

Java for Flight Software
by Eddie Benowitz

Abstract:

We discuss how we apply several design patterns to real-time Java. Both design patterns and real-time Java are part of the open literature. We will not discuss any control or scheduling algorithms. We discuss how to use existing Java features to check measurement units. We do not discuss past or existing flight software. We will discuss our technology infusion approach for real-time Java, including the use of standard Java language features, to provide more maintainable flight-like software.

JavaTM for Flight Software

Al Niessner, Ed Benowitz

Autonomy and Control Section 345

Jet Propulsion Lab, California Institute of Technology

Supervisor: Abdullah Aljabri

Funded by CSMISS at the Jet Propulsion Lab and Ames Research Center

Agenda



-
- Motivation
 - Advantages of Java
 - Real-time Java
 - Architecture
 - Software
 - ↗ Real-time layer
 - ↪ Scheduling
 - ↪ Memory management
 - ↗ Attitude Control
 - ↗ Units
 - ↗ Fault Protection
 - ↗ Other infrastructure
 - ↪ Logging
 - ↪ Dynamic class loading
 - Accomplishments
 - Future Work

High maintenance = High cost

- Current flight software is difficult to maintain
 - Lack of strong type-checking and parameter-checking
 - Lack of pointer and array safety allows
 - ☞ Easy corruption of memory
 - ☞ Silent failures
 - Cannot express pluggable components
 - ☞ C
 - Global variables
 - No encapsulation
 - ☞ C++
 - Multiple inheritance problems exist
 - “friend” breaks encapsulation
 - Completely manual memory management
 - Units not explicit
 - ☞ Disastrous consequences
 - No use of modern Integrated Development Environments and debuggers
 - Low-level concurrency primitives
 - ☞ Concurrency not part of the language
 - ☞ Concurrency issues are difficult to reproduce
 - Error-prone switch statements and preprocessor directives
 - Globally shared namespace

Task Summary



Objective:

- Perform technology infusion of real-time Java for spacecraft control software
 - Develop a flight-like Java software prototype
 - Real mission design (DS1)
 - Flight-like platform (PPC)
 - Explore and promote the advantages of Java

Approach:

- Prototype the Attitude Control System performing a detumble maneuver using:
 - Pure Java system
 - Existing spacecraft simulation
 - Closed-loop control
- Demonstrate the advantages of Java by using:
 - Best practices in Object-Oriented development
 - Design Patterns

Advantages of Java (1 of 3)



-
- Maintainability
 - Java eliminates the most common programming errors with
 - ☞ Stronger type-checking at compile-time and run-time
 - ☞ Automatic memory management
 - ☞ Array-bounds checking
 - ☞ Variable-initialization checking
 - Java provides single inheritance / multiple interface inheritance
 - Java has strong encapsulation
 - Extensibility
 - Java allows dynamic class loading
 - Java facilitates an Object-Oriented approach

Advantages of Java (2 of 3)



-
- Readability
 - ↗ Interfaces
 - ↗ Exceptions
 - ↗ Packages
 - ↗ Language support for multiple threads and synchronization
 - Middleware
 - ↗ Large standard class library
 - People
 - ↗ Huge community
 - ↗ Short ramp-up time for developers

Advantages of Java (3 of 3)



Source: National Institute of Standards and Technology (NIST)

- Java's higher level of abstraction allows for increased programmer productivity.
- Java is easier to master than C++.
- Java is secure by keeping software components (including the VM itself) protected from one another.
- Java is highly dynamic, supporting object/thread creation at run time.
- Java supports component integration and reuse.
- The Java language and platform support application portability.
- The Java technologies support distributed applications.

