Architecture, Requirements & Design

Garth Watney
NASA Jet Propulsion Laboratory
17 October 2022

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

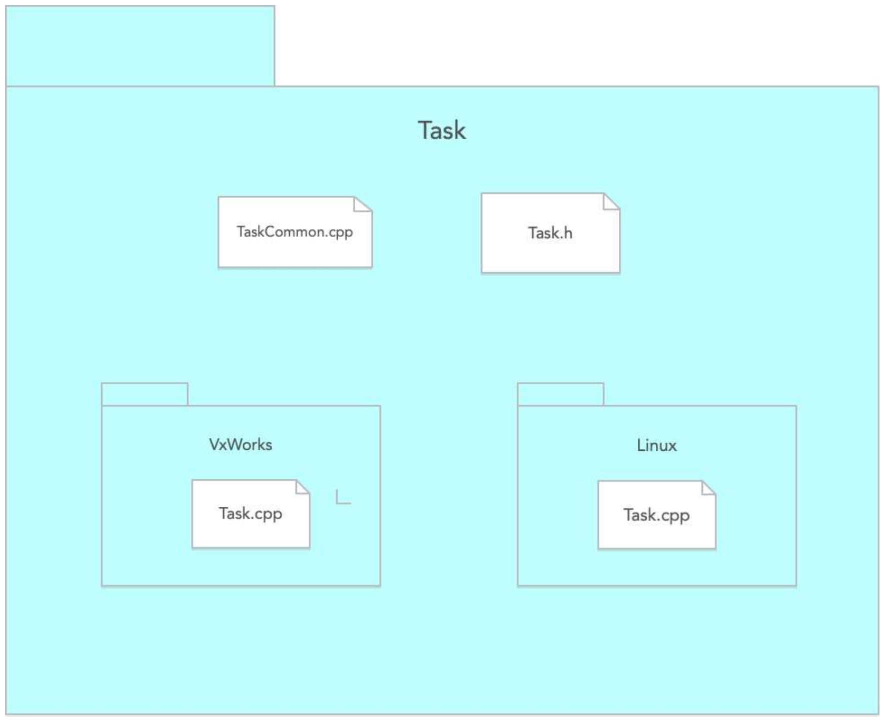
- Modularity
    - Modularity improves software development quality and maintainability
    - Decompose the software into a collection of modules (components or libraries)
    - A module is
        - A unit of work assigned to a developer
        - Has a well-defined set of requirements
        - Has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

Software Architectural Attributes

- Module Coupling
  - The extent that modules (components) are related to each other
  - Examples of high coupling (bad)
    - Control coupling – one component controls the flow of another component by passing a “what-to-do” flag
    - Data coupling – components share a common data space
    - Content coupling – components share common code or data structures
- Module Cohesion
  - The extent that data and functions inside a module (component) 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 component 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.

Other Software Architectural Principles

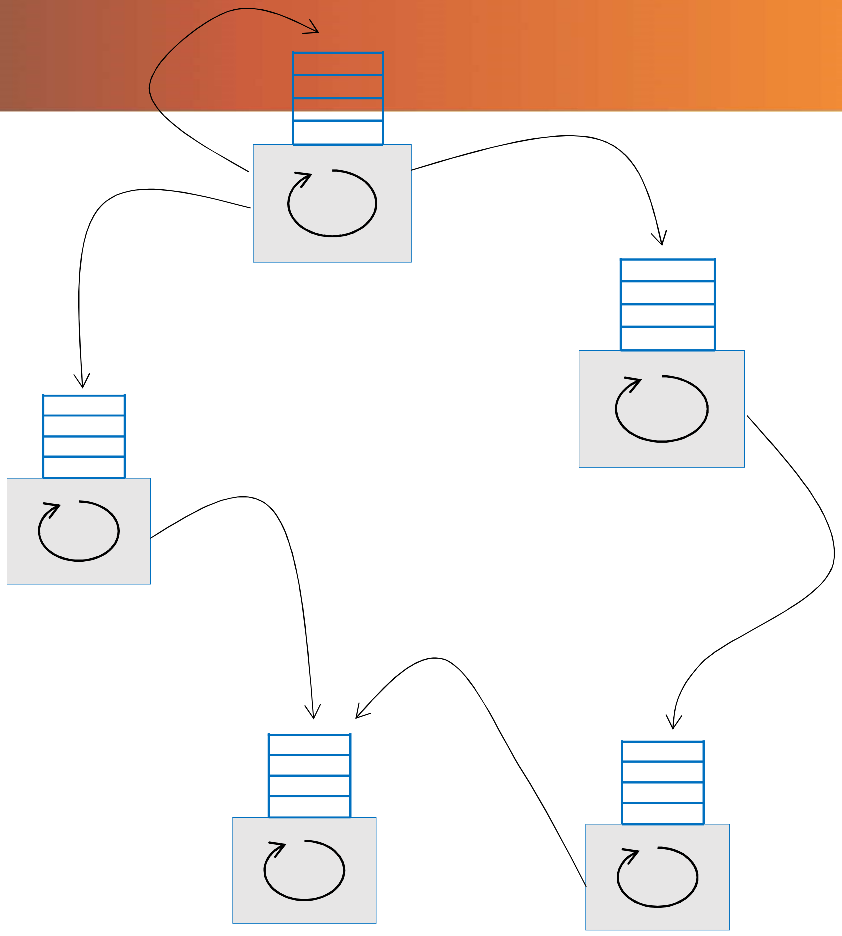
- No dynamic memory allocation after initialization
  - Deterministic behavior
- No multiple class inheritance
- Limit class hierarchy
- Integrity checks
  - Asserts
- Performance
  - The software should perform well in a resource constrained environment.
- Keep it simple
  - If your code is complicated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

