# Basic Embedded Software Engineering: The Software Lifecycle

# Learning Objectives

Based on requirements specifications, develop

- high-level design specs with tests
- detailed design specs with tests
- other testing as needed

Understand how to provide and verify traceability across the V throughout the process

Learn how to perform design reviews and code walkthroughs

# Lecture Topics

Software lifecycle V-model overview

Implementing the V

- Requirements specification
- High-level module and test design (state-based, control-flow-based)
- Detailed module and test design
- Code Design Reviews
- Unit testing
- Integration testing

*Portions derived from* ***Software Engineering: A Practitioner's Approach,*** *6th Edition, Roger S. Pressman, McGraw-Hill*

# Software Engineering Definitions

"The establishment and use of sound engineering principles in order to ***obtain economically*** software that is ***reliable*** and ***works efficiently*** on ***real machines.***" [Fritz Bauer 1969]

- *Goals*

"The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software; that is, the application of engineering to software." [IEEE 1993]

- *Methods*

How can we make software *correct enough* without making the company go bankrupt?

- Need to be able to develop software efficiently – *this requires a set of sound processes* built into the company

# Capability Maturity Model (CMM)

CMM developed by CMU's Software Engineering Institute (SEI) to define maturity of software development processes

**Level 1: Initial.**

- Ad hoc and occasionally even chaotic.
- Few processes defined.
- Success often requires "heroes". Overtime, weekends, no vacation.

**Level 2: Repeatable.**

- Basic project management processes track cost, schedule and functionality.

**Level 3: Defined.**

- Processes for management and engineering are documented, standardized, and integrated into overall organization software process.

**Level 4: Managed.**

- Software process and product quality are measured quantitatively.
- Metrics are used to adjust process.

**Level 5: Optimizing.**

- Continuous process improvement is performed.
- Innovative ideas are applied and evaluated using metrics.

# Real World: CIMM

What could possibly be worse than Level 1 (*ad hoc* and *chaotic*)?

Capability Im-Maturity Model, developed by Capt. Tom Schorsch, U.S. Air Force

The CIMM adds maturity levels below Level 1

**Level 0: Negligent**

- "Indifference:
  - Failure to allow successful development process to succeed.
  - All problems are perceived to be technical problems.
  - Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process.
  - Reliance on silver pellets."

**Level -1: Obstructive**

- "Counterproductive:
  - Counterproductive processes are imposed.
  - Processes are rigidly defined and adherence to the form is stressed.
  - Ritualistic ceremonies abound.
  - Collective management precludes assigning responsibility.
  - Status quo forever."

# Real World: CIMM

**Level -2: Contemptuous**

- "Arrogance:
  - Disregard for good software engineering institutionalized.
  - Complete schism between software development activities and software process improvement activities.
  - Complete lack of a training program."

**Level -3: Undermining**

- "Sabotage:
  - Total neglect of own charter,
  - conscious discrediting of peer organizations software process improvement efforts.
  - Rewarding failure and poor performance."

http://www.stsc.hill.af.mil/crosstalk/1996/11/xt96d11h.asp

Finkelstein, "A Software Process Immaturity Model," *ACM SIGSOFT, Software Engineering Notes,* Vol. 17, No. 4, October 1992, pp. 22-23.

# Good Enough Software

How do we make software *correct enough* without going bankrupt?

- Need to be able to develop (and test) software efficiently

Follow a good plan

- Start with customer requirements
- Design architectures to define the building blocks of the systems (tasks, modules, etc.)
- Add missing requirements
  - Fault detection, management and logging
  - Real-time issues
  - Compliance to a firmware standards manual
  - Fail-safes
- Create detailed design
- Implement the code, following a good development process
  - Perform frequent design and code reviews
  - Perform frequent testing (unit and system testing, preferably automated)
  - Use revision control to manage changes
- Perform post-mortems to improve development process

# Software Lifecycle Concepts

Coding is the most visible part of a software development process but is not the only one

Before we can code, we must know

- What must the code do? *Requirements specification*
- How will the code be structured? *Design specification*
  - *(only at this point can we start writing code)*

