

# Engineering Methodologies Employed for Development of the Hubble Space Telescope On-Board Payload Control Software

**Glenn Foley**  
**Embedded Software Engineer, ICT Centre**  
**5-April-2008**

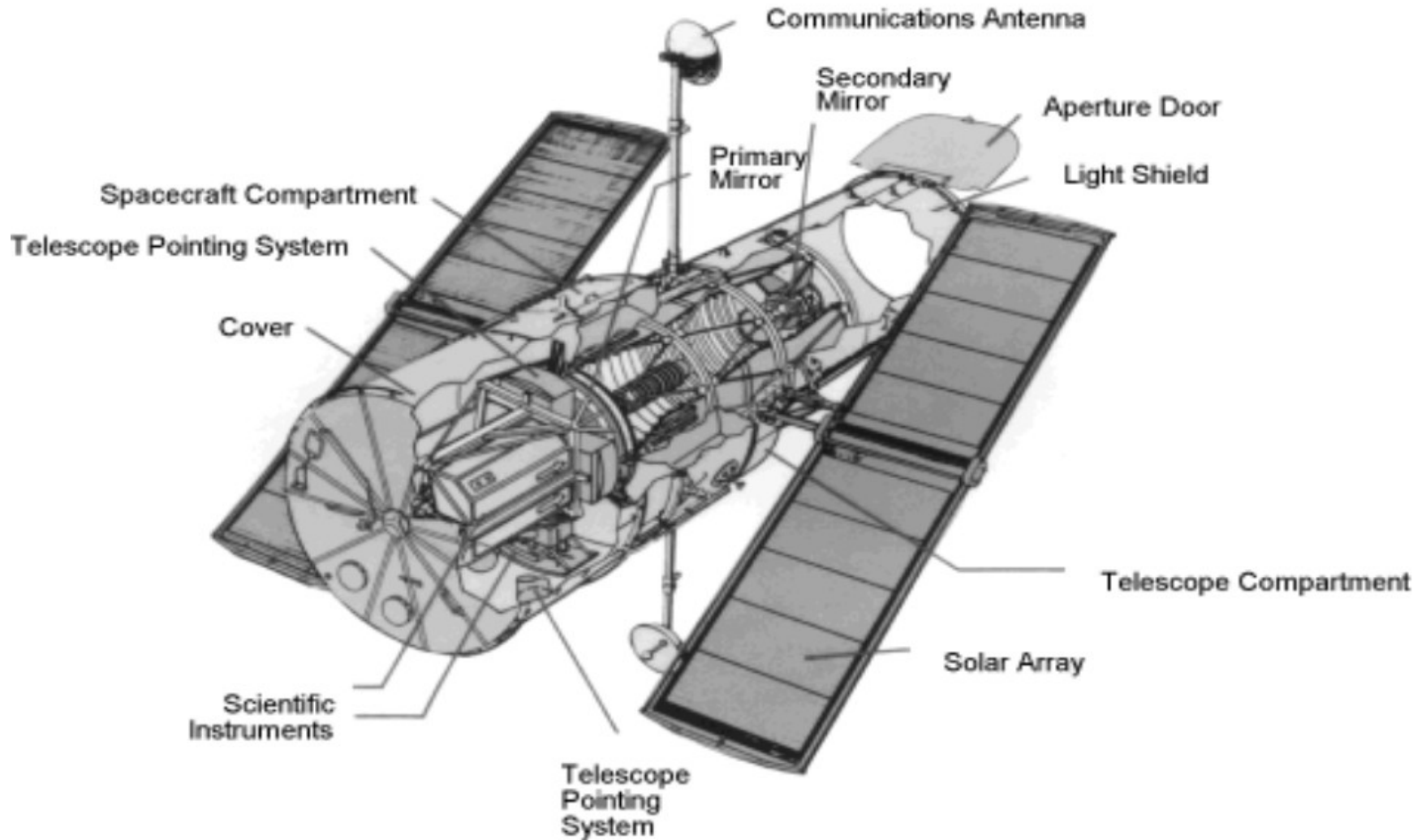
# Overview

- Hubble Space Telescope (HST) spacecraft - background
- HST payload flight software development methodology
- Applicability to Sensor Networks Team

# HST spacecraft - background

- Mission objectives
- Major functional subsystems
  - Support systems module
  - Science instrument control and data handling subsystem
- Components and responsibilities of payload flight software
  - Execute astronomical observations
  - Assure health and safety of subsystem components
  - Process realtime and stored commands
  - Generate contributions to science and engineering data streams
- Major players: scientists, engineers, and spacecraft operators
- My role

# HST Spacecraft Configuration



# Background to software development process

- Motivation for rigour
  - Limited window of visibility in which to observe or attempt to troubleshoot a problem on-orbit
  - Desire to maximise efficient use of the spacecraft
  - Risk of permanently damaging an international resource
  - Avoid at all costs public humiliation and the scorn of the taxpayer
- Methodology not NASA-wide, but carried over from that used by IBM Federal Systems Division during initial implementation of the HST central payload control computer (NSSC-I) flight software
- This methodology has been utilised virtually unchanged for 20+ years - NSSC-I, then five later-generation follow-on scientific instruments (last two due for launch & installation this August)

