Java for Flight Software

Edward G. Benowitz, Albert F. Niessner
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
{Edward.G.Benowitz, Albert.F.Niessner}@jpl.nasa.gov

***Abstract*— This work involves developing representative mission-critical spacecraft software using the Real-Time Specification for Java(RTSJ)[1]. Utilizing a real mission design, this work leverages the original flight code from NASA's Deep Space 1(DS1), which flew in 1998. However, instead of performing a line-by-line port, the code is re-architected in pure Java™, using best practices in Object-Oriented(OO) design. We have successfully demonstrated a portion of the spacecraft attitude control and fault protection, running on a standard Java platform, and are currently in the process of taking advantage of the features provided by the RTSJ. Our goal is to run on flight-like hardware, in closed-loop with the original spacecraft dynamics simulation.**

In re-designing the software from the original C code, we have made a number of observations on adopting OO techniques for flight software development, and we explain the benefits of this approach. We have taken advantage of design patterns[7], and have seen a strong mapping from certain patterns to the flight software. The state design pattern eliminates the need for long, error-prone switch statements. The facade pattern is used for communication between threads, hiding queues where necessary, or allowing direct method calls. To ensure the correctness of measurement units, numerical computations are performed via an abstraction layer that checks measurement units at compile-time.

Our approach places an emphasis on pluggable technology. Interfaces, in conjunction with a façade pattern, expose only the behavior of a subsystem, rather than exposing its implementation details. Since the RTSJ reference implementation does not currently support debugging, we chose to apply pluggable technology to the scheduler and memory allocation interfaces. Thus, real-time client code can be run on a standard Java virtual machine, allowing the code to be debugged in a graphical development environment on a desktop PC at the cost of decreased real-time performance. Once non-real-time issues have been debugged, the real-time aspects can be debugged in isolation on an RTSJ-compliant virtual machine.

I. Introduction

A. Motivation

Flight software has a high development cost, due in part to the difficulty in maintaining the code. The lack of maintainability stems from the limitations of current implementation languages, which we now discuss. There is a lack of strong type-checking and parameter checking. Memory can easily be corrupted due to the lack of pointer checking and array-bounds checking. Without operating system protections, these problems can occur as silent failures. Concurrency primitives are very low-level, and are not part of the language. A typical program will abound with error-prone switch statements and preprocessor directives. And only a globally shared namespace is available.

Pluggable components cannot be expressed with traditional flight software techniques. A pluggable component is a software article which only exposes its interface (behavior) and not its implementation. Pluggable components allow different implementations to be swapped, without requiring modifications to the rest of the code. The C language does not provide this level of encapsulation mostly because of its procedural orientation. Although C++ attempts to provide encapsulation, multiple inheritance problems exist. Additionally, the encapsulation can easily be broken by using the friend keyword.

B. Advantages of Java

To address these issues, we are investigating Java as an implementation language for flight software. Java improves maintainability with its strong type-checking at both compile-time and run-time. Additionally, Java checks array boundaries, and ensures that variables are initialized. Standard Java provides automatic memory management, and Real-Time Java allows several forms of manual memory management where required (see III-B for details). Multi-threading and higher-level concurrency primitives are built into the language as well. Java can easily express pluggable components, provides for full encapsulation, and allows single inheritance with multiple interface inheritance. Java also provides extensibility through inheritance and dynamic class-loading.

Aside from the advantages of the language itself, the Java platform includes a large standard class library with support for most programming needs. Due to the large Java developer community, additional Java components are available from the internet, often for free.

According to NIST[9], Java's higher level of abstraction leads to increased programmer productivity. The Java platform, coupled with Java language, improves application portability. Additionally, Java is easier to master than C++, and supports component integration and reuse.