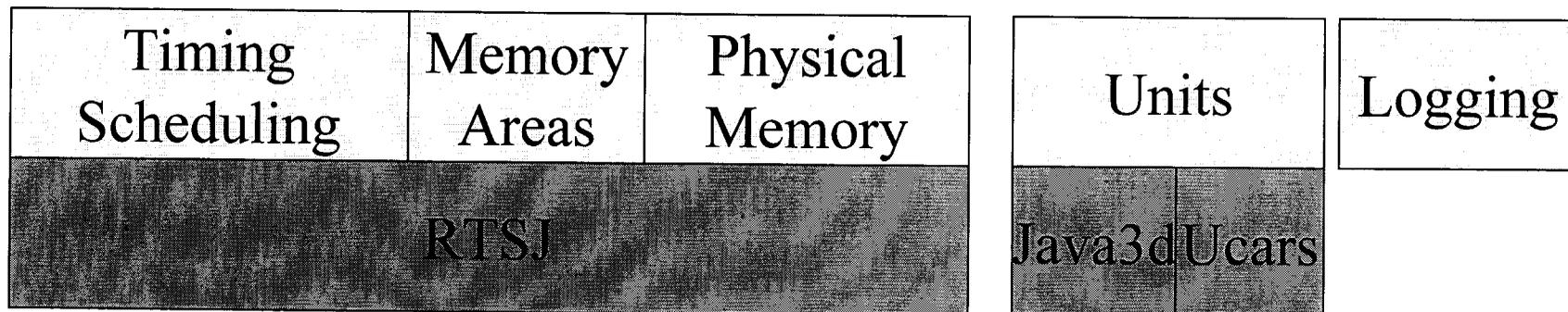
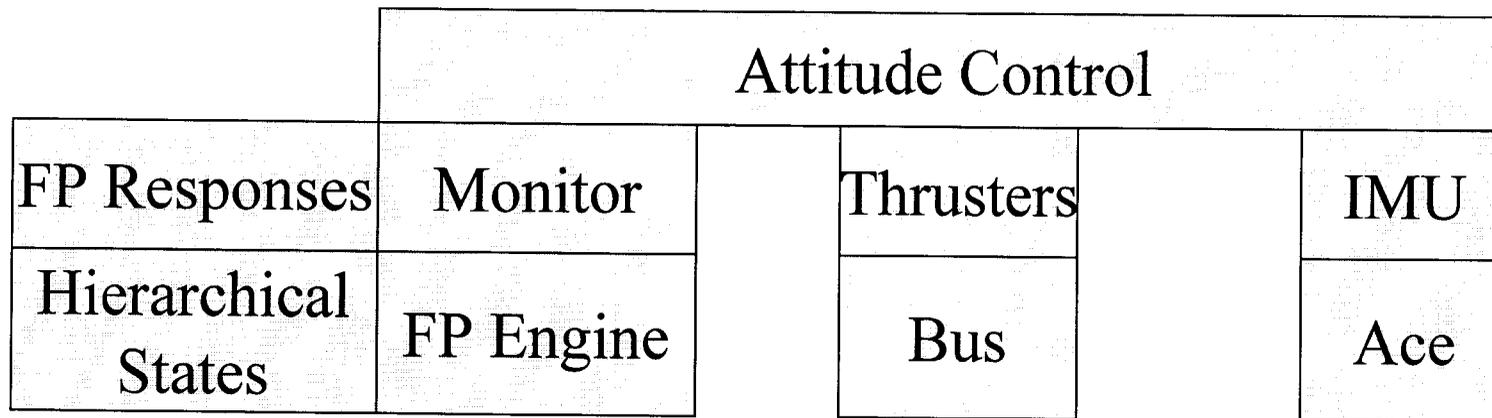
Real-Time Spec for Java(RTSJ) **JPL**

- Scheduling
 - ↗ Periodic threads
 - ↗ One shot timers
 - ↗ Automatic priority inversion avoidance
- Handle signals
- High-resolution timing interface
- Runs on an Real-time Operating System (RTOS)

Real-Time Spec for Java(RTSJ) **JPL**

- Access to Physical Memory
 - ↗ Restricted to a specified region
- Memory Allocation
 - ↗ Guaranteed linear-time allocation
 - ↗ Scoped memory
 - More general than stack allocation
 - ↗ Immortal Memory
 - ↗ Avoid garbage collection pauses
 - At expense of not touching the heap
 - Less straightforward interaction with heap-using threads

Architecture



Key Architectural Themes



-
- Favor maintainability by
 - ↗ Using pluggable technology
 - ↗ Taking full advantage of Java features
 - ↗ Making extensive use of Design Patterns
 - ↗ Reusing COTS technology where appropriate
 - Optimize the most critical areas methodically

-
- 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

-
- One-shot
 - ↗ Run me after 30 seconds have passed
 - Periodic behavior
 - ↗ Run me once every half second; I will use at most 25% of the processor
 - Deadline
 - ↗ Run me using 30% of the CPU; I promise to finish in 2 seconds. Report an error if I miss my deadline
 - Implementations for both desktop and real-time Java