Software Architectural Views

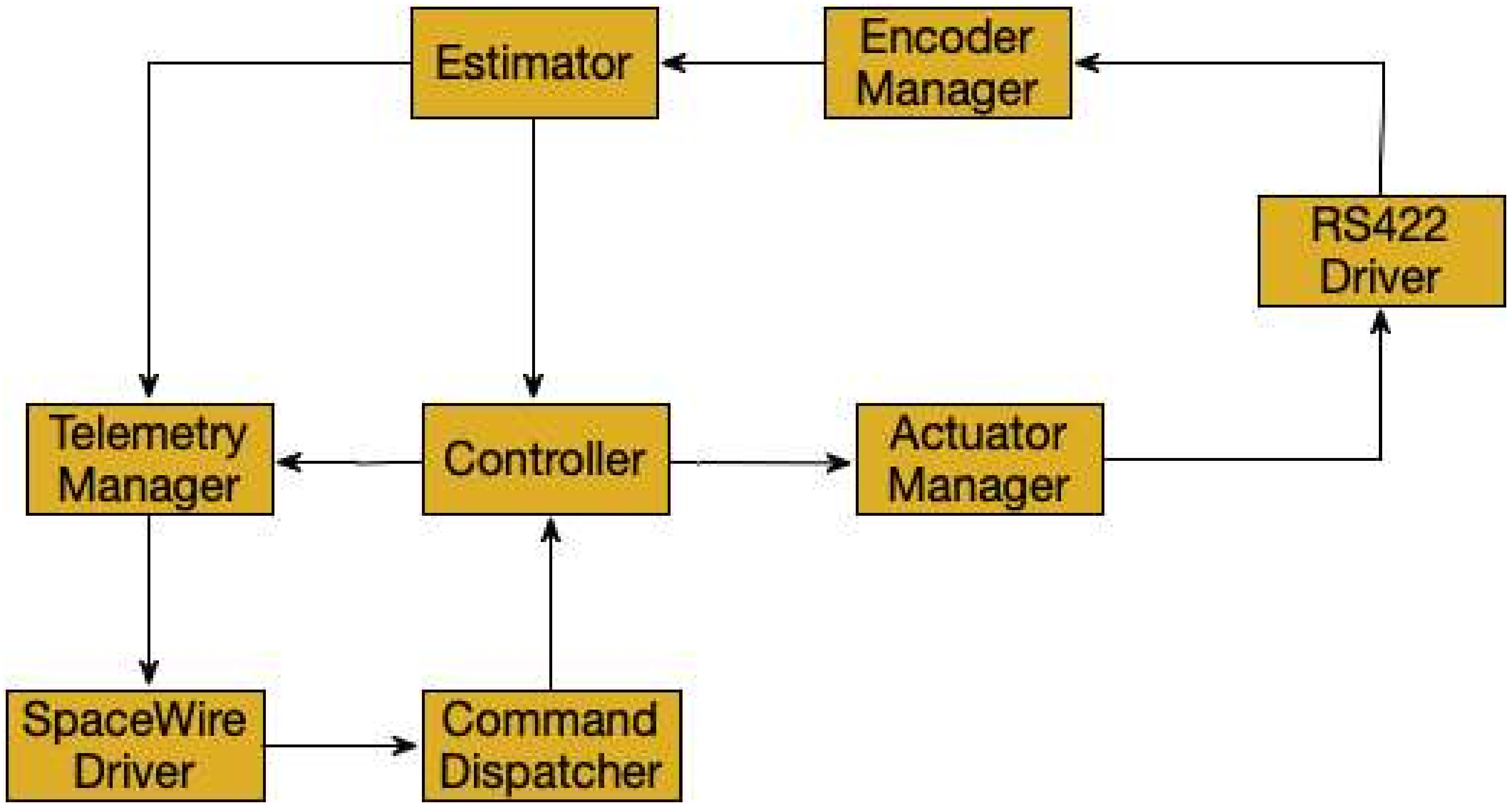
- Software architecture is captured by different views or perspectives.
- These perspectives encompass the software architectural model
- These perspectives are not mutually exclusive

Architecture Views:
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)