C. Approach

We specifically chose to favor maintainability above all else during the architectual, design, and implementation phases of the development. Maintainability requires making extensive use of design patterns, taking full advantage of Java language features, using pluggable technology, and making appropriate use of commercial, off-the-shelf libraries and tools. During the performance evaluation phase, it is known that some maintainability will be sacraficed in order to optimize parts

of the system to meet performance requirements. However, empirical evidence from systematic profiling of the application will dictate where and what is to be optimized, as opposed to prematurely attempting to optimize based on intuition, instincts, and/or assumptions of behavior.

D. Tools

COTS graphical development tools were used extensively in this project. Specifically, the open-source Eclipse[6] integrated development environment provided graphical code editing, browsing, debugging, and refactoring capabilities. Headway's Review[8] product was used to graphically inspect our design, and allowed us to maintain a consistent architecture. Additionally, JProbe[13] was used to examine memory usage, and to identify critical regions for future optimization.

An RTSJ-compliant virtual machine is required for running real-time Java applications. In addtion to the the RTSJ reference implementation, several additional RTSJ implementations are just now becoming available: JRate[3], OVM[10], and FLEX[11].

II. Patterns

A. States

One of the more common design patterns we used was the states design pattern. To appreciate the usefulness of the pattern, we first describe the error-prone behavior which this pattern eliminates. We then discuss the advantages of the states design pattern, and show examples of how we applied the pattern to our attitude control and our fault protection subsystems.

1) Problems with past state representations : In the past, states were represented as either booleans or enumerated types that typically generated a long switch statement which selects the appropriate action based on the state variable. An example of typical code is shown in Algorithm 1. There are two major problems with the switch statement approach.

Firstly, the programmer must manually construct a switch statement, allowing the possibility that the programmer forgets a break statement. The programmer must also remember to consider all possible states within a switch statement. If the programmer forgets a particular state, at best the error will be caught by an assertion statement at run-time.

Secondly, the switch statement lacks extensibility. Consider the scenario in which one new state is needed, which amounts to adding an item to the enumeration. Every possible method call involving the state variable would have to be manually found and updated. If the new state were not included, this error would not be caught until run-time, assuming the programmer was diligent enough to use the assertions at the end of switch statements.

2) Advantages of the States Design pattern: The state pattern eliminates the need for long switch statements and repeated checking of flags. Instead, each state is a separate class but implements an interface common across the states. In this way, we can use polymorphism to automatically determine the appropriate code to execute in a given state, rather than manually checking a flag. Clients of a state class call methods directly on the state interface, while maintaining a reference to an instance of this interface. When the state needs to be changed, the reference to the interface is changed with a simple assignment to a different implementation of the interface. With this approach, we can ensure that all states have an implementation of the required methods, because of compile-time checking. There is no need for a run-time assertion check on this, enabling errors to be detected earlier. Algorithm 2 shows how each state provides its own implementation of doAction(), cleanly separating which code logically belongs to each state. We see that dynamic dispatch is used to automatically call the correct state, instead of manually having to check the state variable at the beginning of a function, as was the case in the traditional example in Algorithm 1. Note that all the code associated with a particular state is neatly confined to a single class. If a new state becomes necessary, a new implementation of the interface can be created. This is much cleaner than having to manually track the usage of an enumerated type throughout numerous functions containing complex switch statements.

Algorithm 1 Using switch statements for states in C

```
enum colorstate{ red, green, yellow};
void doAction(colorstate current_color)
{
  switch(current_color)
  {
     case red:
        break;
     case green:
        break;
     case yellow:
        break;
     default:
        assert(false);
  }
}
...
colorstate current_color;
current_color = red;
doAction(current_color);
current_color = green;
doAction(current_color);
```

3) Usage in Attitude Control and Fault Protection: In spacecraft, typically the Attitude Control System(ACS) will progress through a series of states. For the case of DS1, several example states include an idle state, a detumble state in which the spacecraft attempts to reduce its angular velocities, and a sun-pointing state. In our implementation, we use the state pattern to represent the state of the ACS, following the example design described in Section II-A.2. Using the factory pattern to hide the state implementation details, we provide an AttitudeControlStateFactory which returns concrete implementations of our State interface. The ACS behavior will thus change when we call a method to transition the ACS to a new state, passing in an implementation of the state interface obtained from the state factory.

