Design and Implementation of Java Technology Flight-like Software

Ed Benowitz and Al Niessner
Technical Staff
Jet Propulsion Laboratory
Learn how Java and Design Patterns can be used to develop embedded, real-time software.

- Aim to pass along some of our experiences in applying Java to developing real-time software
- And we'll also discuss how we applied design patterns in this environment
Speaker’s Qualifications

- Ed Benowitz is an embedded, real-time developer at the Jet Propulsion Laboratory
  - Previous incarnations:
    - Java sustaining engineer for Sun Microsystems
    - Embedded developer for Raytheon
  - Over 6 years of Java development experience
  - MS from UCLA
• Al Niessner is an embedded, real-time developer at the Jet Propulsion Laboratory
  — Previous Incarnations:
    • Embedded real-time developer at Applied Research Laboratory at the Pennsylvania State University doing DSP work for torpedoes (Ada and C40 assembly)
    • Real-time developer at Raytheon doing military communications (Ada, C, and assembly)
  — Total of 10 years experience of producing successful, embedded, real-time products
  — MS from Penn State
Presentation Agenda

- Motivation
- Approach
- Real-time programming in Java
- Use of Design Patterns
- Demonstration

Make sure to stay till the end. We'll be demonstrating our work, showing a simulated spacecraft moving in 3d. 100% of the simulated spacecraft's code is implemented in Java, including its real-time control loop.
Motivation

- Problem: Lack of maintainability
  - Weak checking
  - Error-prone switch statements
  - Low-level concurrency primitives
  - Measurement units not explicit
  - Can't express pluggable components
  - Completely manual memory management

- Difficult to maintain flight software written
  - Traditionally in C
- No strong type checking, pointer checking, array-bounds checking
- Long and error prone switch statements
  - Forget a break
  - Adding a new case involves going through many functions
  - Run-time assertion check in the default case
- Concurrency is not part of the language, concurrency issues is difficult to debug and reproduce
- Measurement unit errors resulted in the loss of the Mars Climate Orbiter mission
- C++ has multiple inheritance problems and the friends notion, along with the lack of interfaces, makes it difficult to express pluggable components. Pluggable components expose only their interface, not their implementation
- Memory management
To investigate Java's application for flight software, we've constructed pure Java prototypes.

To ensure a realistic scenario, we've used a Deep Space 1 mission design for our requirements which flew in 1998.

The real-time Java specification provides the necessary functionality. It is possible to implement a real-time system in pure Java without resorting to JNI. JNI can and should be avoided, whenever possible.

We've focused on the attitude control subsystem, performing a detumble. The control system must stop a spacecraft from spinning. Thus, attitude control is a good example of a real-time control loop.

Fault protection demonstrates some autonomous behavior as a way of responding to on-board faults.
Approach

- Exploit features of Java
- Design Patterns
- OO best practices
- Emphasize maintainability
- Profile and optimize methodically

To improve the maintainability, we want to use Java in conjunction with best OO practices, including Design patterns.

Specifically, the state, facade, and factory patterns make frequent appearances.

- Use pluggable components
- Optimize according to profiling data, not assumptions
  - Use memory profile data to select RTSJ memory management features
Real-Time Specification for Java

- Memory management
- Scheduling
- Timing
- Physical Memory
- Asynchrony
- Priority inversion avoidance
- http://www.rtj.org/

- Ways to allocate memory outside of the garbage collector
- With certain coding restrictions can guarantee you will not be preempted by the garbage collector
- Scheduler can create periodic threads for example
  - Flexible scheduling parameters
- High resolution timing
- Access to physical memory,
  - But can specify a restricted range
- Handle asynchronous events
- Could potentially write Java device drivers
Real-Time Layer

- Abstraction over RTSJ
  - Allows debugging on the desktop
  - Provides
    - Scheduling
    - Timing
    - Memory access

- Current RTSJ tools do not support debugging or profiling
- Want to be able to use powerful Java IDEs.

- Provides real-time services as pluggable components
- Emulation of real-time features provided on a desktop platform
- Allows debugging using the power of
  - COTS Integrated Development Environments (IDE's)
  - COTS graphical debuggers
- Examine logical errors in a modern debugger
- Deal with real-time issues in isolation on the real-time Java VM
- After logical errors are debugged on a standard Java
- On a real-time VM, a debugger would interfere, so
  nothing is lost
Periodic threads to deal with Runnables needed to execute within the RTI, plus an offset.

One shot timers are typically used for timeouts, ensuring progress even if a device does not respond within the timeout, for example.

Deadlines can be specified.

Fair performing, but adequate desktop implementation used for debugging.

Real-time version delegates to RTSJ.
Memory Areas

- Heap Allocation
  - Standard in Java
  - Automatic management
  - Unpredictable garbage collector

- Standard Java features fully automatic memory management
- Every object is heap-allocated
- Unpredictable garbage collection pauses
- Garbage collector may have higher priority than a critical thread with real-time requirements
Immortal memory is a new allocation scheme in the RTSJ

- Allows you to allocate object permanently
- Avoid interaction with GC, because there's no need to GC memory which cannot be freed
- To avoid leaks we propose
  - Doing immortal allocations at application initialization time, perhaps within static initializers
  - Objects running in immortal memory should only allocate within their constructors.