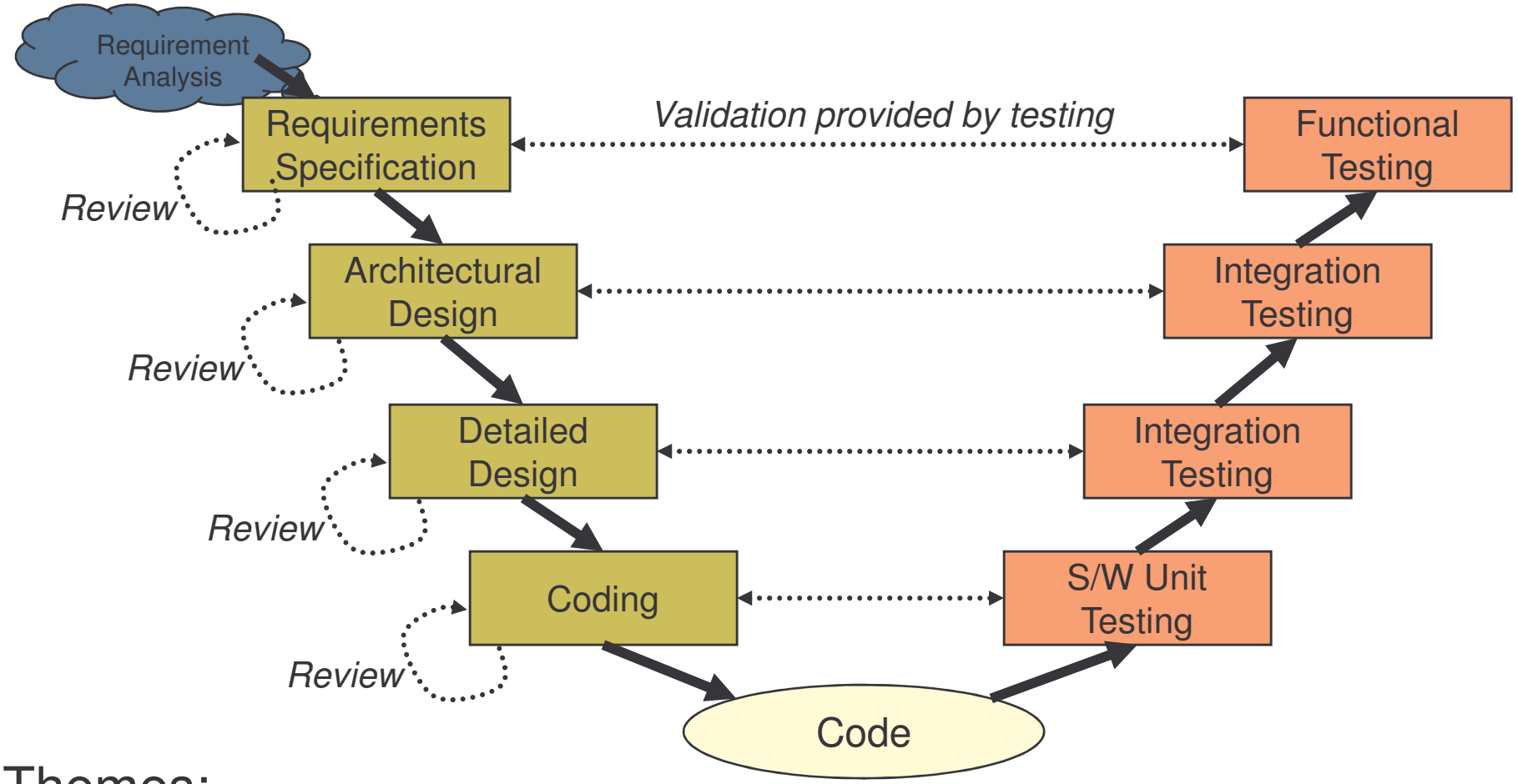
How will we know if the code works? *Test plan*

- Best performed when defining requirements

The software will likely be enhanced over time

- Corrections, adaptations, enhancements & preventive maintenance

# V Model Overview




Themes:

- Link front and back ends of life-cycle for efficiency
- Provide “traceability” to ensure nothing falls through the cracks

# Implementing the V

1. Requirements specification
   - Define what the system must do
2. Architectural (high-level) module and test design (state-based, control-flow-based)
3. Detailed module and test design
4. Coding and Code Inspections

# 1. Requirements Spec. & Validation Plan

Result of Requirements Analysis

Should contain:

- *Introduction* with goals and objectives of system
- *Description of problem* to solve
- *Functional description*
  - provides a “processing narrative” per function
  - lists and justifies design constraints
  - explains performance requirements
- *Behavioral description* shows how system reacts to internal or external events and situations
  - State-based behavior
  - General control flow
  - General data flow
- *Validation criteria*
  - tell us how we can decide that a system is acceptable. (*Are we there yet?*)
  - is the foundation for a validation test plan
- *Bibliography and Appendix* refer to all documents related to project and provide supplementary information

# Specification Guidelines

- Use a layered format that provides increasing detail with Deeper levels
- Use consistent graphical notation and apply textual terms consistently (stay away from aliases)
- Be sure to define all acronyms
- Write in a simple, unambiguous style
- Avoid vague terms (sometimes, often, etc.)
- Beware of incomplete lists
- Avoid passive voice which hides responsibility
- Avoid dangling pronouns
- Beware of implied requirements (e.g. numerical base)