In addition to using states in the ACS, the DS1 Fault Protection subsystem[12] made extensive use of states. Fault Protection responses were formally specified using StateFlow state-charts, from which the flight code for the responses was

Algorithm 2 Using the states pattern in Java

```
interface ColorState
{
  public void doAction();
}
public class Red implements ColorState
{
   public void doAction()
   {
   }
}
public class Green implements ColorState
{
   public void doAction()
   {
   }
}
...
ColorState red = new Red();
ColorState green = new Green();
ColorState current_color = red;
current_color.doAction();
current_color = green;
current_color.doAction();
```

automatically generated. The semantics of StateFlow state-charts allow the designer to compose states together, and to include code which will be executing upon entering or exiting a particular state. The auto-generated flight code made extensive use of goto statements, as shown in the example in Algorithm 3. The auto-generated code must explicitly call the enter and exit methods.

Algorithm 3 Auto-generated C state change example

```
    if (counter == 1)
    {
    }
    else
    {
      goto jout;
    }
    exit_init_state();
    enter_sun_state();
    return;
jout:
    ...
```

However, our Java approach uses an extended version of the states design pattern. Capturing a subset of the StateFlow semantics, we provide a direct mapping between states and objects. Each state provides for three designated blocks of code to be executed at the appropriate time: on entry to a state, during a stay in a particular state, and on exit from a state. Each state specified by the state-chart corresponds directly to a Java object in our implementation. We require each state to implement the abstract methods onEntry(), during(), and onExit(). Additionally, the logic necessary to change states and to call the appropriate methods on state transitions is handled by an abstract hierarchical state class, separating the response logic from state transition logic. Changing states is as simple as a call to newstate.activateState(oldstate);. The Java version corresponding to the C code in Algorithm 3 is shown in Algorithm 4. In the Java example, the this variable refers to the current object, which is an instance of a state implementation class. We are able to take advantage of polymorphism here to automatically determine the proper entry() and exit() methods to call.

Algorithm 4 Java state change example

```
if(counter == 1)
{
  sun_state.activateState(this);
}
```

B. Facade

Real-time applications will typically contain multiple threads which need to communicate with one another. There are several strategies available for inter-thread communication:

- A thread may directly call a properly synchronized method, gaining access to shared data. Additionally, the developer can take advantage of Java's wait() and notify() methods.
- A more traditional flight software approach of buffering messages in a queue may be used. Once a message is de-queued by the serving thread, it will execute the method specified by the queued message.

In both of the above cases, however, we want to ensure that the communication method itself is not hard-coded into client threads. This decoupling ensures that the inter-thread communication method can be changed, without requiring a full rewrite of all clients.

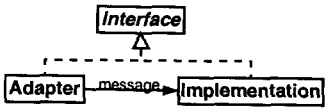
To create this abstraction over the communication, we enlist the facade pattern. According to [2], the facade pattern shields clients from complex subsystem implementations details, and provides a simpler interface for the client. In our case, each inter-thread communication is presented as a method call to an interface. The interface abstracts the desired functionality provided by serving class, which may or may not be in another thread.

For the case of a direct method call, the serving object simply needs to implement the interface, allowing clients to make the method call directly. For the message passing possibilities, two implementations of the interface will exist: the actual object implementation, and the adapter. The adapter class has a reference to the true implementation object. The adapter will then provide implementations of all methods specified in the interface. Within a method implementation, the adapter creates the necessary message object, and then calls the appropriate method on the server thread to enqueue the message. Later, the server dequeues the message and calls the appropriate implementation method. Throughout this indirect call by buffered message passing, the client thread was unaware of the need for the packaging and queuing of the messages. A UML diagram of our use of the facade pattern is shown in Figure 1.