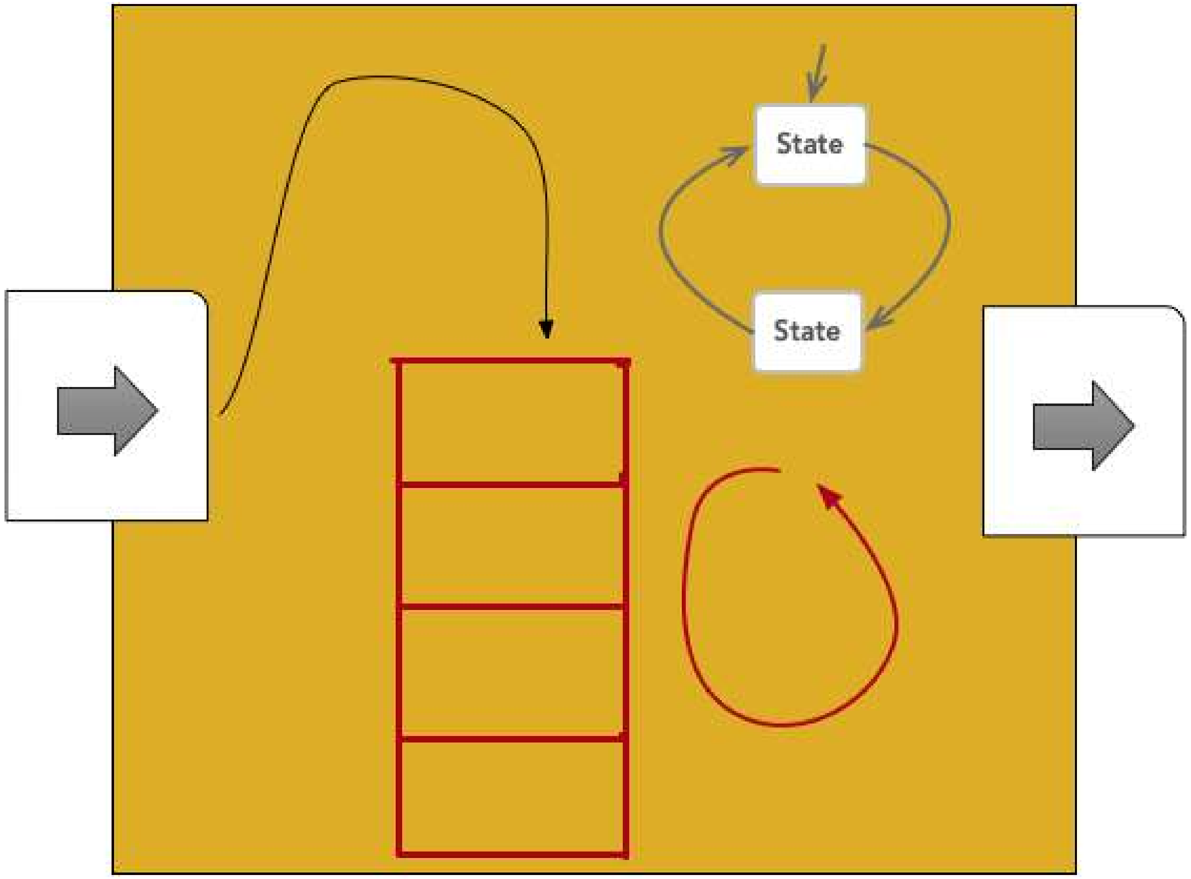


Architecture Views:
Component View



Software Component Encapsulation

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



Software Requirements

- Requirements are typically layered
  - Mission Requirements
  - Project Requirements
  - System Requirements
  - Subsystem Requirements
  - Flight software Requirements
  - Component Requirements
- Requirements are traced up and down
  - Higher level requirements are satisfied by lower level requirements
  - Lower level requirements are traced back to upper level requirements

Software Requirements

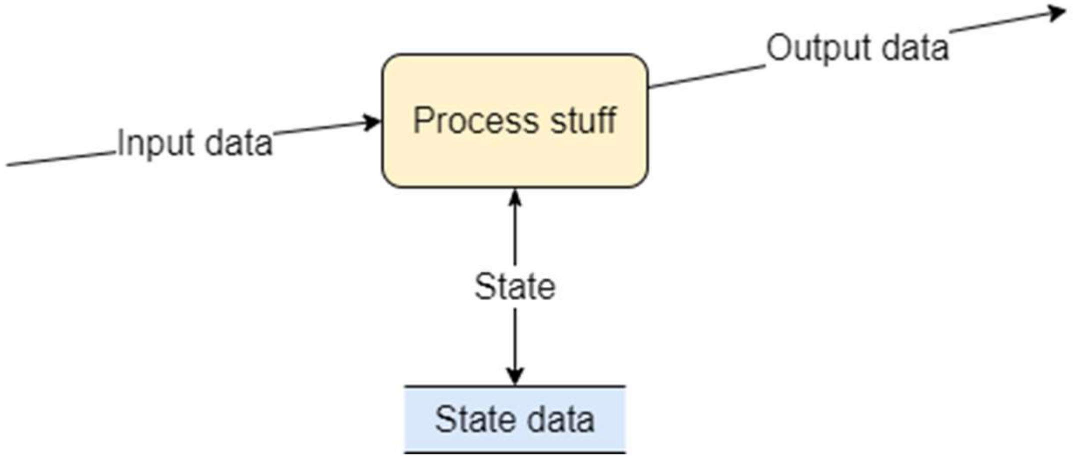
- Focus on Flight Software Requirements
  - FSW Requirements should be:
    - Concise
    - Unambiguous
    - Testable
    - A shall statement
  - Help you, the developer, to know what you are implementing
  - A contract to the project on what you are implementing
    - Avoid nasty surprises when your software is finally delivered.
  - Examples:
    - The FSW shall produce spacecraft health information via telemetry.
      - Context information can also be provided as an auxiliary
        - Spacecraft health information consists of the following …
    - The FSW shall allow ground operators to change parameter values defined in the parameter dictionary.
    - The FSW shall receive files from the ground, validate them, and store them in flash memory.