# Specification Review

High-level review to ensure specification is

- complete
- consistent
- accurate

Difficult to “test” specification, so need to consider and discuss requirements interactively

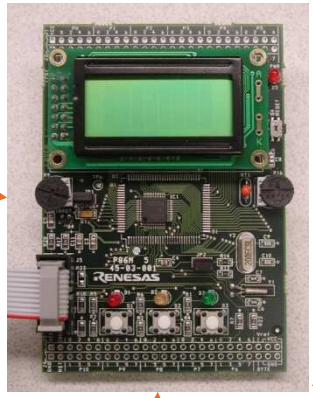
# Chart Maker General Requirements

SONAR


*Depth*
*NMEA 0183*
*4800 baud*
*RS232*

*Lat/Lon Position*
*NMEA 0183*
*4800 baud*
*RS232*

GPS




*SPI*

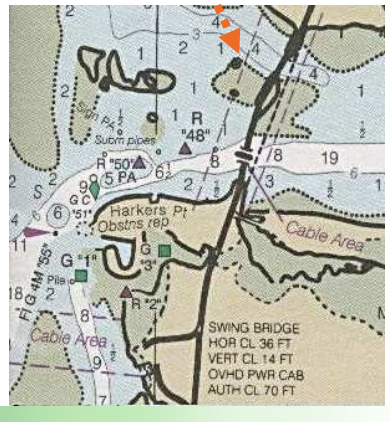
DataFlash Memory

*Download*
*RS232*

*RS232 + USB*

*Debugger*

*Off-line Data Processing*




Three operating modes

- Standby
- Record
  - Get depth information from Sonar via RS232
  - Get position information from GPS via RS232
  - Store most recent depth from each position in DataFlash
- Download
  - Download depth and position information from DataFlash

# Chart Maker SW&HW Requirements Specification

1. Input data
    1. Accept NMEA-0183 format data sentences from GPS receiver and extract time, date, position, speed over ground and bearing information while discarding sentences with CRC errors
    2. Accept NMEA-0183 format data sentences from depth sounder (SONAR) and extract depth, battery voltage, speed through water, and water temperature information while discarding sentences with CRC errors
2. Data storage
    1. Every two seconds, record most recent extracted data to data flash memory in the following human readable text form:
        1. Depth data shall be stored as DXX.X, measured in feet
        2. Latitude position data shall be stored as [N|S]XX.YYYY, with XX measuring degrees and YYYY measuring thousandths of minutes
        3. Longitude position data shall be stored as [E|W]XXX.YYYY, with XX measuring degrees and YYYY measuring thousandths of minutes
        4. Battery voltage shall be stored as BXX.X, measured in volts.
        5. etc.

# Chart Maker SW&HW Requirements Specification

3. User interfaces
    1. Primary user interface
        1. The primary user interface shall have a push-button momentary switch and a text LCD.
        2. Pressing the switch shall toggle between enabling (RECORD) and disabling (STANDBY) data recording.
        3. Disabling recording shall allow power to be turned off after no more than one second without any data loss.
        4. The LCD shall display amount of data flash memory used, and certain data received from the GPS (position, speed over ground, time) and depth sounder (depth, speed through water, battery voltage)
    2. Secondary (remote) user interface
        1. This interface shall be accessed through a serial port (19200 baud, 8 data bits, no parity, one stop bit)
        2. This interface shall provide the following functions to the user
            1. Erase all data flash memory contents
            2. Download all data flash memory contents
4. Robustness
    1. A watchdog timer shall be used to detect an out-of-control program within 100 ms.
    2. If data is detected from both NMEA sources, the device will automatically switch to record mode.
5. Power
    1. Power shall be supplied to the chart maker at 12 V.
    2. Maximum current consumption in RECORD and STANDBY modes shall be no more than 100 mA

# Implementing the V

1. Requirements specification
2. Architectural (high-level) module and test design (state-based, control-flow-based)
   - Define big-picture view of what pieces will exist, how they will fit together, and how they will be tested
3. Detailed module and test design
4. Coding and Code Inspections

# 2. Architectural (High-Level) Design

Architecture defines the structure of the system

- Components
- Externally visible properties of components
- Relationships among components

Architecture is a representation which lets the designer…

- Analyze the design's effectiveness in meeting requirements
- Consider alternative architectures early
- Reduce down-stream implementation risks

Architecture matters because…

- It's small and simple enough to fit into a single person's brain (as opposed to comprehending the entire program's source code)
- It gives stakeholders a way to describe and therefore discuss the system

