Docs	Externals
Core classes	1.8	1.5	2	8	4	SWIG, Doxygen
Current engine	2	2	3	4	3	SWIG
Custom engine	4	4	7	3	2	SWIG
Custom API	7	8	10	2	2	SWIG

The data provided in this table is, by its nature, extremely ill defined. In order to refine it, the following steps should be taken:

- All of these approaches depend on SWIG providing wrapper code. This approach needs to be validated. The following factors need to be part of that validation:
  - MATLAB needs to call through the SWIG wrapper into the wrapped shared library and access the wrapped classes both for reading and writing, on all of the platforms that are targeted for support.
  - SWIG code generated from GMAT header files without extensive customization is preferred to making API specific SWIG inputs because it minimizes the effort needed to synchronize ongoing GMAT development with the API. This usage of GMAT headers needs to be validated.
  - Python needs to follow the same process as MATLAB.
- The feature set that is exposed through the API needs to be defined. The use cases evaluated here all access GMAT resources, but no commands. That means, for example, that there is no access to propagation with stopping conditions, no access to the solver control sequence for targeting, optimization, or estimation, and no

access to the outputs generated from performing a run. Nor is it clear what it means to "perform a run" in this context.

- The data presented here is taken from a development point of view. The effort required to test the resulting system is extremely difficult to estimate from that perspective, but need to be considered as part of the process of determining how to proceed with this effort.

One final note on the approach going forward: both approaches 2 and 3 build on the exposure of GMAT class internal access that is required for approach 1. One possible avenue for proceeding is to invest some effort into implementing approach 1 for a select set of functions, designing the core component access in a way that does not preclude exposure of the engine components once the development and test teams have had some exposure to the work involved. At that point, the team would have more experience working with the API, which would help to clarify how best to proceed going forward.

Next Steps

In order to proceed with this project, Thinking Systems recommends that the following steps be taken:

- Preproposal (estimated to be 3-4 weeks work):
  - ✴ Build a small SWIG based test case and run it on all platforms of interest. Here is the set I'd select
    - ✦ Expose the state converter or the epoch converter (Groundwork for this is already in place)
    - ✦ Expose the message interface (This allows debug messages to be seen)
  - ✴ Connect the test case to MATLAB and show that it works.
  - ✴ Decide if SWIG is viable for this work.
  - ✴ Identify a target set of features for initial exposure through a GMAT API.
- Propose the work, using lessons learned from the preproposal work to make the proposal stronger.
- Build the initial API functionality.
- Iterate to build additional functionality as needed.

Appendix: External Tools

The approaches described in this document depend on the ability to create wrapper code that exposes GMAT's native C++ objects to Java and Python. These wrappers can be created either by hand, building wrapper code for each exposed object, or using a software tool that automates the process. Constructing wrapper code by hand is time consuming and error prone, and presents maintenance issues for code that is modified to meet evolving project needs. Two tools were examined as automation candidates, and are outlined here.

JNAerator

(From Wikipedia, http://en.wikipedia.org/wiki/JNAerator)

> JNAerator is a computer programming tool for the Java programming language which automatically generates the Java native access (JNA) or BridJ code needed to call C and Objective-C libraries from Java code. It reads in ANSI C header files and emits Java code. Some optional customization can be done through command line options, which can be saved in configuration files. JNAerator does not need any native compiling beyond that of the targeted dynamic library (all of the glue code is in Java), which helps simplify the process of binding Java to C native libraries when compared to Java Native Interface (JNI)-based means.
>
> Its output is typically larger and harder to use than hand-crafted JNA bindings, but it saves time and effort for bindings of large libraries with JNA.

JNAerator provides interfaces that are accessed from Java based on ANSI C compliant header files by building JNA based Java code. It has support for C++, but does not currently support some advanced features of C++ like template classes and multiple inheritance, both of which are part of GMAT's core code. JNAerator output does not provide Python interfaces.

A list of projects using JNAerator can be found at http://code.google.com/p/jnaerator/wiki/ProjectsUsingJNAerator.

SWIG

(From Wikipedia, http://en.wikipedia.org/wiki/SWIG)

> SWIG (Simplified Wrapper and Interface Generator) is an open source software tool used to connect computer programs or libraries written in C or C++ with scripting languages such as Lua, Perl, PHP, Python, R, Ruby, Tcl, and other languages like C#, Java, Modula-3, Objective Caml, Octave, and Scheme. Output can also be in the form of XML or Lisp S-expressions.

SWIG supports a much more extensive set of languages, providing support for both our potential target languages (Java and Python) along with Perl, C#, Ruby, and Octave. It supports templates and multiple inheritance, and therefore is a better fit for some of the classes in GMAT that use these features.

A list of projects using SWIG can be found at http://www.swig.org/projects.html.