Software Requirements

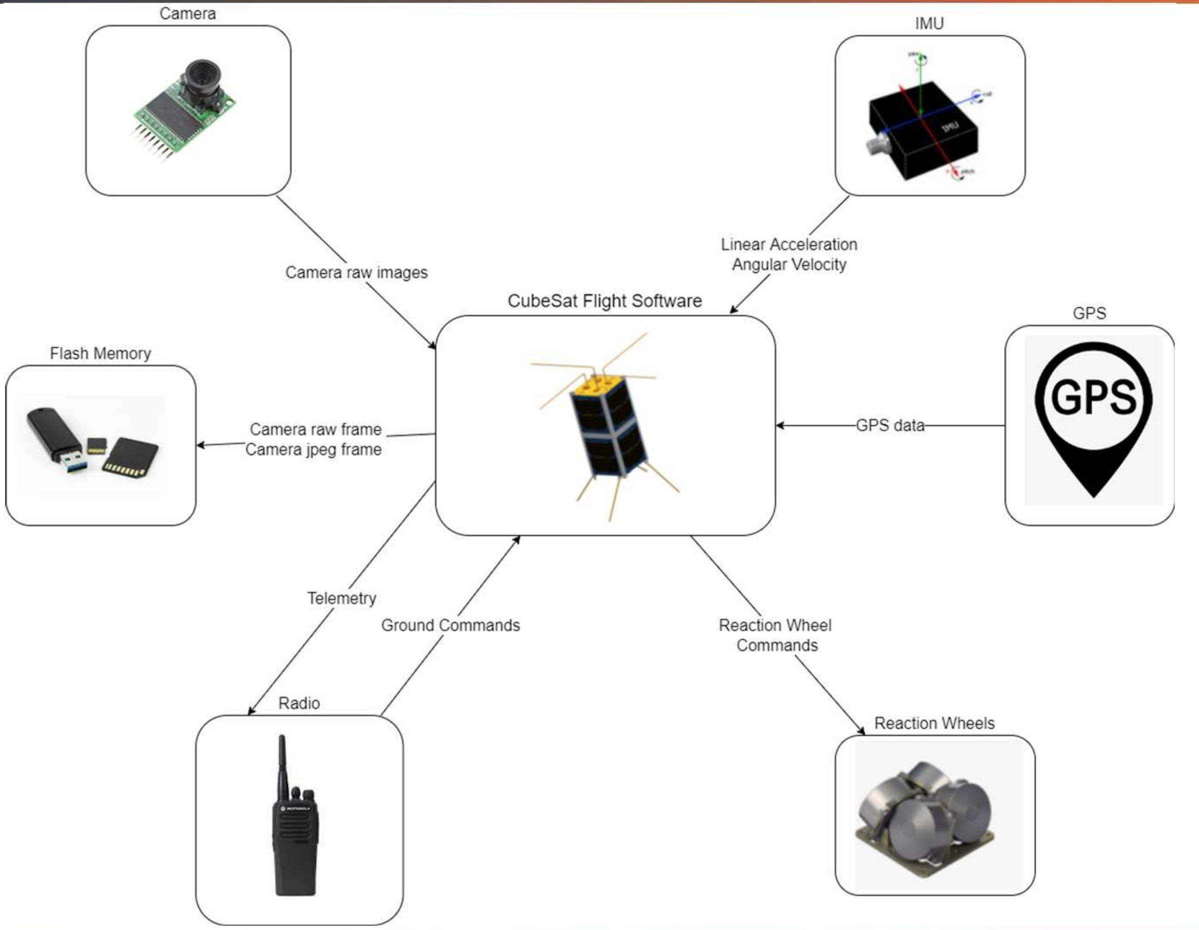
- Before design and code:
  - First answer the question of What are we building?, NOT How do we implement it?
- Thoroughly understand what we building
  - Follow a process that will develop the FSW Requirements
- Structured Analysis is a process that generates a Data Flow Diagram:
  - Answers the question: What are we building
  - Naturally produces the FSW Requirements
  - Not meant to show design
  - Does not capture timing, sequence and synchronization of processes
  - Breaks the system down into smaller manageable chunks
  - A graphical diagram that is relatively easy to understand for software/system engineers
  - A clear and detailed information about the system – processes and boundaries
  - Shows the logic of the data flow

Data Flow Diagram Model

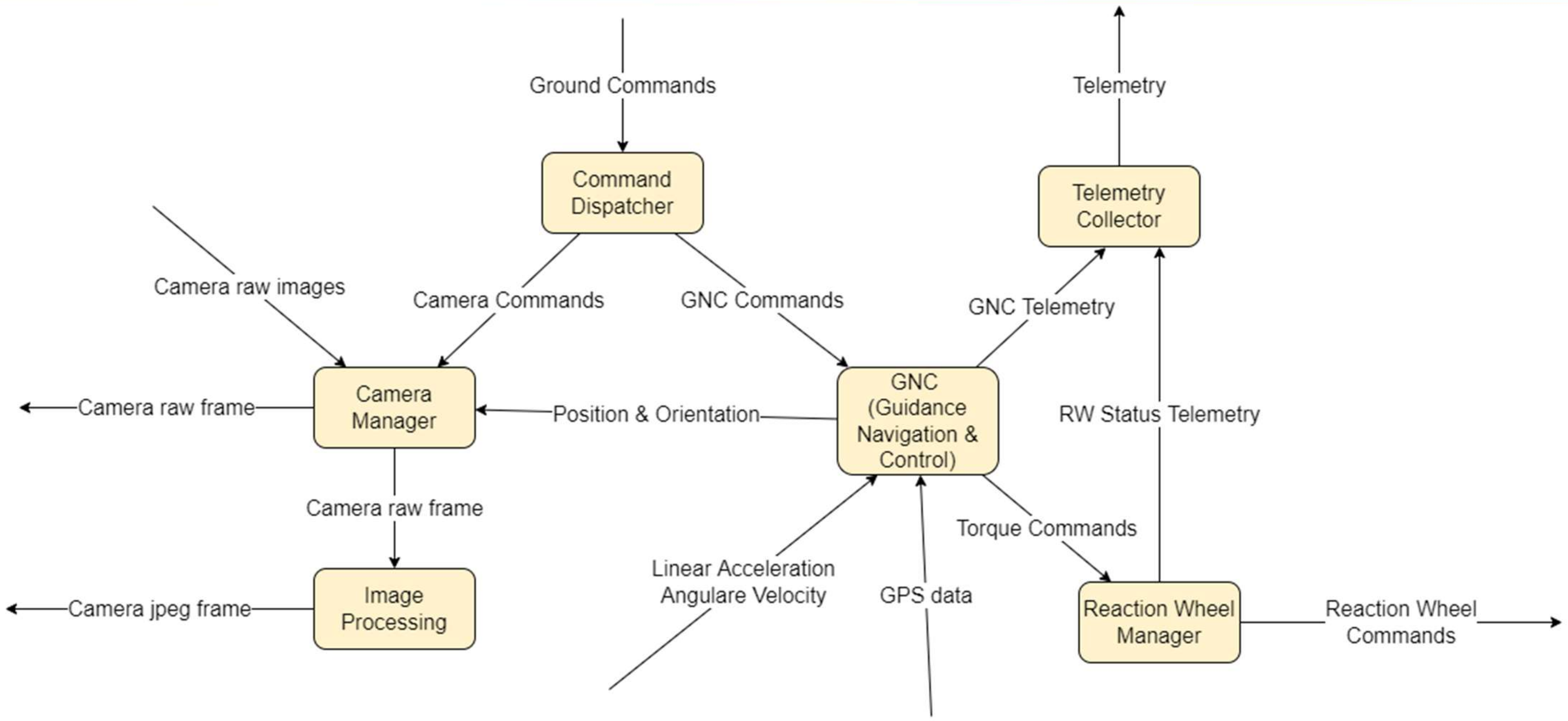
- Process (data transformation)
- Data flows
- Data stores
- Levels
  - Context diagram
    - Focus on system interfaces
  - Process decomposition
    - Overviews of system processing
  - Deeper dives
    - Detailed views



