A Reusable Java Technology Fault Protection Framework

Ed Benowitz
Technical Staff
Jet Propulsion Lab
Learn an approach to fault protection using Java and state diagrams.
Speaker’s Qualifications

- Ed Benowitz is a software developer at the Jet Propulsion Laboratory
  - Previous incarnations:
    - Java sustaining engineer for Sun Microsystems
    - Developer for Raytheon
  - Over 6 years of Java development experience
  - MS from UCLA
Can we bridge the gap between systems engineers and software engineers?

- Fault protection requires a very close coordination between system engineering and software engineering. Is there a way to bridge this gap? We'll investigate this question as we explain our approach to a fault protection implementation in Java.
Presentation Agenda

- Introduction to Fault Protection in Space
- Approach
- Fault Detection
- Response Design and Implementation
- Response Scheduling
Fault Protection in Space

- Hardware and software failures
- Automatically take appropriate measures
- Tight interaction between
  - System engineering
  - Fault protection subsystem
- Can we implement it in Java?

- Fault protection is a necessary functionality for spacecraft
- Can we implement it in Java?
- In inter-planetary missions, can't fix hardware, and difficult to make changes to software
- Onboard fault protection needs to keep the spacecraft in a safe state, even when faults occur. This typically needs to happen without intervention from the ground.
- To implement fault protection, the software engineer must work closely with the system engineer.
Approach (1/2)

- Pure Java implementation
- Use of design patterns
- Create reusable components

- Base our work on Deep Space 1 and Deep Impact missions
- Use pure Java
- Re-architect using best practices in Object-Oriented development
  - Include design patterns
  - States pattern specifically
- Code must remain independent from a particular mission
- Create a set of reusable components
Single failure operation:
- Capability is provided to recover from a single fault and continue its mission. Multiple faults are handled sequentially, where only one response is active at a time.
- Other missions that have used this level of Fault Protection include Galileo, Mars Pathfinder, DS1 and Deep-Impact.
- Several sets of reusable components
  - Threshold is used for detecting erroneous data, an indication that a fault is present
  - Responses, implemented as state charts, provide a way to deal with a previously detected faults. The state chart notation, as we will see, is a convenient way for a system engineer and a software engineer to communicate response specifications
- The engine is used to schedule responses, enforcing the single fail operation regime. We will briefly touch on its capabilities
Fault Detection

- Persistently bad data may indicate a fault
- Threshold component
  - Tracks values over time, judges if high/low
- Publish/subscribe model for notifications
- Threshold implemented with States pattern

- The publish subscribe pattern can be used to listen to data, or to listen for state changes. Quite similar to the idea of event handling done in AWT, for example.
- We'll now discuss how fault detection can be done via a threshold component.
- The threshold component itself is implemented with the states pattern. We'll discuss this pattern a bit later.
High and low boundaries indicated the expected range of the data.

Persistence indicates how long a data must be out of range before we indicate a high or low value.

Confidence indicates how long the data must be in-range before we indicate that the value is nominal again.
- Data value changes over time
- At some point cross a threshold
- If the data is above the threshold value for a given amount of time (which is stored in a persistence variable), then the state of the threshold component is declared high.
- Similarly for low
Data value changes over time
This time the data changes from being high to eventually nominal
If the data is below the threshold for a certain amount of time (indicated by the confidence variable), then we report a nominal value.
Threshold Interface

```java
public interface Threshold
{
    public void changeParameters(
        double min, double max,
        int confidence, int persistence,
        int decay);
    public boolean isHigh();
    public boolean isLow();
    public void update(double value);
}
```

- Simplified interface
- Decay is used to determine when to reset internal history
- Publish/subscribe mechanism not included here
- Reflects the parameters we discussed previously
- Update is called when the value being watched has changed
- Decay indicates how long we should wait before erasing the counts of history
Once a threshold detects that a value is high, we declare that a symptom exists. A symptom is some indication of a malfunction.

Based on symptoms, the fault protection system acts as a doctor, and determines the likely underlying cause of these symptoms. This is known as the fault.

Once the fault is determined, the subsystem then executes the appropriate response in an attempt to deal with the fault.

There is a built-in mapping between symptoms and faults and between faults and responses.
Response Design and Implementation

- Interaction with the system engineers
- Need clear communication
- Need common language

- Responses depend heavily on input from system engineers
We'll explain some of the advantages of this approach.

- State charts need not be restricted to the fault-protection domain. In general, however, graphical state-charts can be used as a way of specifying
State-Charts (2 / 3)

- Advantages
  - Precise specification
  - Visual representation
  - Easy for system engineers to analyze
  - Possibility of auto-coding
  - Debug the design before coding

- Auto-coding was in fact done for both DS1's state charts and Deep Impact. DS1 flew in space with all of its fault protection responses auto-coded from state charts.
- Designers could exercise the state charts in Matlab's StateFlow beforehand.
- Auto-coding for Java is a possibility, but it was outside of our scope for the time being
State-Charts (3 / 3)

- Stateflow state-charts can express
  - States, which can be composed
  - Transitions
  - Code blocks
    - Entry
    - Exit
    - During

- Composition is a useful abstraction for working with large systems
  - Although not traditionally used in say, the finite state automata of computer science theory
- In state composition, an entire state machine is embedded within a parent state
- Child state machine runs when its parent state is active.
- Each state can specify a block of code to be run on exit, entry, and during.
interface ColorState
{
    public void doAction();
}

public class Red implements ColorState
{
    public void doAction()
    
    ColorState current_color = new Red();
    current_color.doAction();
    current_color = green;
    current_color.doAction();

- Avoid the problems with switch statements
  - Forgetting a break
  - Adding a new state requires going through all functions
    finding the right place to insert a new case
  - No need to assert(false) at the end of a switch
- Easy to extend
- Clearly separates each state into its own implementation class
- States as objects
For this slide, hierarchy refers to composition, not inheritance.
Each state will override onEntry, onExit, and during.
Activate method handles the book-keeping of calling the proper methods at the proper times.
Clear mapping between a statechart and a Java Object.
Users would typically subclass LeafState.
Another valid approach would be to explicitly make transitions themselves objects. This is left as an exercise for the reader.
For our case, states had typically 1 transition, or occasionally 2 so the explicit transitions as object was not chosen.
For this particular implementation, variables stored across states but local to a state chart are stored within a response.

States are then given a reference to the response for context.
Example Java State (1 / 2)

```java
public class InitState extends LeafState {
    public void onEntry()
    {
        response.incrementCounter();
    }
}
```

- Shows the same state chart as previously, translated to Java
public void during()
{
    if(response.getCount() == 1)
        statex.activate(this);
    else if(response.getCount() == 2)
        statey.activate(this);
}
Response Scheduling

- One response at a time
- Responses can call other responses
- Long-running response can pause
  - Allow another response to interrupt it
- Reusable implementation

- Engine is derived from C++ version, credits to Garth Watney
- Design remains identical
  - Needed to be identical to assist in the verification of the C++ version
  - Behavior matched C++ so well that a V&V bug detected by NASA Ames in the Java version indicated an identical bug in the C++ version
  - Except that responses are an interface in Java instead of an abstract class as in C++
Summary

- Fault Protection can be implemented in Java
- Java facilitates implementation of the states design pattern
- State-charts transfer system engineering knowledge to software
- Fault protection components can be reused
State-charts bridge the gap between system engineering and software.
Q&A