# Software "Quality," "Robustness," and "Reliability"

- Definitions and metrics innumerable, but conspicuous by their absence
- "Homework Assignment Method" - (potential to) demonstrate minimum required functionality (once) - "Dancing Bear Syndrome"
- "Old School" - these are "tested" into the product just before delivery
  - A flaw discovered at this point may have repercussions deep back into the basic design of the system, requiring either:
    - Much wasted effort to go back and re-evaluate basic system structure, or (more commonly):
    - A work-around which is "ugly" and may adversely affect system performance (a kludge)
- "Enlightened Approach" - these are side-effects of the development process
  - Importance of testing is not minimised but rather is elaborated upon
  - Testing is the LAST line of defence, not the ONLY line of defence

# Overview of phases of the software development lifecycle

- Requirements definition
- Preliminary design
- Detailed design
- Coding
- Unit testing
- Component/Integration/System/Acceptance Testing
- Documentation and peer review (throughout)

# The lifecycle phases in detail (1 of 6)

- Requirements specification

- Avoid scenario of "You guys start coding, I'll go find out what they want!!"
- Problem must be clearly and unambiguously stated
- Any constraints about the pre-existing framework, context, or physical environment in which the solution must function are addressed
- Except where there are existing interfaces to respect (or other special cases), this says WHAT to do, not HOW to do it

# The lifecycle phases in detail (2 of 6)

- Requirements specification (con't)
  - It is important to use this as a first opportunity for both the CUSTOMER and the DEVELOPER to THINK about what has to be done (and ultimately commit to it)
  - In reality, there is iteration and negotiation here, but ideally after this stage, the developer has ALL of the information required to implement the system
  - Requirements stated at too high of a level are USELESS and an empty gesture - "the devil is in the detail"
  - On HST, had to assemble and negotiate often contradictory requirements coming from several engineering and scientific disciplines
  - Need for "rapid prototyping" is NEVER an excuse for poor requirements definition

# The lifecycle phases in detail (3 of 6)

- **Preliminary design**

- Analyse implementation options and risks
- Formulate a general context and software architecture upon which the requirements will be implemented
- Allocating the requirements to individual software functions within that defined architecture
- Verify assumptions about interfaces and what capabilities are in place to utilise

- **Detailed design**

- The logic of those functions identified in preliminary design is expanded to the lowest level via program design language (PDL) or pseudo-code

- **Code development**

- Implementation of the PDL in proper compilable computer code

# The lifecycle phases in detail (4 of 6)

- Unit test - development

- Detects general logic errors with more discrimination than is allowed in code inspection
- Offers more insight into software behaviour and better manipulation of target context than will be available in later test phases
- Exercise ALL paths through the code
- Verify nominal cases and boundary conditions of all equations and expressions
- Verify "start-up" and "shut-down" conditions where applicable
- Usually performed per source code file, per single or grouped functions
- Typically employs driver program with simulation of target environment (e.g. in-circuit emulator [ICE])
- Specify initial conditions, inputs, outputs
- All behaviour must be predictable and quantifiable
- ANY change to a source code file INVALIDATES a previously successfully executed unit test

# The Code Inspection - rules and roles (1 of 4)

- **Purpose - check for:**
  - Fulfillment of applicable requirements
  - Faithful implementation of design
  - Logical correctness, appropriateness, thoroughness, and elegance
  - Clarity of documentation
  - Adherence to standards
- **Code Inspection also:**
  - Aids in detection of errors by introduction of fresh perspective from other team members
  - Increases and diversifies product expertise within team (cross-training)
- **Players**
  - Reader - leads group through code inspection package and provides background where necessary
  - Moderator - records action items on Discrepancy Report form
  - Author - present ONLY to answer questions
    - The author is NEVER the reader - the idea being if independent reader can't figure out what product is doing, this indicates a problem in itself
  - Audience (inspectors) - other members of S/W development team (OCCASIONALLY special invited guests)

# The Code Inspection - rules and roles (2 of 4)

- Author's responsibilities

- Schedule and coordinate meeting and facilities
- Distribute **HAND DELIVERED HARD COPIES** of code inspection package **AT LEAST 2** full business days before scheduled inspection (otherwise inspection must be rescheduled)

- **ALL Code Inspection attendees are EXPECTED to spend SEVERAL HOURS reviewing package**

- Otherwise this phase is an empty gesture and there is no value added
- This is planned into their workload as part of their duties
  - It is **NEVER** acceptable to "have a quick browse" of the package just before the Inspection
- Moderator records statistics of review time spent by each participant on the Code Inspection Discrepancy Report form

# The Code Inspection - rules and roles (3 of 4)

- Contents of Code Inspection package (no late addenda)
  - Meeting announcement
  - List of package contents
  - Brief background and overview
  - Relevant requirements and design references
  - Indented calling tree of functions
  - Source code, line-numbered (MUST provide proof of clean-compile)
  - Assembler/compiler listings, if relevant
  - Unit test plan
    - Includes description of each test case with setups/inputs and expected results/verifiers
    - When given ANY line of executable code, author should IMMEDIATELY be able to identify AT LEAST one test case that exercises that path VERIFIABLY