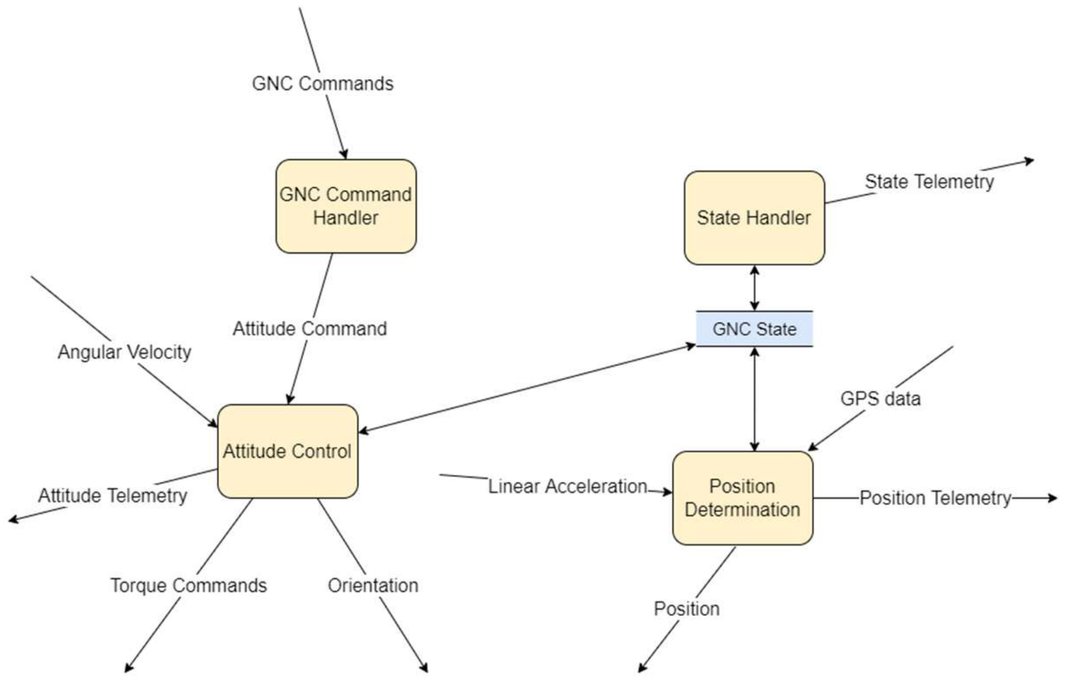
Flight Software Context Diagram



Top Data Flow Diagram



GNC Data Flow Diagram



- The FSW shall convert the GNC Command to a 3 axis pointing attitude command with an earth reference frame.
- The FSW shall send the GNC state as a telemetry channel.
- The FSW shall compute a quaternion position (a,b,c,d) from the Linear Acceleration (x,y,z) and current GPS data and send the position out as a telemetry channel.
- The FSW shall compute 3 axis torque commands (x,y,z) from a pointing attitude command and the angular velocity data.