C. Pluggable Components, Factories, and Dynamic Class-Loading

Pluggable components are specified by an interface because implementation details are not visible from classes using the

Fig. 1. Communication facade



components. To hide the implementation class of a particular component, a factory is used instead of directly calling a constructor. The factory is tasked with constructing a particular instance of the pluggable component, and returning the component as an interface. An abstract factory provides a further step of abstraction. Each instance of an abstract factory can construct an instance of a pluggable component in a different fashion, typically instantiating a different implementation class.

We have used dynamic class-loading, in conjunction with abstract factories as a replacement for the C preprocessor. By using this approach, we allow the user of an application to choose at run-time the implementation corresponding to a particular interface. That is, the implementation of a pluggable component can be chosen at run-time. Clients of the interface use an abstract factory to request an instance of an interface. A particular implementation of an interface will have its own concrete factory as a subclass of the abstract class factory. The proper concrete class factory is dynamically loaded at run-time, returning the corresponding implementation of the interface. The advantage of this approach is that we can swap out implementations at run-time. Specifically, this was used to choose between our desktop scheduler implementation and the RTSJ scheduler implementation at run-time.

In the long term, dynamic class-loading has much greater potential for spacecraft missions. Current practice requires reloading a binary image of the executable to a spacecraft, followed by a reboot. We envision that Java's dynamic class-loading facilities could be used to provide additional functionality to a spacecraft by uplinking new class files to a running system, without requiring a reboot. This capability is outside of our current scope, but would be an interesting avenue for further research.

III. Real-Time Layer

A. Scheduler

The nature of flight software requires that certain threads execute at certain times, and the times that the threads execute depends on the type of work being done. For instance, control loops run periodically while watchdogs run once at some time to signal the system of a potential problem. Defining the temporal boundaries and contraints for these threads is independent of the scheduling algorithm being used, but communicating these constraints to the scheduler is dependent on the scheduling algorithm and its implementation. We chose to apply the pluggable technology approach to our scheduler so that we can use whatever scheduling algorithm is available to us. Currently, we provide several varieties of scheduling requests:

- A one-shot timer. The scheduler will run a block of code after at a specific time.
- Periodic behavior: The scheduler will then run the block of code at the client specified rate.
- Standard: The scheduler will run the block of code when possible.

All of these behaviors can also specify a deadline that when crossed will cause a secondary block of code to be executed. Additionally we provide facilities for specifying a maximum percentage of CPU usage by a particular thread. The scheduler to be used is selected at run-time and instantiated through the use of dynamic class loading and factories.

Since the RTSJ reference implementation does not currently support debugging, our choice of pluggable technology allowed us to use the desktop for debugging. When running within an RTSJ-compliant virtual machine, our scheduler interface simply delegates out to the underlying RTSJ implementation. However, when running on a standard desktop Java virtual machine, the scheduler component uses our own implementation, written only using standard Java features. We emulate, as best as possible, the real-time scheduling features on a standard Java platform. Clients may chose between the RTSJ scheduler implementation and the desktop scheduler implementation at run-time.

Thus, real-time client code can be run on a standard Java virtual machine, allowing the code to be debugged in a graphical development environment on a desktop PC at the cost of decreased real-time performance. Once non-real-time issues have been debugged on a standard Java VM, the real-time issues can be debugged in isolation on an RTSJ-compliant virtual machine.

B. Memory Areas

With the addition of the RTSJ's scheduling and memory management features, come new failure modes and programming pitfalls. The developer must consciously avoid violating memory area rules, and must ensure that no memory leaks occur. We present a series of guidelines for using the RTSJ memory management features. We provide a set of recommendations for memory allocation, showing scenarios that take advantage of memory areas provided by RTSJ. In addition, restrictions are placed on memory allocation scenarios that are particularly error-prone.

1) Immortal memory : Immortal memory is a new allocation scheme provided by the RTSJ. Once an object is allocated in Immortal memory, it is never freed. The advantage of this approach is that objects allocated in immortal memory have no need for interaction with the garbage collector. The disadvantage is that memory leaks are now possible. We recommend that allocations to immortal memory be performed in static initializers. We also require that object which are running in immortal memory only allocate in their constructors. With these restrictions in place, memory leaks can be avoided. However, this also places severe restrictions on which classes may be used. To use a JDK class in while running in immortal memory, one must inspect the source code to ensure that allocations are only performed in the constructor.

