Organizational Charter Focuses on Embedded Systems and Software Products

- Embedded software for
  - Advanced robotic spacecraft platforms
  - High-bandwidth satellite payloads
  - High-power laser systems

- Emphasis on both system management and payload software

- Reusable, reconfigurable software architectures and components

- Languages: O-O to C to asm

- CMMI Level 5 for Software in February 2004; ISO/AS9100; Six Sigma

- High-reliability, long-life, real-time embedded software systems

Software Development Lab

Software Analysis

Software Peer Review

Software Process Flow for Each Build, with 3-15 Builds per Program

Prometheus / JIMO

NPOESS

JWST

EOS Aqua/Aura

Chandra

GeoLITE

AEHF

MTHEL

Airborne Laser

Restricted
Prometheus Spacecraft Supports Jupiter JIMO Mission over 9 to 14 Year Duration
Prometheus Spacecraft for JIMO and Related Missions Enables Data-Intensive Science

- Spacecraft configuration PB1
  - 58m length
  - 36,375kg launch mass
  - 5 processors, excluding redundancy
  - 250mbps transfer, 500gbit storage, 10mbps downlink
  - Gas cooled power with 200kW Brayton output
  - Stows in 5m diameter fairing
- **Embedded software** implements functions for commands & telemetry, subsystem algorithms, instrument support, data management, and fault protection.

- Size of on-board software growing to accelerate data processing and increase science yield.

- Software “adds value” to mission by enabling post-delivery changes to expand capabilities and overcome hardware failures.
### Research Investigates Systems and Software Engineering Principles, Benefits, and Tradeoffs

<table>
<thead>
<tr>
<th>PRINCIPLES</th>
<th>BENEFITS and TRADEOFFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifecycle models: Frequent synchronized design cycles and system releases</td>
<td>Organization of and parallelization within large-scale projects</td>
</tr>
<tr>
<td></td>
<td>Rapid feedback and innovation</td>
</tr>
<tr>
<td></td>
<td>Visibility into stabilization and handoffs</td>
</tr>
<tr>
<td>System architectures: Layered system architectures containing embedded meta-language programs and interpreters</td>
<td>User-customizability</td>
</tr>
<tr>
<td></td>
<td>Multi-platform portability</td>
</tr>
<tr>
<td></td>
<td>Automated testing</td>
</tr>
<tr>
<td>Reuse analysis: Reconfigurable component-driven development</td>
<td>Sustainable multi-project reuse</td>
</tr>
<tr>
<td></td>
<td>Lower component defect rates</td>
</tr>
<tr>
<td></td>
<td>Lower component development effort</td>
</tr>
<tr>
<td>Structure analysis: Inter-component connectivity analysis</td>
<td>Lower component defect rates</td>
</tr>
<tr>
<td></td>
<td>Lower component defect correction effort</td>
</tr>
<tr>
<td></td>
<td>Lower component development effort</td>
</tr>
<tr>
<td>Defect detection techniques: Disciplined team-based peer reviews</td>
<td>Early lifecycle defect detection</td>
</tr>
<tr>
<td></td>
<td>Low out-of-phase defect rates</td>
</tr>
<tr>
<td></td>
<td>High return-on-investment for prevention</td>
</tr>
<tr>
<td>Measurement and prediction: Automated measurement-driven analysis infrastructure using predictive models</td>
<td>Early identification of high defect or high effort components</td>
</tr>
<tr>
<td></td>
<td>Statistical process control</td>
</tr>
<tr>
<td></td>
<td>Pro-active process guidance</td>
</tr>
</tbody>
</table>
Establish Embedded Systems and Software Design Principles: Strategies

- Adopt risk-driven incremental lifecycle with multiple software builds to improve visibility, accelerate delivery, and facilitate integration and test.
- Share best-of-class software assets across organizations to leverage ideas and experience.
- Define common software processes and tools to improve communication and reduce risk.
- Conduct early tradeoff studies and end-to-end prototyping to validate key requirements, algorithms, and interfaces.
- Design simple deterministic systems and analyze them with worst-case mindset to improve predictability.
- Analyze system safety to minimize risk.
- Conduct extensive reviews (intra-build peer reviews, five build-level review gates, higher level project reviews) and modeling, analysis, and execution (testbed conceptual, development, engineering, and flight models called CMs, DMs, EMs, and FMs) to enable thorough understanding, verification, and validation.
## Incremental Software Builds Deliver Early Capabilities and Accelerate Integration and Test

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATP</td>
<td>PMSR</td>
<td>SM PDR</td>
<td>SM CDR</td>
<td>BUS I&amp;T</td>
<td>SM A&amp;I</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11/04</td>
<td>1/05</td>
<td>6/08</td>
<td>8/10</td>
<td>8/12</td>
<td>8/13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Flight Computer Unit (FCU) Builds