# Architectural Styles for Software

Several general styles or patterns exist

Each describes

- *Components* which perform some function
- *Connectors* with which components communicate, coordinate and cooperate
- *Constraints* which define how components fit together
- *Semantic Models* which allow designer to determine system behavior by composing component and connector behavior

# Architectural Patterns

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

- *operating system process management* pattern
- *task scheduler* pattern

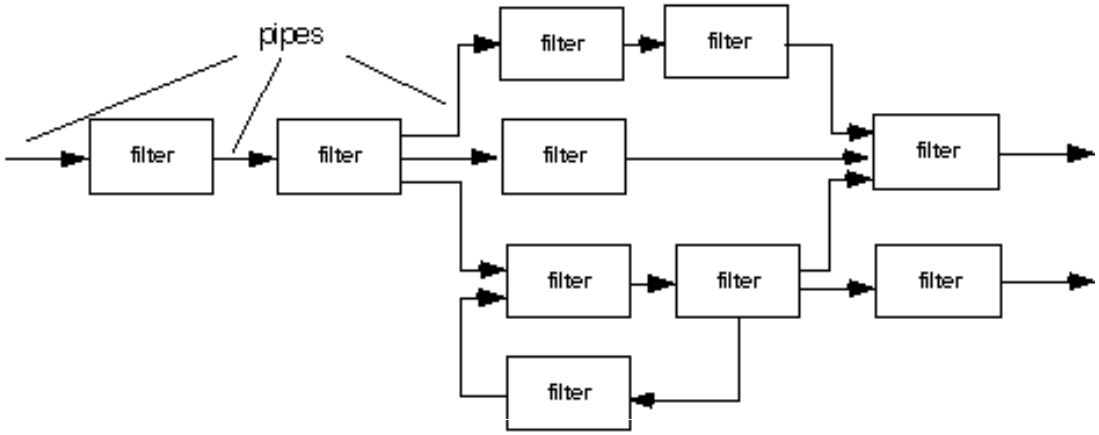
Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:

- a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an *application level persistence* pattern that builds persistence features into the application architecture

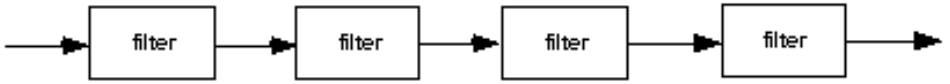
Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment

- A *broker* acts as a 'middle-man' between the client component and a server component.

# Data-Flow Architectures



(a) pipes and filters



(b) batch sequential

Transforms input data into output data

Pipes and filters

- Pipes communicate data
- Filters transform data

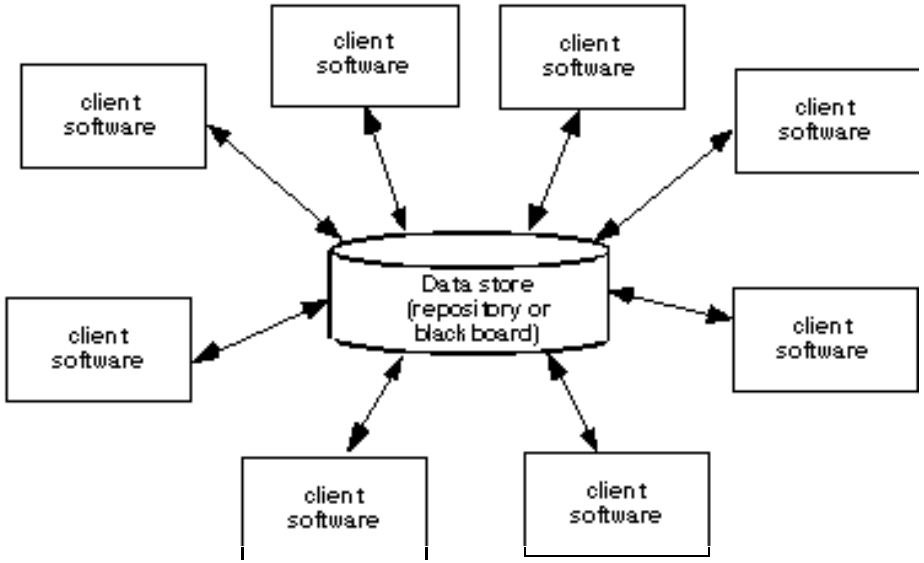
Filters operate independently of each other

Commonly used for signal processing

# Object-Oriented Architectures

- Components contain data and operations
- Message passing used to communicate and coordinate among components
- Many approaches possible

# Data-Centered Architectures




Central data repository is core component

Other components (clients) modify, add, or delete data.