Thanks to Peter Dibble's Real Time Java Platform Programming
Memory Areas

- Scoped Memory
  - Allocate a block of scoped memory
  - Free entire scope at once
  - Generalization of a C stack
  - Restrictions

- Immortal only allows permanent allocation; scope is a memory area which allows dynamic memory allocation without the garbage collector.
- A scope of a given size is first allocated
- When a thread "enters" a scope, its allocations take place within that scope.
- When all threads have exited a scope, all the memory within the scope is reclaimed.
- Similar to a C stack or using alloca, which is cleared at the end of a function call
- Restrictive rules on who may use
- Only local variables and certain available scopes can reference memory allocated on the scope
- May be necessary to copy data from scoped to another memory area, before the scope is exited
Use of Scoped Memory

Where to allocated the scoped memory itself?

For the case where threads are not dynamically created during the application run

- Can profile each thread to see how much memory it will use per RTI
- Allocate a scope of the appropriate size for each thread
- Create the scope in immortal memory
- Enter and leave the scope per RTI
- All objects allocated on a per-RTI basis can be scope allocated, without creating additional work for the garbage collector
- These objects are freed automatically each RTI as the thread leaves the scope

For threads which are dynamically created, may need to allocate scope within the heap
Nested Scopes

- Entering a scope multiple times
  ```java
  // enter scope
  for(i=0; i < 1000; i++)
    // allocate
  
  // enter scope
  for(i=0; i < 1000; i++)
    // enter scope
    // allocate
  ```

- RTSJ also provides the ability to nest scopes. We now show an example of when to use this feature
- Assume nobody else is using the scope. Every time we enter, allocate, and then exit from the scope, the memory allocated is freed
- In the first example, we must allocate a scoped memory area to be big enough to accommodate all 1000 elements. Then, after the for loop has completed, all can be freed at once.
- However in the second example, a nested scope is entered. The nested scope is entered and exited once per iteration of the loop
- This means that memory is freed after each loop iteration
- So the size of the scope can be reduced to the amount of memory allocated in one iteration
States Design Pattern

- States are Objects
- Eliminate long, unreadable switch statements
- Stronger type checking
- Reduce code complexity
enum colorstate { red, green);

void doAction(colorstate current_color){
    switch(current_color){
        case red: break;
        case green: break;
        default: assert(false);
    }
}

colorstate current_color = red;
doAction(current_color);
doAction(current_color);
current_color = green;
doAction(current_color);

- This shows the pitfalls of representing state in the traditional C state paradigm
- Error prone switch statement
  - Runtime assertion check
  - If we had to add a yellow state, would have to hunt for all uses of colorstate
  - No guarantee that all enumerated choices were presented in the switch
This shows the improved Java version of the same ColorState, implemented using the states design pattern.

By using dynamic dispatch, we've eliminated the need for switch statements.

Runtime checking isn't needed, since each state is forced to provide the required interface.

Easy to add a new state.

All the code for a particular state is together in an implementation class.

States are objects.
• In legacy code, a check against an idle flag was used to guard function calls.

• Instead, we have an idle state, eliminate the need for manual checking

• A spacecraft will typically proceed to detumble to stop spinning, which we will demonstrate today. At some point after a detumble the spacecraft would then acquire the sun, using a different control loop. This is represented by a different attitude control state
Facade Pattern

- Use to hide inter-thread communication

```
    Interface
     |
     V
Adapter message Implementation
```

- Hide inter-thread communication behind a facade
- Two classes derive from the same interface
- The adapter class implements the message packaging if necessary
- Use message passing if the implementation class is not properly synchronized
- Or can bypass the adapter if the class is thread safe, passing an instance of the implementation
Abstract Factories

- Pluggable Components
  - Show the interface
  - Hide the implementation
- How to instantiate different implementations?

- Pluggable components
- Only the interface is exposed, not the implementation
- This allows one implementation to be substituted for another
- In Java we have the interface concept to help express pluggable components.
- But we must make sure to hide the implementation class, even during construction time
- This can be accomplished by having a factory class, which is responsible for returning an instance of an interface.
- Factories are used throughout the Java API
  - For example, JAXP
- But how can we use factories to create different implementations of the same interface?
Abstract Factories

- Abstract Factory
  - Each factory creates a different implementation of the interface
  - Factories implement a common interface
- Our extension
  - Allow AbstractFactories to be chosen at run-time

- Abstract factories
  - Implement a common interface typically
  - Each factory provides a different way of creating an instance of an interface
- We would like to allow the particular interface implementation to be chosen at run-time
  - Increased flexibility
  - No need to recompile between configuration changes
  - Use as a replacement for the C preprocessor
We use dynamic class loading, to choose abstract factory implementations are runtime, thus allowing the components which implement a particular interface to be chosen at runtime.

This enables two separate implementations of the scheduler, desktop and real-time. We can switch between the two implementations at run-time without the need for a recompile.

Call static method getInstance on the AbstractFactory. Name is resolved via ResourceNames, which queries run-time properties for the abstractfactory class to instantiate. Resource names then dynamically loads a concrete factory. AbstractSensorFactory then returns the ConcreteFactory to a client who may then create sensor objects using the factory.
Checking measurement units at compile-time
Logging and Telemetry

To improve maintainability, measurement units are checked at compile time.
- Ensure meters / second gives a velocity.
- Wraps around JCP's units framework, java3d for vecmath
- Syntax becomes difficult, need operator overloading

Logging
- An extension of JDK1.4, apache logging
- Can be used for telemetry as well
Demo
Summary

- NASA is creating flight-like Java software prototypes today
- Apply Design Patterns and OO techniques to improve maintainability
RTSJ is propelling Java into space
Q&A