2) Scoped memory usage: Scoped memory provides a means to dynamically allocate and free memory without using the garbage collector. Object allocated within a scope persist for the lifetime of the scope. Once the number of threads within a scope reaches zero, all objects allocated within the scope are destroyed. Additionally, scopes may be nested. The advantage for application programmers is that a large number of objects can be allocated and freed at once, without creating excess work for the garbage collector. One can think of scopes as a generalization of the C stack with the exception that the objects are finalized in the case of scopes.

A particular scoped memory region is represented by a scoped memory object, which itself must be allocated in a memory region. If one allocated a scoped region on the heap, the scoped memory object itself would be subject to interference from the garbage collector. For our application, all threads are created at application startup time. In this case, we can allocate scopes in immortal memory, and have examined the possibility of creating separate scopes on a per-thread basis. This scope allocation paradigm is quite similar to having one C stack per thread. The thread would then enter its own scoped memory, perform allocations, and then leave the memory area, automatically destroying the scope-allocated data. The size of the scope can be determined by profiling the memory usage characteristics of a particular thread, taking into consideration the requirements of the application and the available hardware resources.

With the RTSJ, it is also possible to nest scopes. We now provide an example of how to exploit this feature. Suppose we are inside a scoped memory region. Assume we have a for loop, which allocates 1K per iteration, and the loop runs 1K times. Further assume that the scoped memory region is entered once before the loop. The scoped memory area must be 1 meg in size. However, if we enter a nested scope once per iteration of the loop, all of the memory allocated will be freed at the end of each loop iteration. Thus, we can reduce the scoped memory size to 1K. So we see that using nested scopes within loops allows us to reduce the memory usage. Further examples of the use of scoped memory can be found in [5].

The difficulty with scopes is their limited lifetimes. We envision entering and leaving a scoped memory region once for every iteration of our control loop. However, some data will need to persist beyond the lifetime of the scope, so we must provide mechanisms for copying data out of a scoped memory region. To facilitate this, we recommend providing memory areas as parameters to factories. These factories could then be used to copy and construct objects in arbitrary memory areas.

3) No Heap Real-time Threads: As a new thread class, the RTSJ introduces NoHeapRealTimeThreads(NHRTT). NHRTTs cannot access object allocated on the heap, thus avoiding interactions with the Garbage collector. There are several choices available for an application architecture using NHRTTs: either all threads will be NHRTTs, or a mix of NHRTTS and real-time threads will be permitted. Designers who choose to use all NHRTTs will be working in a restrictive environment: all data which cannot be discarded upon leaving a scope must be allocated in immortal memory. In this environment with only manual memory management, some of the benefits of Java disappear, and another implementation language may be more suitable.

We now consider the case of an application containing both NHRTTs and real-time threads. To maintain a clean architecture, the application architect should attempt to make inter-thread communication as transparent as possible. However, moving data between a NHRTT and a real-time thread requires special handling. Data moving between the two threads must be transferred using a wait-free queue provided by the RTSJ. But due to the requirements of NHRTTs, objects passed through the queue will typically have to be allocated in immortal memory. However, to avoid memory leaks these queued objects must be managed. We have pondered using two possibilities, both of which we find unsatisfactory.

The first possibility is to implement a pool of objects which can be used for communicating with immortal memory. This has several disadvantages. The objects allocated within the pool cannot be immutable, decreasing the application's maintainability. Additionally, a pool can only contain objects of a single class. As a second possibility, consider replacing the pool with a pre-allocated block of bytes in immortal memory. Objects being sent would then be serialized to this common area, allowing objects of different classes share the same memory block. Unfortunately this introduces an unnecessary serialization cost at run-time. The serialization approach essentially reduces to a Fortran common block, and forces classes to provide otherwise unnecessary serialization logic. In summary, we have not been able to determine a clean way of moving objects from a NHRTT to a real-time thread without major changes to the application's architecture, and we consider this to be an open problem for the RTSJ community.

