CubeSat Flight Software Workshop

Flight Software Architecture Principles

Garth Watney
Flight Software Developer
June 3, 2019

Jet Propulsion Laboratory
California Institute of Technology

Software Architecture

- Software Architecture is different from Software Design
- Software Architecture describes the skeleton and high level infrastructure of the software
    - Independent of the application domain
- Software Design describes the implementation of the domain within the software architecture
    - Breaks the software down into elements
    - Describes the purpose of each element
    - Describes the inner workings of each element
- Software Architecture is important
    - Antidote to software chaos
    - Glue and foundation that holds the software together
- Be vigilant against architectural erosion
    - Maintain the architectural integrity throughout development

Software Architecture

- Examples of different software architecture
  - Pipes and Filters
  - Publish and Subscribe
  - Client-Server
  - Blackboard
  - Data-base
  - Event-driven
  - Component
- Classic JPL Flight Software Architecture
  - Multi-threaded module-based architecture
  - Modules only communicate through events using message queues
  - Static point to point connection
  - Monolithic
- Component based architecture
  - A component is a unit of computation with a well-defined interface
  - A component has no symbolic dependencies on other components
    - Compile, Load and Execute independently of other components
  - Components only communicate with each other through ports
  - Components encapsulate threads and queues and states

Software Architectural Attributes

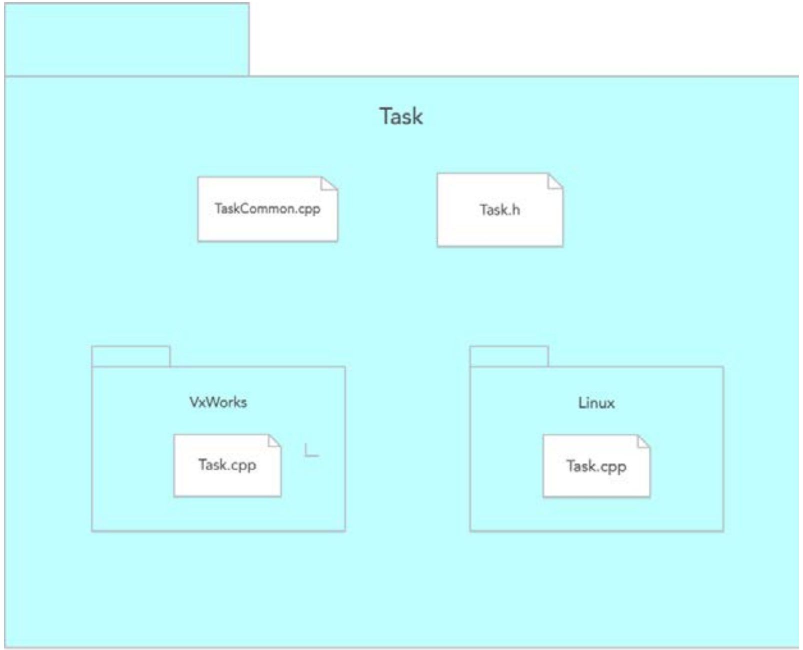
- Modularity
  - Modularity improves software development quality and maintainability
  - Decompose the software into a collection of modules (components or libraries)
  - A module is
    - A unit of work assigned to a developer
    - Has a well-defined set of requirements
    - Has a well-defined interface
    - Unit tested before being delivered into the integration build

Software Architectural Attributes

- Module Coupling
  - The extent that modules are related to each other
  - Examples of high coupling (bad)
    - Control coupling – one module controls the flow of another module by passing a “what-to-do” flag
    - Data coupling – modules share a common data space
    - Content coupling – modules share common code or data structures
- Module Cohesion
  - The extent that data and functions inside a module belong to each other
  - Examples of high cohesion (good)
    - All the functions and data for a device driver pertain to the operation of the device
    - A Telemetry Manager module only processes telemetry channels and not commands
- Strive for Low Coupling and High Cohesion

Software Architectural Attributes

- Portability
  - Software that is portable to a desktop workstation is significantly easier to develop.
  - Ensure your software is readily portable to your desktop workstation (Linux/Windows) and not just the embedded target
  - Hide Operating System differences in an OSAL (Operating System Adaptation Layer).
  - Avoid the use of scattered conditional compilations by creating different implementations of a class or function at the lowest level.