- **FCU1**: Prelim Exec and C&DH Software
- **FCU2**: Final Exec and C&DH Software
- **FCU3**: Science Computer Interface
- **FCU4**: Power Controller Interface
- **FCU5**: AACS (includes autonomous navigation)
- **FCU6**: Thermal and Power Control
- **FCU7**: Configuration and Fault Protection

### Science Computer Unit (SCU) Builds

**Note**: Science Computer builds for common software only (no instrument software included)

- **SCU1**: Prelim Exec and C&DH Software
- **SCU2**: Final Exec and C&DH Software

### Data Server Unit (DSU) Builds

- **DSU1**: Prelim Exec and C&DH Software
- **DSU2**: Final Exec and C&DH Software
- **DSU3**: Data Server Unique Software

### Ground Analysis Software (GAS) Computer Builds

- **GAS**: Preliminary Ground Analysis Software
- **GAS2**: Final Ground Analysis Software

### Delivered to, Usage

- **JPL/NGC, Prelim. Hardware/Software Integration**
- **JPL/NGC, Final Hardware/Software Integration**
- **JPL, Mission Module Integration**
- **JPL, Power Controller Integration**
- **NGC, AACS Validation on SMTB**
- **NGC, TCS/EPS Validation on SSTB**
- **NGC, Fault Protection S/W Validation on SSTB**

### Legend

- **N** = Design Agent
- **N** = Performer of Activity
- **JPL** = 1 Requirements
- **NGC** = 2 Preliminary Design
- **Role/activity shared by JPL and NGC** = 3 Detailed Design
- **Prototype Activity** = 4 Code and Unit Test/Software Integration
- **Verification and Validation** = 5 Verification and Validation

© Copyright 2001-2014. Richard W. Selby and Northrop Grumman Corporation. All rights reserved.
Synchronize-and-Stabilize Lifecycle Timeline and Milestones Enable Frequent Incremental Deliveries

**Phases**

- Development
  - Subproject I
    - 2-4 months (1/3 of all features)
  - Subproject II
    - 6-10 weeks
      - Code and optimizations
      - Testing and debugging
      - Feature stabilization
  - Subproject III
    - 2-5 weeks
      - Integration
      - Testing and debugging
    - 2-5 weeks
      - Buffer time
- Stabilization
  - 3-8 months

**Timeline**

- Planning
  - 3-12 months
- Project plan approval
  - Schedule complete
- Milestone I
  - Milestone I release
- Milestone II
  - Milestone II release
  - Visual freeze
  - Feature complete
  - Code complete
- Milestone III
  - Milestone III release
  - Optimizations
  - Testing and debugging
- Development
  - 6-16 months
    - (1/3 of all features)
- Stabilization
  - 3-8 months

**Milestones**

- Milestone 0
  - Vision statement
  - Specification document
- Project review
  - Implementation plan
- Major Reviews
  - Postmortem document

**Documents and Intermediate Activities**

- Vision statement
- Specification document
- Prototypes
- Design feasibility studies
- Testing strategy
- Schedule
- Implementation plan
- Optimizations
- Testing and debugging
- Internal testing
- Buffer time
- Beta testing
- Buffer time
- Postmortem document
N² Tables Specify Valid Phase, State, or Mode Transitions to Facilitate Deterministic Designs

Transitions are clockwise

<table>
<thead>
<tr>
<th></th>
<th>Launch</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine Pointing</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>Wheel Maneuver</td>
<td>G</td>
<td>G</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>Thruster Maneuver</td>
<td>G</td>
<td>G</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>Delta V</td>
<td>G</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>A</td>
<td>Coarse Pointing</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>Sun Pointing</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>Safe Haven</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
<td>Survival</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

- **Nominal modes**
- **Transition modes**
- **Contingency modes**
- **Autonomous or Ground commanded**
- **Ground commanded only (real time only)**
Establish Embedded Systems and Software Design Principles: Architectures

- Partition functions across multi-processor architecture (such as flight, science, data, power generation, power distribution) to distribute performance, allocate margins, and improve fault protection
- Define software “executive” foundational layer that is common across multi-processor architecture to enable reuse and flexibility
- Develop software using architectural simplicity, table-driven designs, deterministic behavior, and common interfaces to improve verifiability and predictability
- Adopt high-level command sequencing “macro language” for non-software personnel, such as system engineers, to use, typically structured as table-driven templates of commands and parameters, to improve specifiability and verifiability
- Define simple deterministic synchronous control-loop designs with well defined task structures (typically 3-4 levels), static resource allocation (such as no dynamic memory allocation), and predictable timing (such as minimizing interrupts) to improve understandability and verifiability
- Centralize system level autonomy and fault protection and distribute lower level autonomy and protection to appropriate control points to orchestrate system configurations, ensure timely isolations and responses, and support overall safety
- Dedicate pathways for high-speed data (such as from payload instruments to high capacity storage) to separate specialized processing and faults from core functionality (such as payload versus spacecraft)
- Adopt large resource margins (processor, memory, storage, bus) to accommodate contingencies and post-delivery changes
Common Requirements Enable Software Product Lines and Layered Architectures Across Projects