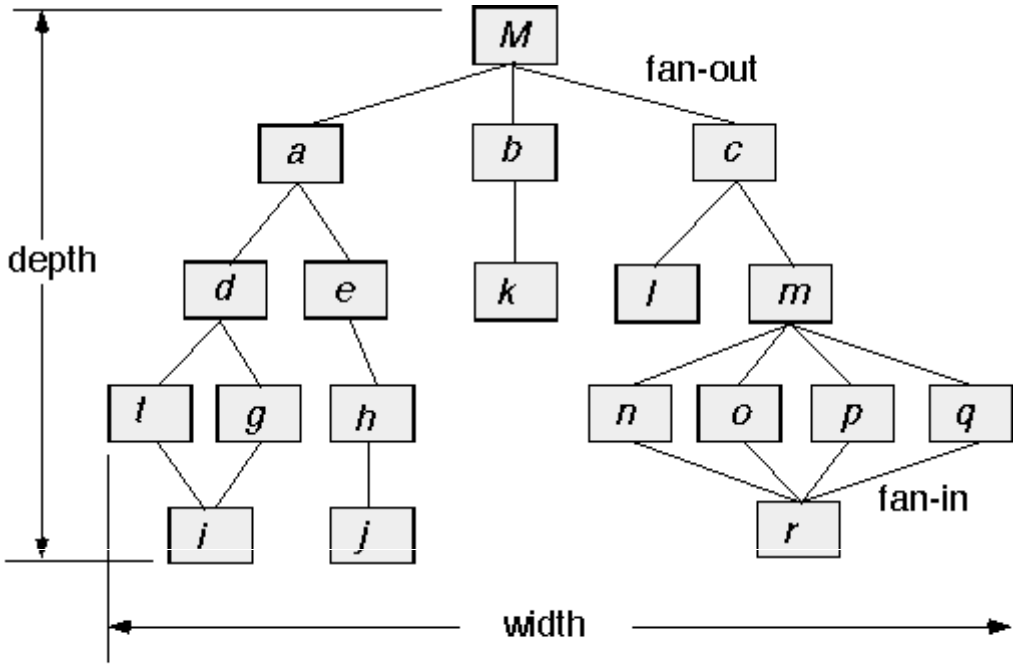
- Each client is independent process

Repository style

- Passive: client software gets data regardless of data changes or activities of other clients
- Blackboard: repository sends change notifications to clients

Easy to add client components without affecting other clients

# Call and Return Architectures




Represents hierarchical nesting of control in program

- "main calls get_data which calls convert_input…"

Scalability and modification straightforward

Sub-types of Call and Return architectures

- main program/subprogram: single processor
- remote procedure call: subprograms may be distributed across remote processors