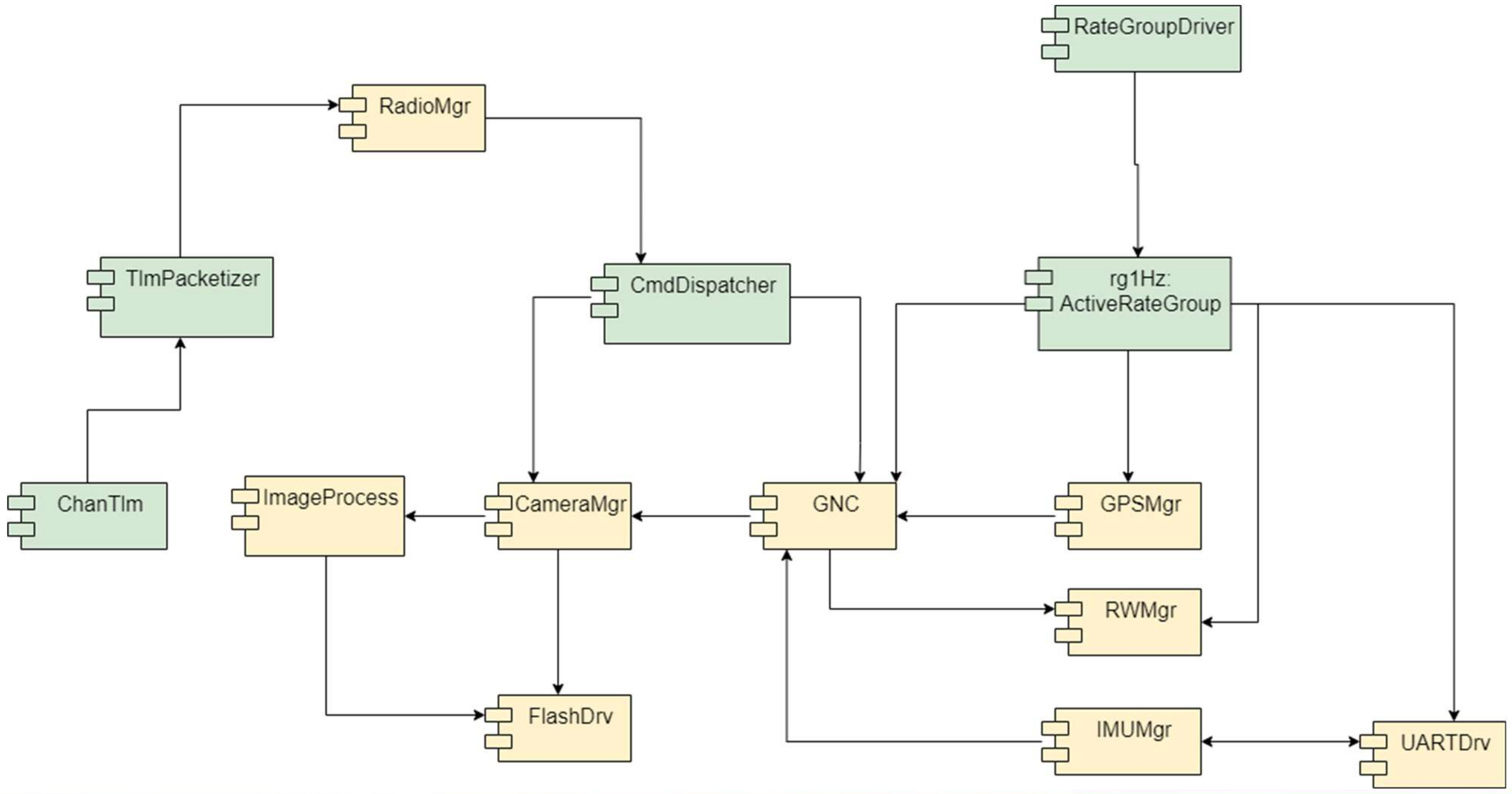
From Requirements to Design

- Data Flow Diagrams (DFD's) specify:
  - Flight Software Interfaces
  - The flow of data to processes
  - The processing and transformation of the data
  - Data stores (state)
  - Used to derive flight software requirements
  - Natural transition to implementation
- Components specify:
  - The concrete implementation of the DFD
  - The periodic scheduling
  - The threads of execution (component type)
  - The component data interfaces
  - The component port types
    - Synchronous or Asynchronous

Design Decisions

- GPSMgr component: Implements interfacing the GPS hardware and the generation of GPS data
- RadioMgr component: Implements interfacing the Radio hardware and the processing of ground commands and flight telemetry
- IMUMgr component: Implements managing the IMU hardware and the generation of linear and angular velocity data
- UARTDrv component: Implements the low level communication to the IMU across the UART bus.
- FlashDrv component: Implements the writing of camera data products to flash
- CameraMgr component: Implements managing the Camera hardware and the Camera Manager processing
- ImageProcess component: Implements Image Processing
- GNC component: Implements the GNC processing
- F’ components ChanTlm and TlmPacketizer: Implement the collection and packetizing of telemetry data from all components
- F’ component CmdDispatcher: Implement the dispatching of ground commands to all components
- F’ components RateGroupDrv and ActiveRateGroup: Implement the periodic scheduling of components processing.