Legend
- Mission-specific
- Common across projects

Mission-Specific requirements

Project 1
- Mission-Specific HW
  Processor & I/O Interfaces
- Mission-Specific SW
  Table-driven configurations, commands
- Common SW
  Functions for Protocols, Fault management, Attitude, Power, Thermal, etc.
  Real-time operating system

Project 2
- Mission-Specific HW
  Processor & I/O Interfaces
- Mission-Specific SW
  Table-driven configurations, commands
- Common SW
  Functions for Protocols, Fault management, Attitude, Power, Thermal, etc.
  Real-time operating system

Project 3
- Mission-Specific HW
  Processor & I/O Interfaces
- Mission-Specific SW
  Table-driven configurations, commands
- Common SW
  Functions for Protocols, Fault management, Attitude, Power, Thermal, etc.
  Real-time operating system

Project 4
- Mission-Specific HW
  Processor & I/O Interfaces
- Mission-Specific SW
  Table-driven configurations, commands
- Common SW
  Functions for Protocols, Fault management, Attitude, Power, Thermal, etc.
  Real-time operating system
Partition Software Functions Across Processors for Performance, Margins, and Fault Protection

Five-processor architecture provides partitioned functions, common executive layer, and growth margins

<table>
<thead>
<tr>
<th>Processor-Specific</th>
<th>Flight SW Functions</th>
<th>Science SW Functions</th>
<th>Data SW Functions</th>
<th>Power Generation SW Functions</th>
<th>Power Distribution SW Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command sequencing</td>
<td>Command execution</td>
<td>Instrument control</td>
<td>Recorder management</td>
<td>Instrumentation</td>
<td>Power conversion loop</td>
</tr>
<tr>
<td>Command execution</td>
<td>Telemetry</td>
<td>Instrument sequencing</td>
<td>Data storage control</td>
<td>Sensor control</td>
<td>Sensor control</td>
</tr>
<tr>
<td>Telemetry</td>
<td>AACS</td>
<td>Instr. data processing</td>
<td>File/byte data protocol</td>
<td>Drive control</td>
<td>Drive control</td>
</tr>
<tr>
<td>Auto navigation</td>
<td>Thermal control</td>
<td>Internal fault protection</td>
<td>Data compression</td>
<td>Coolant loop control</td>
<td>Coolant loop control</td>
</tr>
<tr>
<td>Thermal control</td>
<td>Power coordination</td>
<td>Internal fault protection</td>
<td>Internal fault protection</td>
<td>Time-critical safing</td>
<td>Time-critical safing</td>
</tr>
<tr>
<td>Internal fault protection</td>
<td>System fault protection</td>
<td></td>
<td></td>
<td>Internal fault protection</td>
<td>Internal fault protection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor-Specific</th>
<th>Flight SW Functions</th>
<th>Science SW Functions</th>
<th>Data SW Functions</th>
<th>Power Generation SW Functions</th>
<th>Power Distribution SW Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW Functions</td>
<td>Start-up ROM</td>
<td>Start-up ROM</td>
<td>Start-up ROM</td>
<td>Start-up ROM</td>
<td>Start-up ROM</td>
</tr>
<tr>
<td>Command sequencing</td>
<td>Initialization</td>
<td>Initialization</td>
<td>Initialization</td>
<td>Initialization</td>
<td>Initialization</td>
</tr>
<tr>
<td>Telemetry</td>
<td>Device drivers</td>
<td>Device drivers</td>
<td>Device drivers</td>
<td>Device drivers</td>
<td>Device drivers</td>
</tr>
<tr>
<td>AACS</td>
<td>Real-time O/S</td>
<td>Real-time O/S</td>
<td>Real-time O/S</td>
<td>Real-time O/S</td>
<td>Real-time O/S</td>
</tr>
<tr>
<td>Auto navigation</td>
<td>Time maintenance</td>
<td>Time maintenance</td>
<td>Time maintenance</td>
<td>Time maintenance</td>
<td>Time maintenance</td>
</tr>
<tr>
<td>Thermal control</td>
<td>I/O management</td>
<td>I/O management</td>
<td>I/O management</td>
<td>I/O management</td>
<td>I/O management</td>
</tr>
<tr>
<td>Power coordination</td>
<td>Memory load/dump</td>
<td>Memory load/dump</td>
<td>Memory load/dump</td>
<td>Memory load/dump</td>
<td>Memory load/dump</td>
</tr>
<tr>
<td>Internal fault protection</td>
<td>Task management</td>
<td>Task management</td>
<td>Task management</td>
<td>Task management</td>
<td>Task management</td>
</tr>
<tr>
<td>System fault protection</td>
<td>Shared data control</td>
<td>Shared data control</td>
<td>Shared data control</td>
<td>Shared data control</td>
<td>Shared data control</td>
</tr>
<tr>
<td></td>
<td>Utilities &amp; diagnostics</td>
<td>Utilities &amp; diagnostics</td>
<td>Utilities &amp; diagnostics</td>
<td>Utilities &amp; diagnostics</td>
<td>Utilities &amp; diagnostics</td>
</tr>
</tbody>
</table>