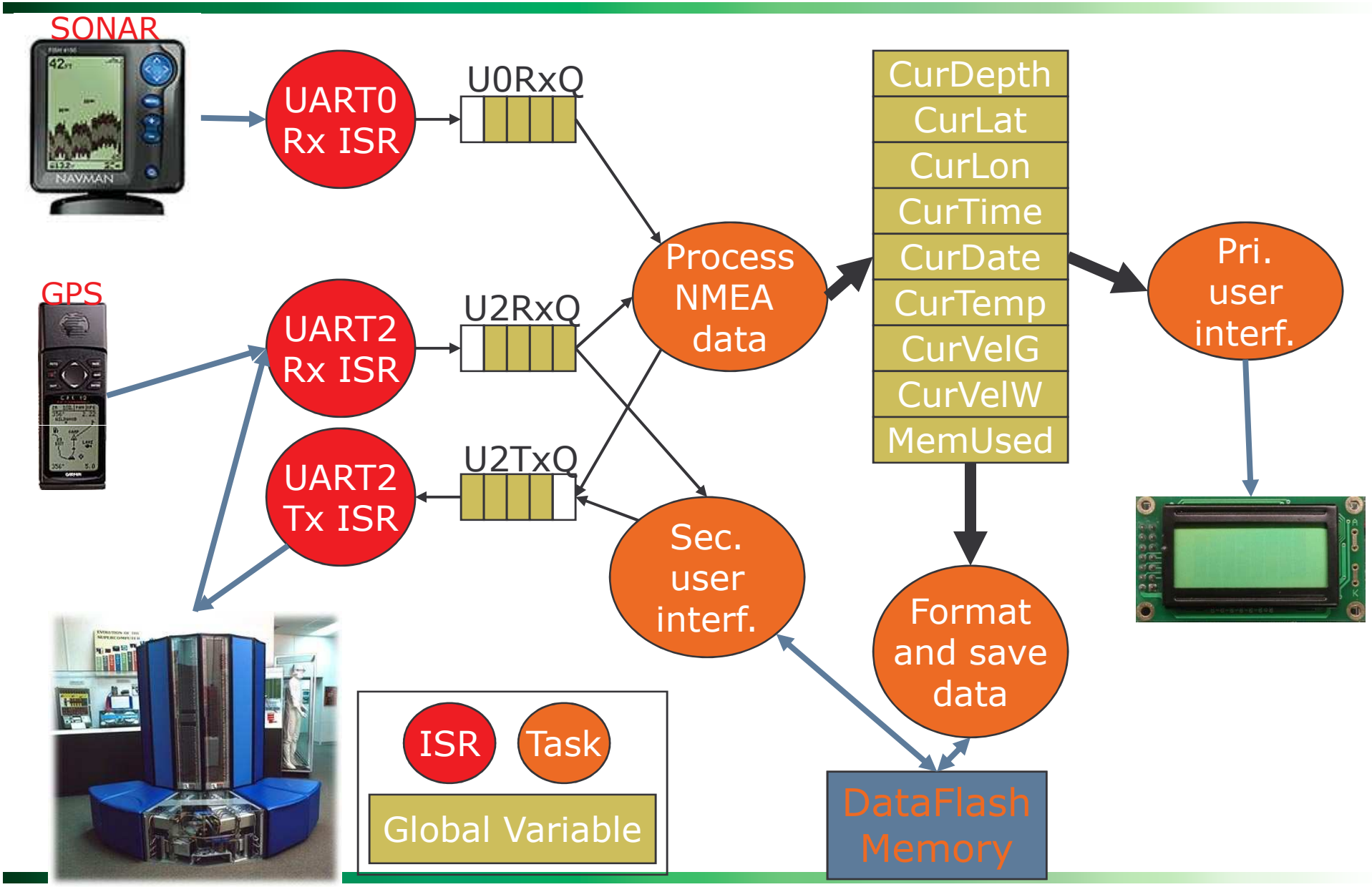
# Data Design

Important data should be clearly specified and described

Principles

- Use Data Dictionary to define data and program design
- Defer low-level decisions until late in design process
- Use abstraction and information hiding to separate logical and physical views of system
- Develop a library of useful data structures and operations for reuse

# Chart Maker Thread and Data Architecture



# High-Level Design for Chart Maker

1. Concurrency
    1. Interrupt service routines shall be used to service UART and timer interrupts.
    2. A non-preemptive run-to-completion scheduler shall be used to execute threads.
2. System initialization
    1. MCU
        1. System clock
        2. Scheduler tick timer
        3. UARTs
        4. Watchdog timer
    2. LCD
        1. The I/O pins for the LCD interface shall be initialized
        2. The LCD shall be reset.
        3. The LCD shall display a start-up screen “….”
    3. Data flash
        1. The I/O pins for the SPI interface with the data flash shall be initialized.
        2. The data flash shall be reset.
        3. The data flash shall be queried to determine if it is present, displaying “Flash OK” on the LCD if it returns the correct response or “Flash Failure” otherwise

# High-Level Design for Chart Maker

3. Interrupt service routines
    1. GPS UART
        1. An ISR u0_rx_isr shall service the UART's receive character interrupt.
        2. The ISR will enqueue NMEA-0183 sentences of type $GPRMC in a circular buffer U0RxQ.
        3. The ISR will complete execution before the next character is received.
        4. The ISR shall increment a global variable sonar_sentence_available after receiving a complete NMEA-0183 sentence of type $GPRMC
    2. Depth Finder/Download UART
        1. An ISR u2_rx_isr shall service the UART's receive character interrupt.
            1. When in DOWNLOAD mode (secondary UI is active), the ISR shall place the received data in a circular buffer U2RxQ before the next character is received.
            2. When not in DOWNLOAD mode (secondary UI is inactive),
                1. The ISR will enqueue NMEA-0183 sentences of type $SDDBT, $VWVHW or $YXXDR in a circular buffer.
                2. The ISR will complete execution before the next character is received.
                3. The ISR shall increment a global variable sonar_sentence_available after receiving a complete NMEA-0183 sentence of type $SDDBT, $VWVHW or $YXXDR
        2. An ISR u2_tx_isr shall service the UART's transmit character interrupt by removing a character from the circular buffer if it is not empty.
    3. Tick Timer
        1. An ISR shall service the tick timer overflow interrupt. This is implemented run-to-completion scheduler.
        2. This ISR shall run at a frequency of TICK_FREQUENCY Hz.