Component Topology diagram

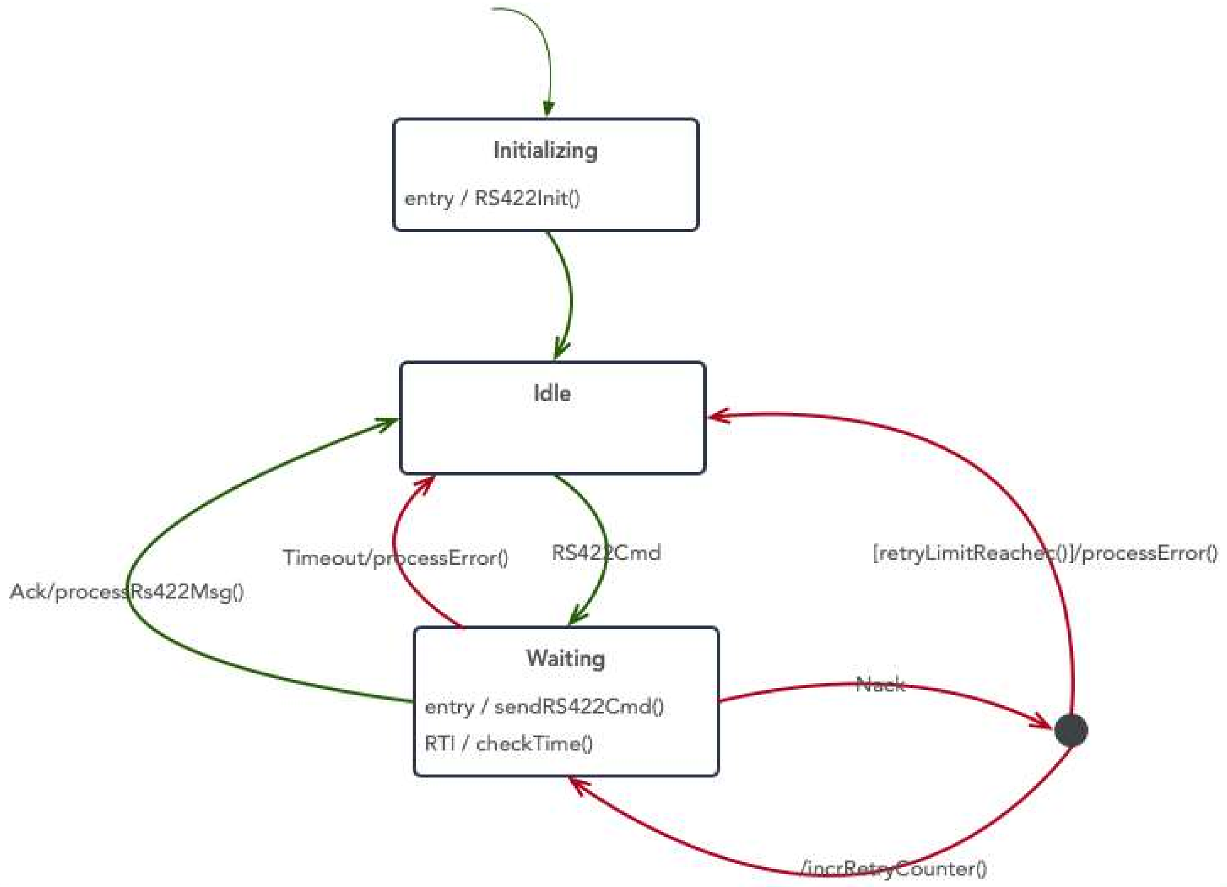


Software Component State

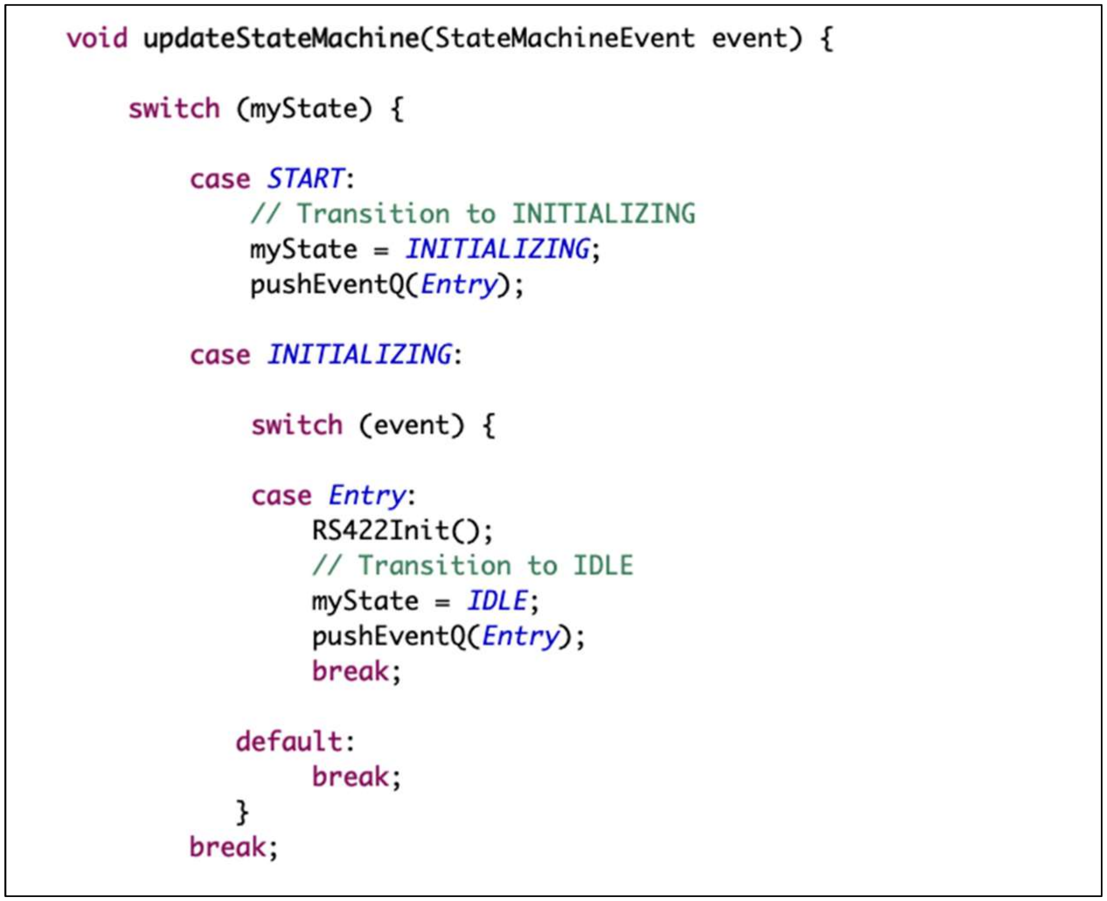
- For each component create a crisp notion of state with a state-machine that has the following identified:
    - Discrete states
    - Events that the state-machine consumes
    - Transitions between the states
    - State entry and exit behavior
- Encapsulate the state-machine logic within a single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