Memory Areas



-
- Standard Java uses heap allocation
 - ↗ Automatic memory management
 - ↗ Garbage Collector can have a higher priority than application threads
 - Immortal memory
 - ↗ Once memory is allocated, it is never freed
 - ↗ Never needs to be garbage collected
 - Scoped memory
 - ↗ Eliminates excessive garbage generation
 - ↗ Many objects can be cleaned up at once
 - Powerful, but potentially tricky
 - Documented a set of guidelines for memory allocation
 - Pluggable
 - ↗ Can run applications on a standard desktop platform

Attitude Control Approach

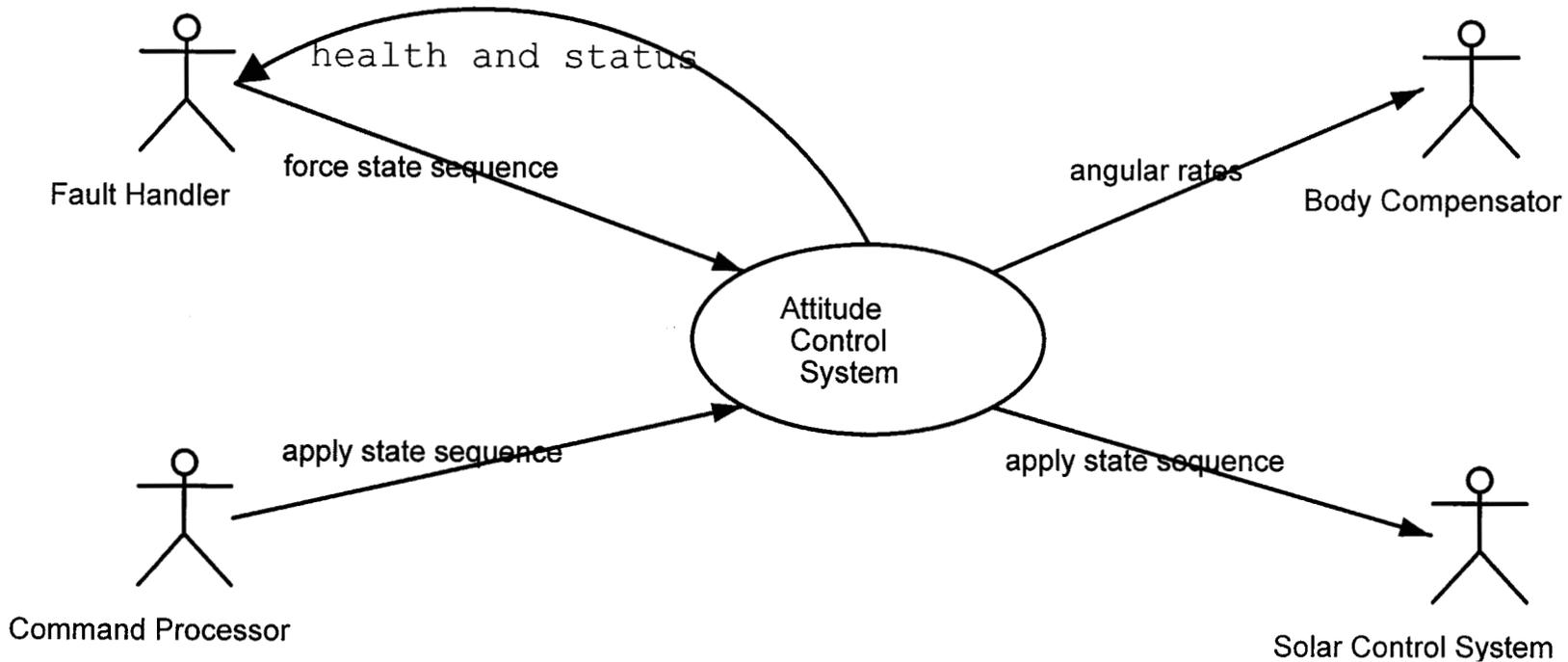


-
- Provide direct mapping between control loop and Objects
 - Code is more accessible and readable to the control engineer
 - Use existing JPL control terminology
 - Employ “State” Design Pattern
 - Treat sensors and compensators are pluggable components
 - Implementation leverages units framework which
 - Reduces debugging time
 - Improves correctness