Software Architectural Attributes

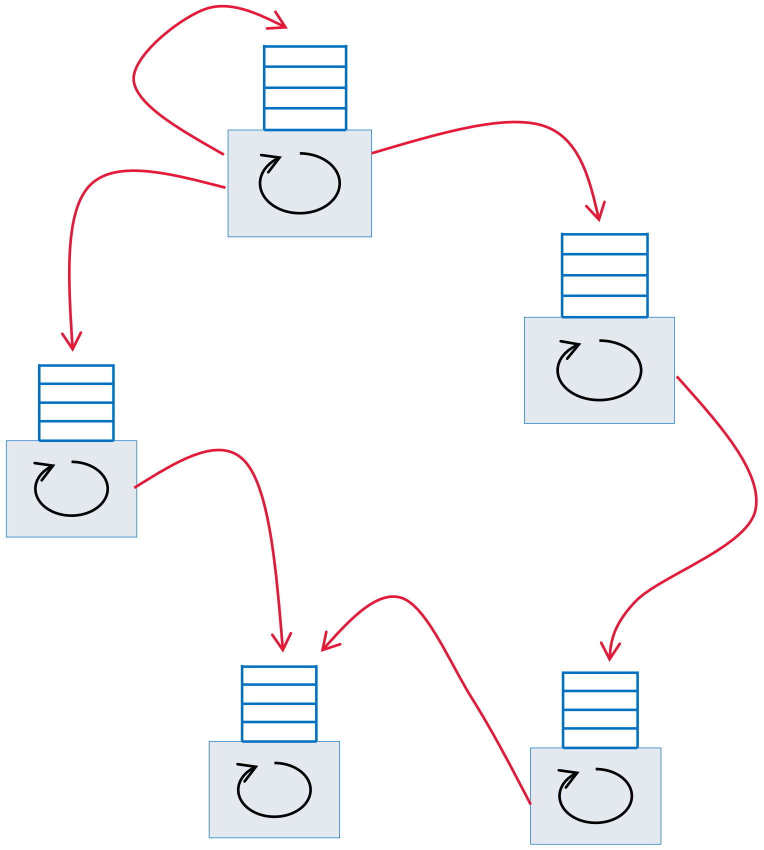
- Reusability
  - Use frameworks, libraries, algorithms, design patterns that are well tested and understood.
  - Fprime framework with its core components are an example of good reusability
  - Quantum Framework is a relatively simple and powerful framework for implementing hierarchical state-machines
  - “Design Patterns” by the “Gang of Four” present well understood software design patterns.