# High-Level Design for Chart Maker

4. Threads
    1. Process_NMEA_Data
        1. This thread shall decode incoming data from both UART
        2. This thread shall detect corrupted sentences by computing a checksum on received data and comparing it with the received checksums. Corrupted sentences shall be discarded without additional processing.
        3. This thread shall decode valid sentences and update relevant global variables.
        4. This thread shall run often enough so that UART receive data buffers can never overflow
    2. Format_and_Save_Data
        1. This thread shall run approximately every two seconds (+/- 10%).
        2. This thread shall be disabled when the system is in STANDBY mode.
        3. This thread shall be enabled when the system is in RECORD mode,
        4. This thread shall read all relevant variables to be recorded, format them into ASCII text according to RS 2.1 and write them to the data flash SRAM buffer via the SPI interface.
        5. When the buffer is nearly full (it is possible for the next thread invocation to overflow the buffer), the thread shall send the data flash controller the SPI command to write the buffer contents to the current flash page.

# High-Level Design for Chart Maker

4. Threads
    3. Primary_User_Interface
        1. This thread shall run ten times per second.
        2. This thread shall read the switch and debounce it, requiring it to be active three times in a row before responding to it.
        3. This thread shall use the debounced switch value to toggle the system between RECORD and STANDBY modes.
        4. Upon switching to RECORD mode, this thread shall enable the Format_and_Save_Data task.
        5. Upon switching to STANDBY mode, this thread shall diable the Format_and_Save_Data task.
    4. Secondary User Interface
        1. etc.