Core Systems: ACS and FP



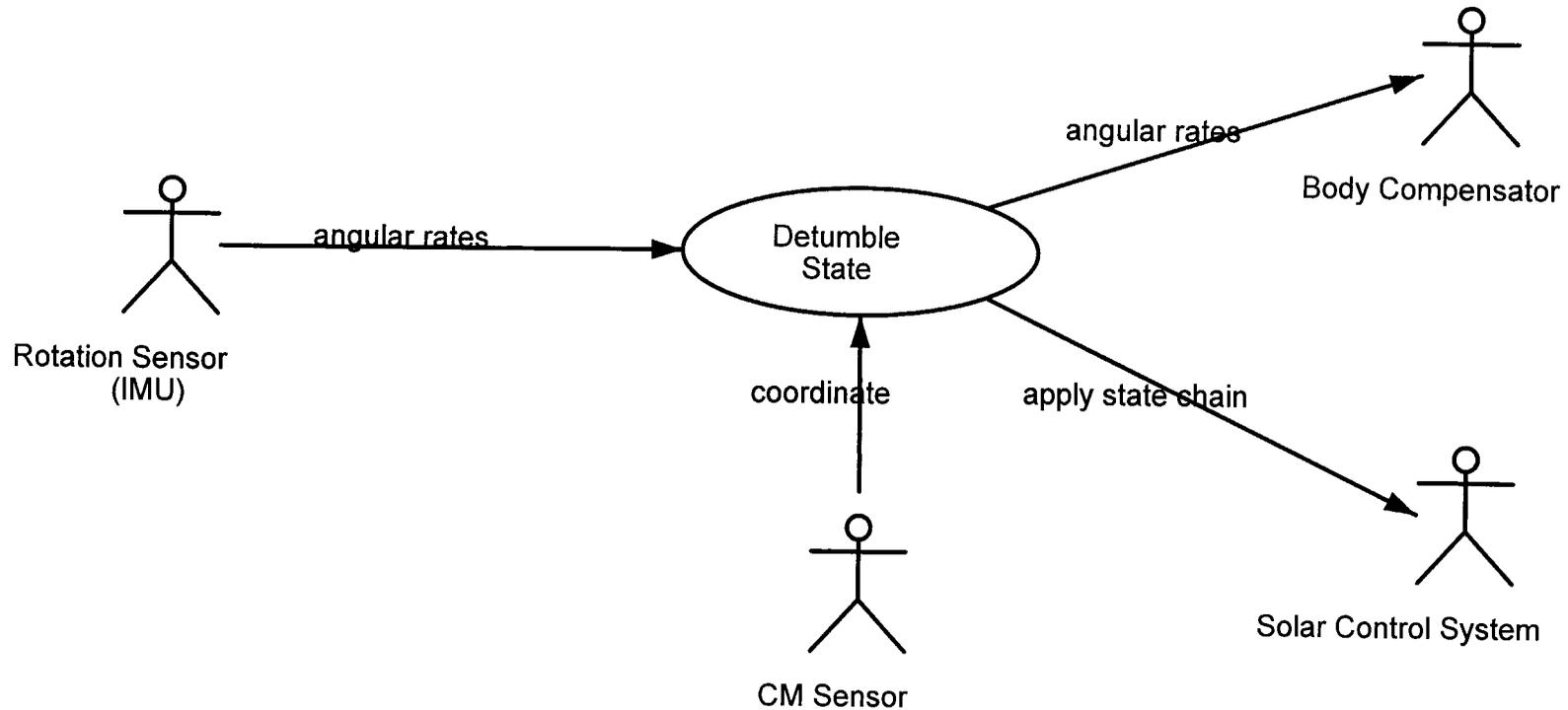
Top Level Use Case



Detumble Use Case



Detumble Use Case

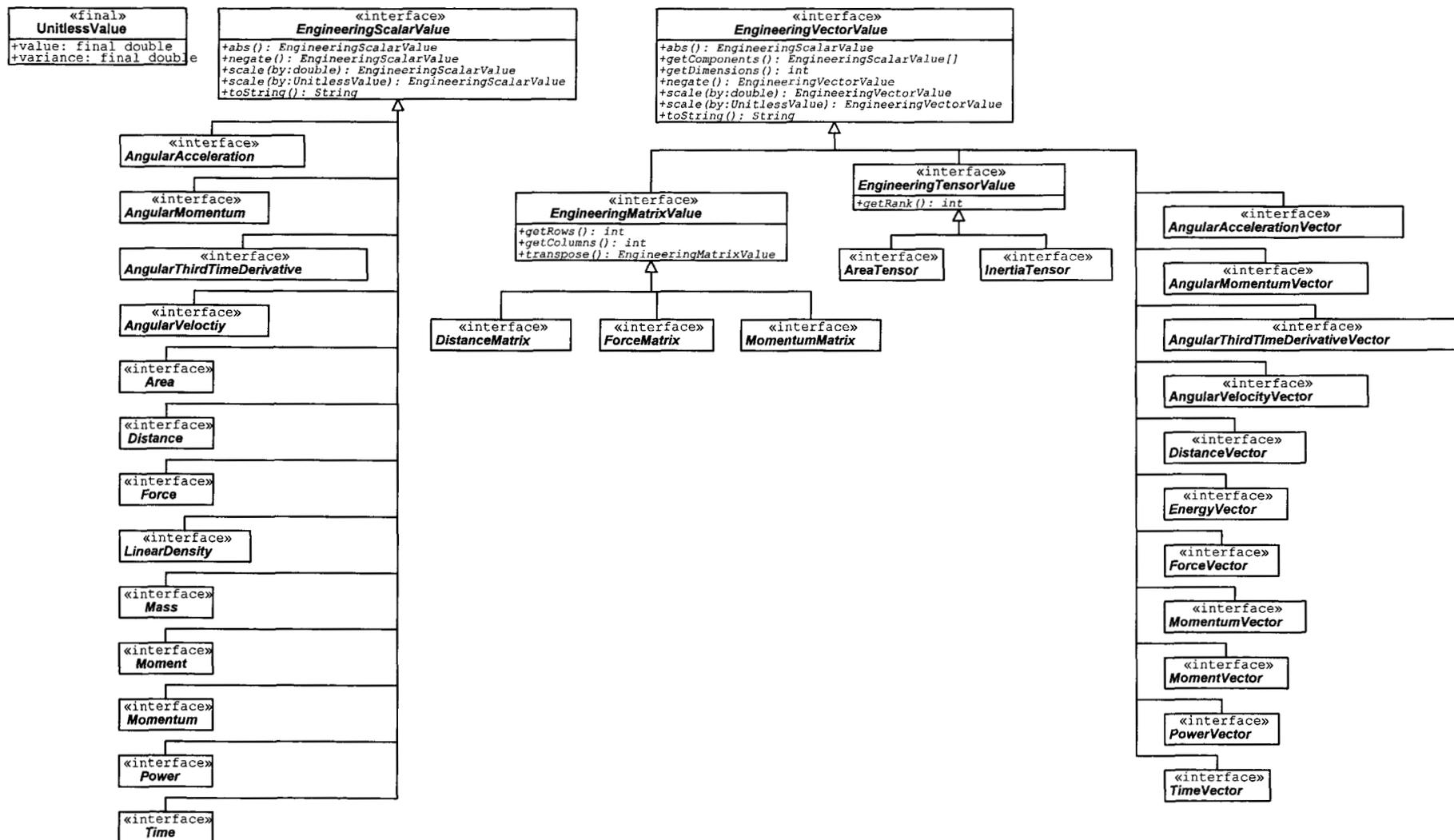


Units Approach

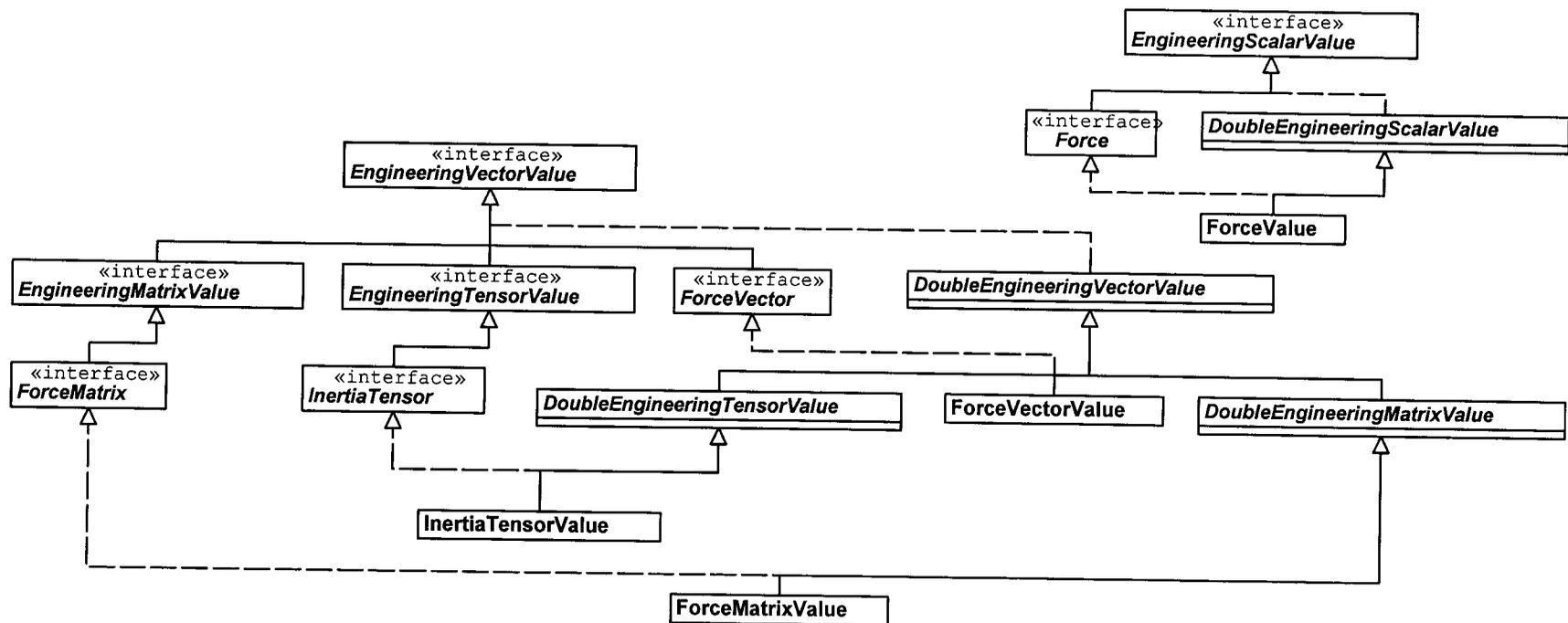


-
- Past practice
 - ↗ Measurement units were not explicitly declared in code
 - ↗ Had disastrous consequences
 - Checking measurement units explicitly at compile time
 - ↗ Ensures proper unit arithmetic
 - ↗ Finds bugs earlier in the development cycle
 - ↗ Implemented with units as Interfaces
 - ↗ Reused COTS frameworks underneath
 - ↻ UCARS units
 - ↻ Vector/Matrix/Quaternion from Java3d