Software Architectural Views

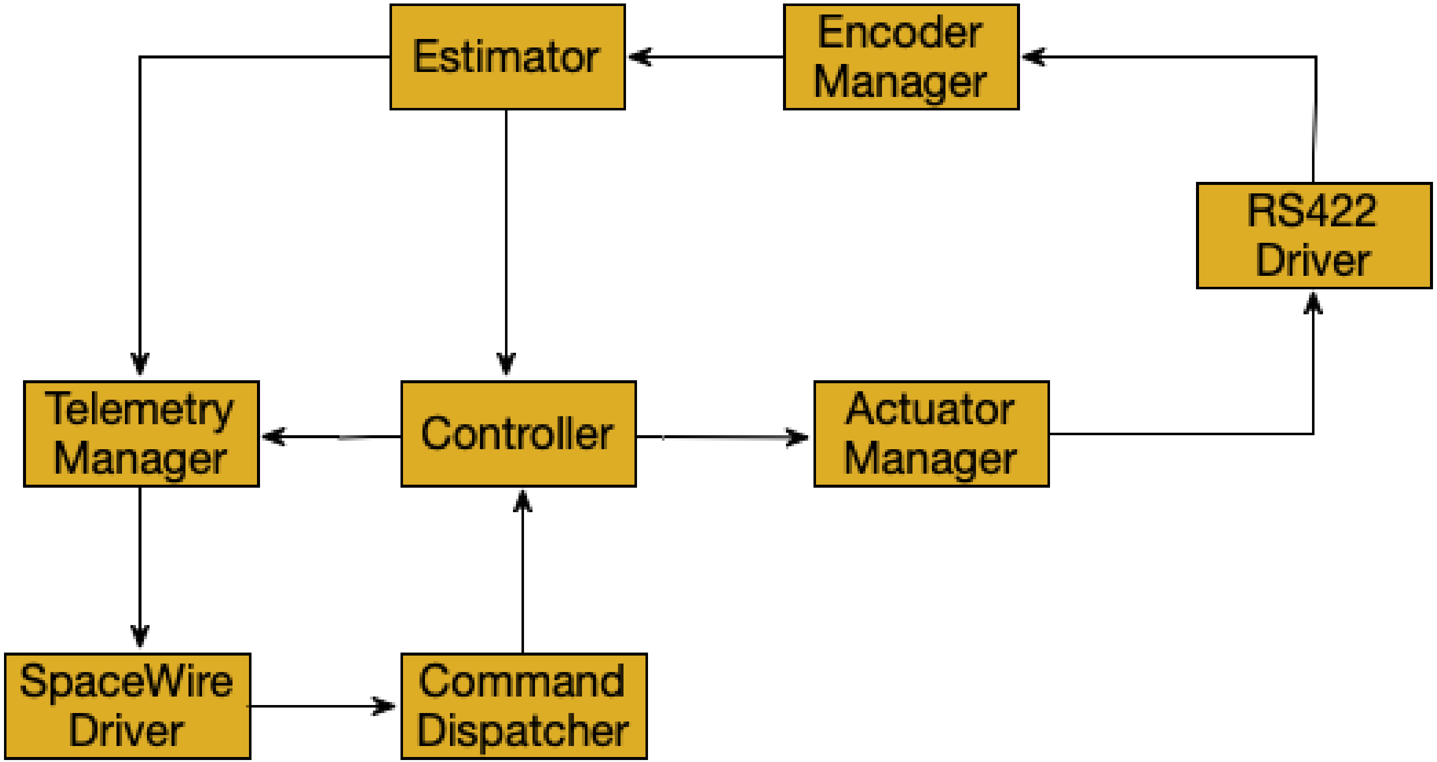
- Software architecture is captured by different views or perspectives.
- These perspectives encompass the software architectural model
- These perspectives are not mutually exclusive

Software Task View

- Tasks are execution threads
- Tasks communicate via event messages which are placed on the task input queue
- Tasks sleep until a message arrives and then process events off their input queue
- Tasks have execution priority
- Tasks can be:
    - Rate-group driven (1 Hz, 10 Hz etc)
    - Data driven
    - Background (continuous low priority task)