UART Driver Component State-machine example

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

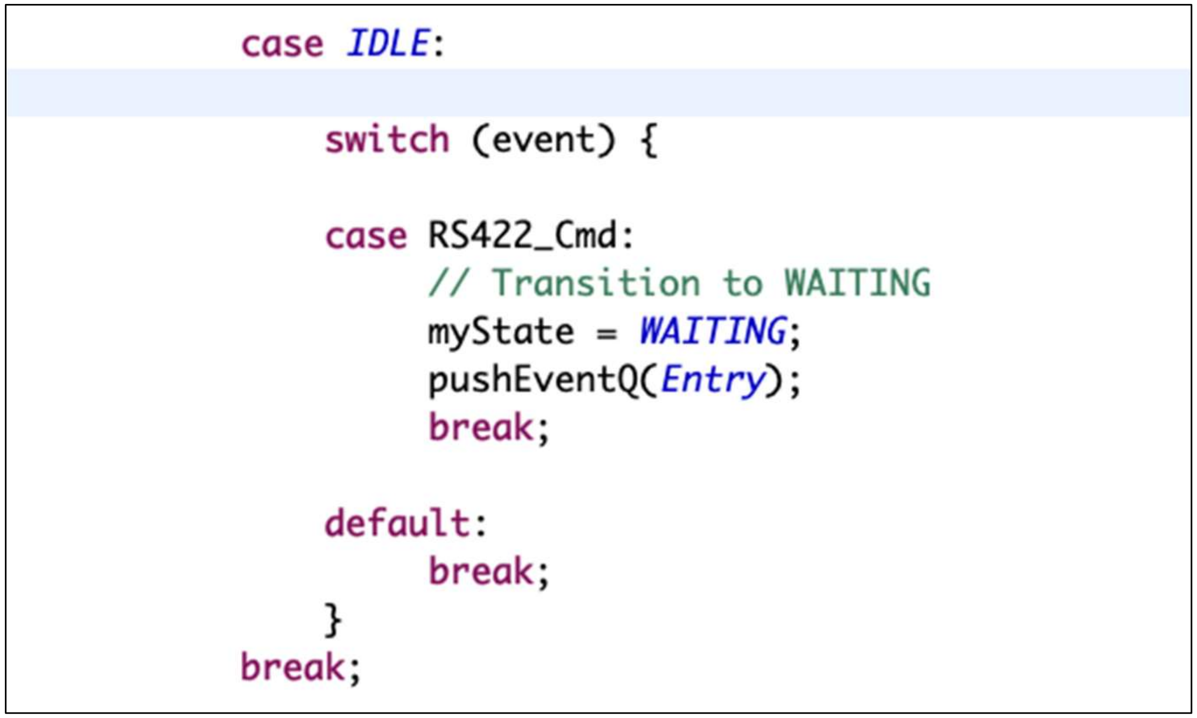


State-machine Implementation



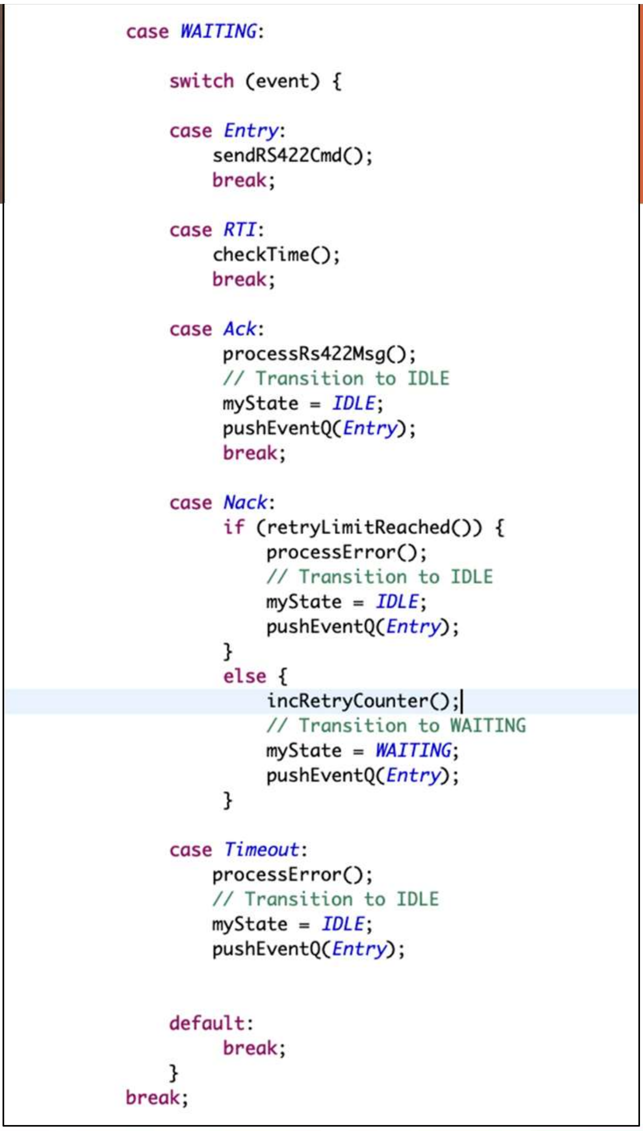
```
void updateStateMachine(StateMachineEvent event) {

    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;
```

Take-home message

- Understand and maintain the software architecture throughout development
- Understand what you are building before making design choices
- Generate requirements from a well understood analysis model
  - i.e. Data Flow Diagram or other models
- Capture the design as a topology model
  - Component instances
  - Component types
    - Active (Thread of execution)
    - Passive
    - Queued
  - Component interfaces
    - Data structure
    - Synchronous/Asynchronous
- Maintain a crisp notion of state for each component
  - Use a good state machine design pattern
  - https://www.state-machine.com