# The Code Inspection - rules and roles (4 of 4)

- After code inspection, moderator makes copy of Discrepancy Report for team leader; original goes to author for disposition/correction
- Generally, the author has one week to reconcile all actions and obtain approval from team leader
- After this, unit testing can be performed
- Upon the successful completion of unit testing, a "promotion request" is submitted to the team leader

# Unit test - execution and source code promotion

- After (and ONLY after) the unit test has been successfully completed, the subject source code (and any associated updates to build procedures) is "promoted" and available FOR THE FIRST TIME to other developers and testers for integration with the rest of the system

# Testing - general philosophy

- The paradox: to fail and to succeed
- The goal: to find problems as early in the development cycle as possible
- "Flavours" of testing
  - Unit - white box
  - System - grey box
  - Acceptance test (to requirements) - black box (ideally "IV&V")
- "Usage" does NOT equate to "testing"
  - Gives SOME confidence in nominal performance
  - Is not a systematic approach - "hit and miss" in anomaly detection
  - Euphemistically "burn-in time" or "beta testing"
  - Puts burden on user to discover errors - not desirable

# The lifecycle phases in detail (5 of 6)

- **System Test (Component Test, Integration Test, Acceptance Test, ...) Development & Execution**
  - Continually bring more pieces of the software together
  - Operate in a context continually approaching the target environment
  - Verify interactions **AMONG** the software components and with the product's interfaces to the target environment
  - Elements of final acceptance testing
    - Special tests
    - Baseline/regression testing
    - Stress testing
    - "Negative" testing (non-nominal scenarios)
  - Roles in formal acceptance testing
    - Test conductor
    - Test operator
    - Test plan document, test procedures (scripts), and deviation forms
    - Results (electronically logged and hard copy) archived
    - Quality assurance officer (QA)

# The lifecycle phases in detail (6 of 6)

- User documentation
  - Occurs in parallel with all other steps
  - Should require only "glue and polish" by end of development cycle
  - Strictly speaking, only the last of a suite of documentation including (but not limited to) requirements and (preliminary and detailed) design documents, interface specifications, and test plans for each level

# Applicability to Sensor Networks Team

- Steps of a good software development process are scalable by project size, schedule, number of team members, new product or continuing development
- Judging by specific experiences in AS Lab as well as general opinions and attitudes at ICT Centre Conference, the time is upon us for CSIRO software development to raise its game
- Rigorous software development methodology does not guarantee defect-free code, but allows greater confidence in the product ("quality," "robustness," and "reliability") and minimises latent "time bombs"

# More rigorous development process doesn't mean more overall time spent

- "Pay me now or pay me later"
- "All-or-nothing" fallacy
  - "Test & documentation" - wild-eyed anxiety
  - "Rapid prototype" - foolish pride
- The false economy
  - Manpower to debug & fix
  - Lost time of troubleshooters
  - Slipped milestones for "customers"
  - Increased stress
  - "Insecurity" - the remaining "time bombs" can manifest at any time
- Good documentation and test require only marginally increased effort once you:
  - Get in the habit of doing it
  - Develop a "rapport" with your product

# Specific relevance NOW

- **Peer review - especially during code development**
  - Software development is ALWAYS a team activity - your work WILL impact others, sooner or later, for better or worse
  - Assure code "does the right things and does things right"
  - Conformance with S/W & H/W interface requirements
  - Adherence to standards
  - Assure adequate in-line documentation and commentary
- **Test definition at many levels**
  - Remember: "usage" != "testing"
  - Regression: did my "one little change" break anything?
  - Don't minimise the importance of diagnostic information during development and troubleshooting, even if not relevant in deployment
- **Preliminary design peer review**
  - Overview of proposed application flow
  - Advertise assumptions about what system capabilities/resources are available; allow shortcomings to be addressed in parallel, not serially

# Conclusion

- My earliest impressions at CSIRO come from the ICT Centre Conference SWEng Special Interest Session - CSIRO SWEngs get the relevance of improved development processes
- In the day when CSIRO did "one time then move on" research, this do & dispose philosophy was sufficient - anything more was seen as a "no added benefit" effort - but in days of increasing justification of relevance where re-usability becomes an issue, software quality becomes more critical
- I'm available to help analyse your current situation for areas which might benefit from a more rigorous development process, or for direct support in catering and implementing these procedures to meet your specific needs

**ICT Centre/Autonomous  
Systems Laboratory**

Glenn Foley  
Embedded Software Engineer

Phone: 07 3327 4111  
Email: [Glenn.Foley@csiro.au](mailto:Glenn.Foley@csiro.au)  
Web: [www.csiro.au](http://www.csiro.au)

[www.csiro.au](http://www.csiro.au)

**Thank you**

**Contact Us**

Phone: 1300 363 400 or +61 3 9545 2176  
Email: [enquiries@csiro.au](mailto:enquiries@csiro.au) Web: [www.csiro.au](http://www.csiro.au)