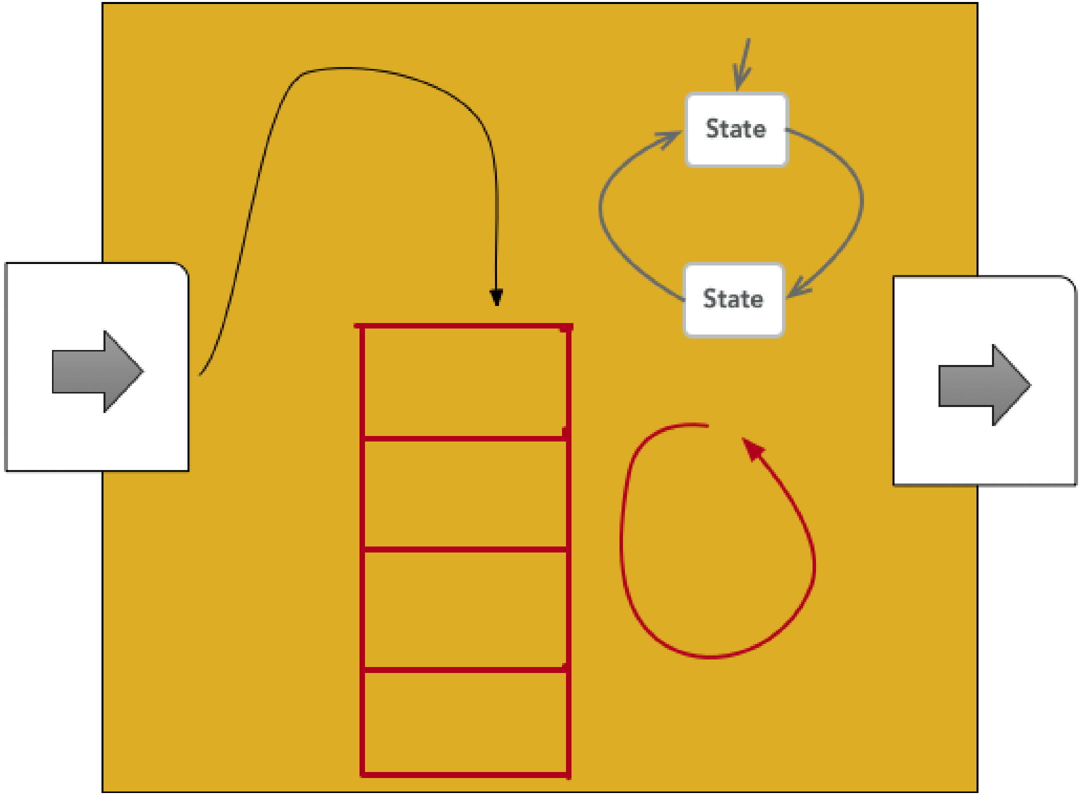


Software Component View



Software Component Encapsulation

- A component encapsulates
  - A task
  - A state-machine
  - An input queue
  - Input and Output Ports