IV. Units

A. Problems with past practice

In current flight software projects, the measurement units are not explicitly part of the software. Perhaps measurement units are designated in an external document or in code comments, but there are no automated checks at either compile-time or at run-time to ensure that unit arithmetic is correct. For example, multiplying a velocity by a time should result in a distance. But since values are only represented as doubles, nothing prevents the developer from incorrectly treating the result of such an operation as a force, for example. We have already seen the disastrous consequences of incorrect units in the Mars Climate Orbiter mission.

B. Our approach

To remedy this problem, we advocate making measurement units an integral part of the application code. Our package provides compile-time checking of measurement units. We provide interfaces for physical units, such as forces, distances, and times, and allow scalars, matrices, and tensors of values with physical units. With measurement units explicitly part of our code, we gain a number of advantages. Since measurement units are checked at compile-time, bugs are detected sooner,

with a lower cost to repair them. Specifically, by using the units framework in our development, the detumble control loop was debugged in only 13 iterations. Because we knew that the measurement units were correct, pinpointing the actual cause of the errors became simpler.

In implementing our units framework, we have made use of COTS class libraries. However, since units are pluggable components, alternative implementations are possible. For performing matrix and vector operations, we take advantage of the classes providing such functionality in Java3d. Additionally, for unit representation, we make use of the Jade library[4]. The admitted disadvantage of using Java for this situation is the lack of operator overloading, since the syntax for performing arithmetic does become quite verbose.

V. Conclusion

A. Summary

We have developed a pure Java prototype attitude control system, capable of performing a detumble maneuver in real-time, along with a pure Java fault protection subsystem. In developing this prototype, we have shown how to apply best practices in OO design. We have demonstrated how to apply design patterns to a realistic flight software development effort. Specifically, we have demonstrated the applicability of pluggable components, factories, states, and facades. The measurement units facility allows the checking of unit correctness at compile-time. We have explored the features of the RTSJ, discussing the usage of memory areas. We have created a pluggable scheduler component, enabled debugging on a standard Java platform. Taken together, our work has exploited Java and RTSJ features to demonstrate how to create more maintainable flight code.

VI. Acknowledgments

This work was supported in part by the Center for Space Mission Information and Software Systems(CSMISS) at the Jet Propulsion Laboratory, and by the Ames Research Center.

References

[1] G. Bollella , J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull, *The Real-Time Specification for Java,* Addison-Wesley, 2000.
[2] J. Cooper, *The Design Patterns Java Companion,* Addision-Wesley, 1998.
[3] A. Corsaro and D.C. Schmidt. "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems." Technical Report 2002-001, University of California, Irvine, 2002.
[4] J.M. Dautelle, "JADE Java Addition to Default Environment", http://jade.dautelle.com/, 2002.
[5] P. Dibble, *Real-Time Java Platform Programming*, Prentice Hall, 2002.
[6] "Eclipse.org" , http://www.eclipse.org/, 2003.
[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
[8] "Headway Software", http://www.headwaysoft.com/, 2003.
[9] NIST Special Publication 500-243, *Requirements for Real-time Extensions for the Java Platform: Report from the Requirements Group for Real-time Extensions for the Java Platform*, 1999.
[10] OVM/Consortium, "OVM: An Open RTSJ Compliant JVM." http://www.ovmj.org, 2003.
[11] M. Rinard et al., "FLEX Compiler Infrastructure", http://www.flex-compiler.lcs.mit.edu/, 2003.
[12] N. Rouquette, T. Neilson, and G. Chen, "The 13th Technology of DS1." *Proceedings of IEEE Aerospace Conference*, 1999.
[13] "Sitraka JProbe", http://www.sitraka.com/software/jprobe/, 2003.