Units



Units Implementation

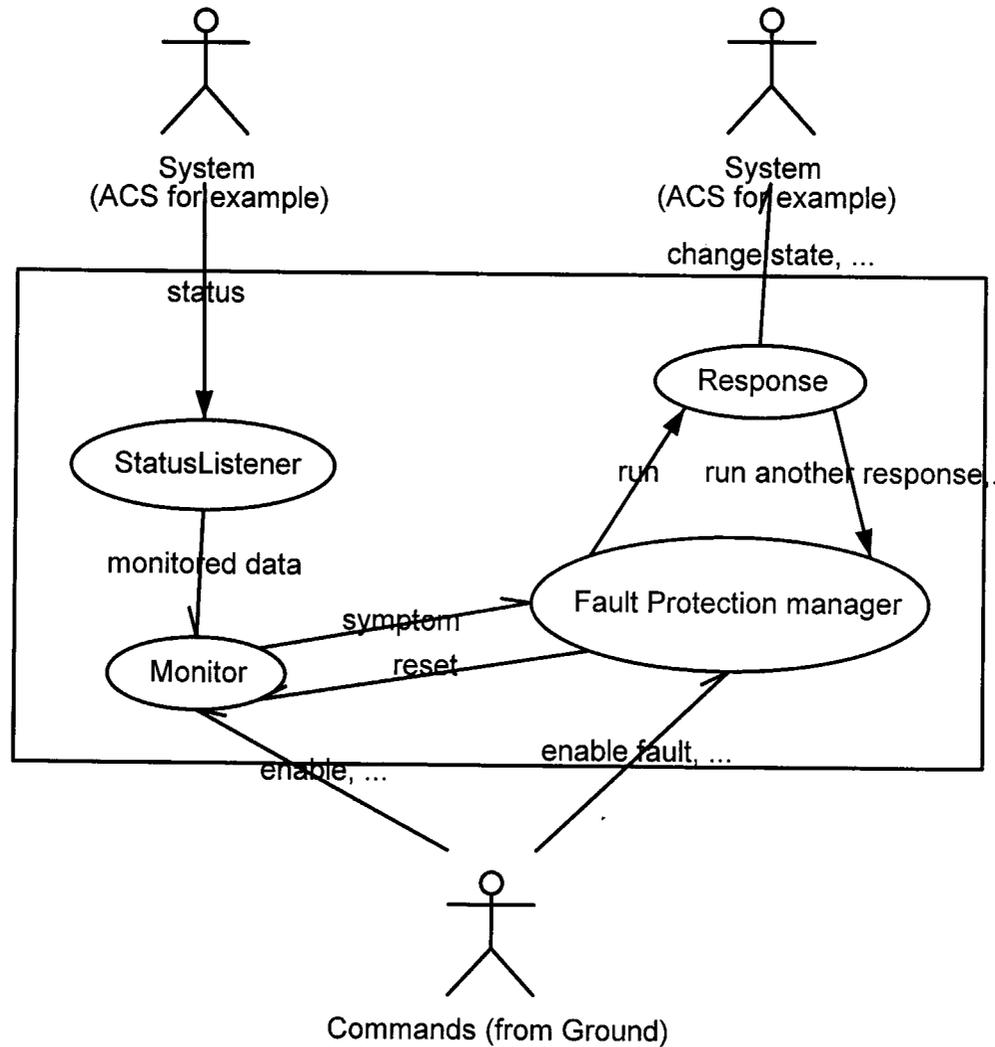


Fault Protection Approach



-
- Use Deep Impact's Fault Protection Engine as the basis
 - Provide direct mapping between state charts and objects
 - Hide communication with other threads via the Façade pattern
 - Provide re-usable Java classes
 - ↗ Fault Protection Engine
 - ↗ Threshold component
 - ↗ Hierarchical states
 - ↗ Composite health and status

Fault Protection Use Case



Fault Protection Terminology



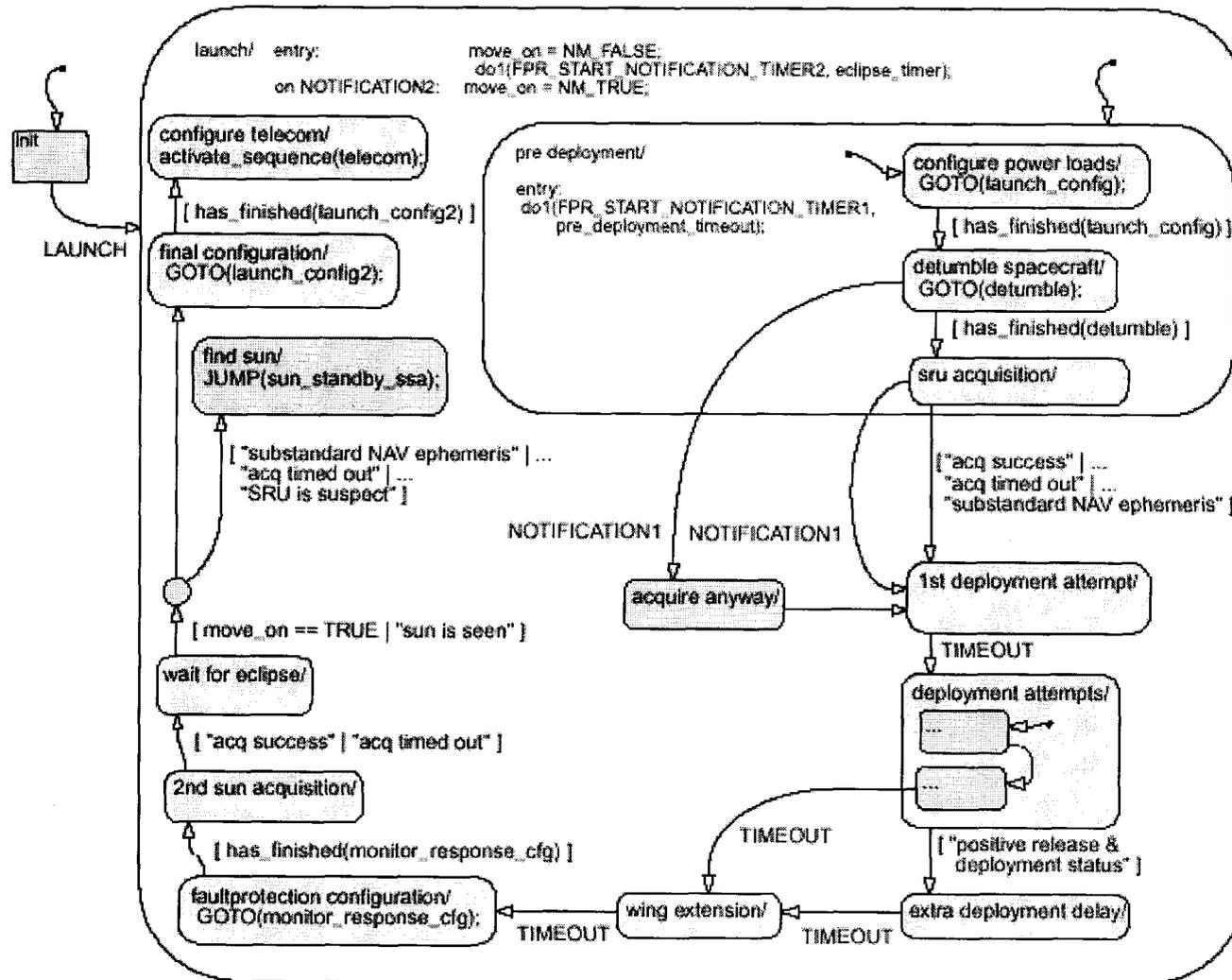
-
- Symptom
 - ↗ Signifies that incorrect behavior is noticed
 - Monitor
 - ↗ After threshold is triggered, notifies a symptom
 - ↗ Can be called by health/status listener
 - Threshold
 - ↗ Used by monitor to watch values over time
 - Fault
 - ↗ The underlying problem which caused a symptom
 - Response
 - ↗ How we deal with a fault
 - ↗ Based on state chart
 - Manager
 - ↗ Provides a mapping between symptoms, faults, responses
 - ↗ Serves as a container
 - Engine
 - ↗ Derived from C++ Deep Impact engine, schedules responses