Margins

- Flight: >50%
- Science: >50%
- Data: >50%
- Power Generation: >50%
- Power Distribution: >50%
32% of Software Components are Either Reused or Modified from Previous Systems

- Data from 25 NASA systems
- Component origins: 68.0% new development, 4.6% major revision, 10.3% slight revision, and 17.1% complete reuse without revision
Analyses of Component-Based Software Reuse Shows Favorable Trends for Decreasing Faults

- Data from 25 NASA systems
- Overall difference is statistically significant ($\alpha < .0001$). Number of components (or modules) in each category is: 1629, 205, 300, 820, and 2954, respectively.
Establish Embedded Systems and Software Design Principles: Techniques

- **Flowdown requirements** systematically from project, system (space, ground, launch, etc.), module (spacecraft, mission, etc.), segment (bus, software, etc.), subsystem/build, assembly, etc. to clarify functionality and accountability.

- Identify a manageable number of “**key driving requirements**”, where key is top-down mission-success and driving is bottom-up design-limiting, to prioritize attention and analysis.

- Define user-perspective “**mission threads**” to focus modeling, end-to-end prototyping, and validation.

- Formulate leading indicators to identify high-fault and high-effort system structure and components.

- Specify “**command**” abstractions that define standalone command primitives with pre-conditions, atomic processing, resource constraints (such as timing), and post-conditions (such as data modified) to enable analysis and predictability.

- Define and enforce “**control points**”, such as centralized sequential command queue and explicit data dependency graphs for read/write of data shared across commands and sequences, to facilitate analysis and isolate faults.

- Include built-in self-tests, invariants, and redundant calculations in implementations to help ensure accurate processing and isolate faults.

- Compare executions of system models and software implementations automatically using toolsets to improve verification.

- Apply workflow tools, checklists, **statistical analyses**, **root cause analyses**, and metric dashboards to improve repeatability, visibility, and preventability.
Interactive metric dashboards incorporate a variety of information and features to help developers and managers characterize progress, identify outliers, compare alternatives, evaluate risks, and predict outcomes.
Data-Driven Statistical Analyses Identify Trends, Outliers, and Process Improvements for Cycletimes

- Control chart of metric data from example Six Sigma projects focusing on change request closure cycletime for software components
- Process improvements decreased variances and decreased means

* Series of process improvements instituted and new control limits calculated

Statistically significant improvement in process performance

Data from one system
**Return-on-Investment (ROI) for Software Peer Reviews Ranges from 9:1 to 3800:1 per Project**

- Return-on-investment (ROI) = Net cost avoidance divided by non-recurring cost
- 2621 defects, 257 reviews, 9 systems, 1.5 years
- High ROI drivers
  - Mature and effective processes already in place
  - Significant new scope under development
  - Early lifecycle peer reviews (e.g., requirements phase)
  - Four of the five programs with >80% requirements and design defects had relatively higher ROI
Analyses of Fault Detection Strategies Characterize Fault Types and Effectiveness of Teaming

- Unit-level fault detection strategies for 32 NASA developers and two-person teams
- Six combined testing strategies detected 67% and three individual techniques detected 50% of the software faults on average (35% improvement)
- Highest percentage of software faults detected when there was a combination of either two code readers or a code reader and a functional tester (75%)
- Combined code reading strategies (cc/cf/cs) exceeded all individual techniques

Legend
- c = Code reading by stepwise abstraction
- f = Functional testing using equivalence partitioning and boundary value analysis
- s = Structural testing using 100% statement coverage
- xy = Two-person combination of technique x and y
Predictive Models Identify Leading Indicators of High-Error and High-Effort Components

- Target: Identify error-prone (top 25%) and effort-prone (top 25%) components
- 16 large NASA systems
- 1920 configurations
- Models use metric-driven decision trees and networks
- Optimizations: consistency & completeness
Opportunities for Improvement and Research

- Model-based engineering
- End-to-end capability analyses, tradeoff analyses, sensitivity analyses, and margin assessments
- Reuse
- Return-on-investment analyses for defining, enhancing, and pruning processes