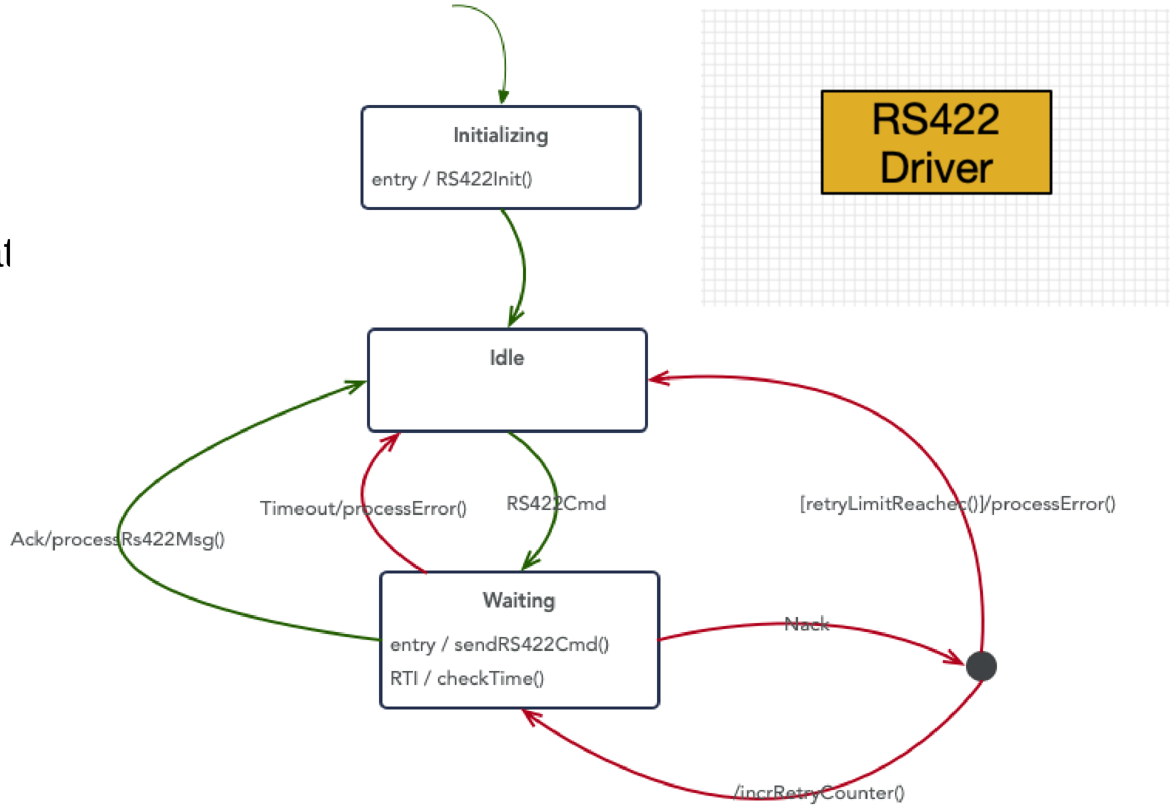


Software Component State

- For each component create a crisp notion of state with a state-machine that has the following identified:
  - Discrete states
  - Events that the state-machine consumes
  - Transitions between the states
  - State entry and exit behavior
- Encapsulate the state-machine logic within a single function or class
- Avoid a fuzzy notion of state with a collection of Boolean flags and scattered state logic.

RS422 Driver Component State-machine example

- The RS422 Driver shall handle commands from different clients.
- The RS422 Driver shall process one command at a time waiting for an Ack or Nack.
- The RS422 Driver shall retry the command upon receiving a Nack up to a specified limit
- RS422 Drvier shall time-out after a specified duration.



State-machine Implementation

```
void updateStateMachine(StateMachineEvent event) {

    switch (myState) {

        case START:
            // Transition to INITIALIZING
            myState = INITIALIZING;
            pushEventQ(Entry);

        case INITIALIZING:

            switch (event) {

            case Entry:
                RS422Init();
                // Transition to IDLE
                myState = IDLE;
                pushEventQ(Entry);
                break;

           default:
                break;
           }
        break;
```

State-machine Implementation

```
case IDLE:

    switch (event) {

    case RS422_Cmd:
        // Transition to WAITING
        myState = WAITING;
        pushEventQ(Entry);
        break;

    default:
        break;
    }
break;
```

State-machine Implementation

```
case WAITING:

    switch (event) {

    case Entry:
        sendRS422Cmd();
        break;

    case RTI:
        checkTime();
        break;

    case Ack:
         processRs422Msg();
         // Transition to IDLE
         myState = IDLE;
         pushEventQ(Entry);
         break;

    case Nack:
         if (retryLimitReached()) {
             processError();
             // Transition to IDLE
             myState = IDLE;
             pushEventQ(Entry);
         }
         else {
             incRetryCounter();
             // Transition to WAITING
             myState = WAITING;
             pushEventQ(Entry);
         }

    case Timeout:
        processError();
        // Transition to IDLE
        myState = IDLE;
        pushEventQ(Entry);


    default:
         break;
    }
break;
```

Other Software Architectural Principles

- No dynamic memory allocation after initialization
  - Deterministic behavior
- No multiple class inheritance
- Limit class hierarchy
- Integrity checks
  - Asserts
- Performance
  - The software should perform well in a resource constrained environment.
- Keep it simple
  - If your code is complicated and ugly, it’s probably wrong.

Jet Propulsion Laboratory
California Institute of Technology

jpl.nasa.gov