Fault Protection Responses



-
- Response
 - ↗ Contains variables used by the state machine.
 - ↗ Is responsible for creation of State objects.
 - ↗ Delegates to states
 - States
 - ↗ Are a fundamental Design Patterns observation:
 - States are Objects
 - ↗ Eliminate long, unreadable switch statements
 - Gain stronger type checking
 - Reduce code complexity
 - ↗ Use abstract methods `onEntry()`, `onExit()`, `during()`, providing
 - Improved Readability
 - Direct correspondence with the state chart

Example Response Diagram



From the Mathworks web page

-
- General Framework for
 - ↗ Diagnostic printing
 - ↗ Event reporting
 - ↗ Telemetry reporting
 - Based on design of
 - ↗ Java 1.4 logging facility
 - ↗ Apache's logging framework
 - Pluggable
 - ↗ Logging formats
 - ↗ Filters

Dynamic Class Loading



-
- Dynamic class loading allows
 - Loading new code into a system at run-time
 - Instantiating an object of a given class
 - by specifying the class name as a string at run-time
 - For maintainability, dynamic class loading
 - Replaces the C preprocessor (conditional compilation)
 - by dynamically choosing classes at runtime
 - In the long term
 - Uplink a new class file to a spacecraft, dynamically load it without rebooting
 - This is outside of our scope

On time, low cost

1) Provided closed-loop detumble demonstration

- ↗ Successful
- ↗ Demo provided on Sept 11, 2002.

2) Delivered Test Articles to Nasa Ames (ARC)

- ↗ Successful
- ↗ Attitude Control
 - ↖ Delivered Sept 23, 2002.
- ↗ Fault Protection Engine
 - ↖ Delivered August 22, 2002.

Accomplishments



-
- Provide demonstrations of code running on a Java desktop platform having DS1-like functionality
 - ☞ Closed-loop detumble
 - ☞ Fault protection
 - Create infrastructure for future Java flight software development
 - Demonstrate how to use Java to create flight-like code which is
 - ☞ Maintainable
 - ☞ Readable
 - ☞ Extensible
 - ☞ Object-Oriented

Goal: Prove that Java has acceptable performance on flight hardware

Run on PPC with existing DS1 simulator

Perform a detumble

Tasks

- Install real-time Java in autonomy lab environment
- Test and debug with simulator
 - Stage 1: Interface to hardware device simulation
 - IMU, thrusters, bus
 - Stage 2: Full attitude control
- Targeted performance improvements
- Measure CPU and memory usage

Contact Information



- Ed Benowitz
 - ↗ eddieb@mail2.jpl.nasa.gov
- Al Niessner
 - ↗ Albert.F.Niessner@jpl.nasa.gov