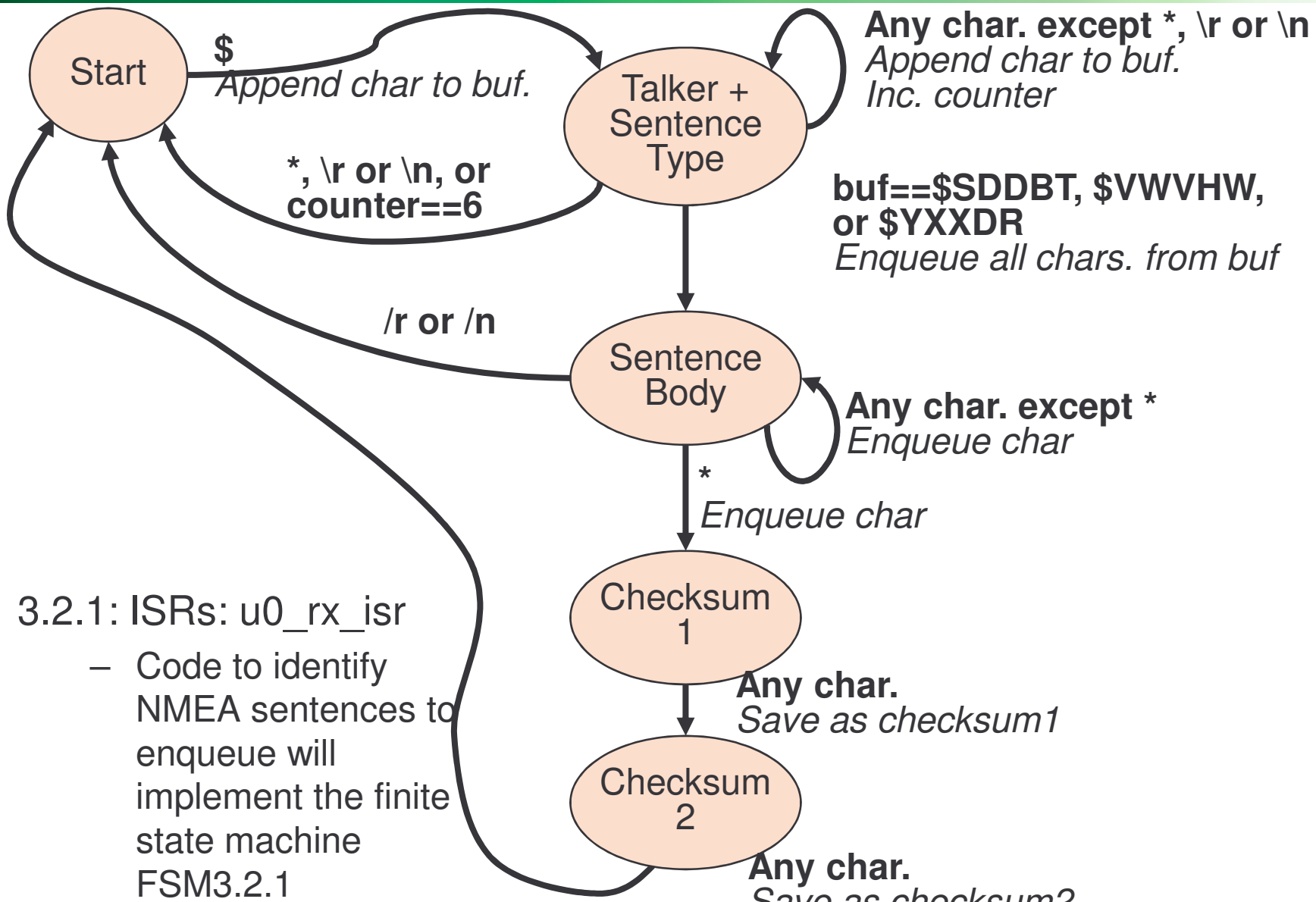
# High-Level Design for Chart Maker

5. Global Data
    1. Logged Data Variables
        1. The CurDepth variable shall be a float, measured in feet, and shall hold the most recently received depth information.
        2. The CurVelG variable shall be a float, measured in knots, and shall hold the most recently received velocity over ground speed information.
        3. etc.
    2. Communication Queues
        1. U0RxQ
            1. This queue shall be implemented as a circular buffer sized large enough to handle all possible incoming NMEA-0183 data while being serviced every two seconds.
            2. Data shall be enqueued by the UART 0 receive character ISR.
            3. Data shall be dequeued by the Process_NMEA_Data thread.
            4. If data arrives when the queue is full, that data should be discarded.
        2. etc.

# Implementing the V

1. Requirements specification
2. Architectural (high-level) module and test design (state-based, control-flow-based)
3. Detailed module and test design
   - Now we design the software, using flow charts or finite state machines
   - Also define module tests (discussed at end of lecture)
4. Coding and Code Inspections

# Example: Detailed Design for Chart Maker




3.2.1: ISRs: u0_rx_isr

- Code to identify NMEA sentences to enqueue will implement the finite state machine FSM3.2.1

# Example: Detailed Design for Chart Maker

4.1: Threads:
Process_NMEA_Data

- Code to process received NMEA data will implement the flowchart FC4.1.

# 4. Coding and Code Inspections

Coding driven directly by Detailed Design Specification

Use a version control system while developing the code

Follow a coding standard

- Eliminate stylistic variations which make understanding code more difficult
- Avoid known questionable practices
- Spell out best practices to make them easier to follow

Perform code reviews

Test effectively

- Automation
- Regression testing

# Version Control

Use a version control system

- "Time machine" for coding
- Procedure
  - Check out code module
  - Do your programming and testing
  - Update the module (in case anyone else has changed it)
  - Check code module into the repository when the code works (commit)
- Examples: CVS, RCS, SourceSafe

Use regression tests

- Verify the code works before checking it into the repository
  - Use test cases which exercise as much of the code as possible
  - Include
- Automation extremely useful
  - Shell scripts, etc.
  - Can even automate finding the bugs sometimes

# Peer Code Review

Inspect the code before testing it

Extensive positive industry results from code inspections

- IBM removed 82% of bugs
- 9 hours saved by finding each defect
- For AT&T quality rose by 1000% and productivity by 14%

Finds bugs which testing often misses

- 80% of the errors detected by HP's inspections were unlikely to be caught by testing
- HP, Shell Research, Bell Northern, AT&T: inspections 20-30x more efficient than testing

# Types of Peer Code Review (1)

- Code inspection
  - IBM, Michael Fagan, 1976
  - Reader (not author) explains what code seems to do
  - Formal, heavy-weight process for reviewing code
    - Preparation stage involves participants reading code
    - Measure where all bugs are, classify them, how quickly bugs are found, etc.
  - Author fixes all problems after review
- Over-the-shoulder review
  - Author explains to reviewer what changed and why
    - Very easy to implement
  - Fix small small problems immediately, large problems afterwards
  - Very light-weight process has disadvantages
    - No way for manager to ensure all changes are reviewed
    - Easy for author to miss a change
    - No way to track nature of bugs, how efficient this process is
    - No way for reviewer to verify bugs were fixed

# Types of Peer Code Review (2)

E-mail pass-around

- Author emails files/change logs to reviewers for discussion (by email)
- Can use version control system to determine automatically what has changed
- Advantages
  - Easy to implement, easy to add reviewers
  - Simple scheduling
- Disadvantages
  - Scales badly – gets hard to track threads and changes for large projects
  - Email slows down communication
  - Doesn't ensure all changes are reviewed
  - Doesn't ensure all fixes are made and are correct

Tool-assisted reviews

- Use special review tool
  - Automatically gathers files, integrated with version control system
  - Provides combined display of file differences, comments and defects
  - Automatically collects metrics (e.g. lines inspected per hour, defects found per hour, defects found per thousand lines of code)
  - Allows monitoring of which changes have been reviewd, and which bugs have been fixed
  - Presents efficient specialized interfaces for coders, reviewers and managers

# Types of Peer Code Review (3)

Pair-programming

- Two developers sit at one workstation
- Only one types at a time
- Advantages
  - Very effective at finding bugs and transferring knowledge
  - Reviewer is very familiar with code, given long-term involvement with development
- Disadvantage
  - Reviewer may be too involved to see big-picture issues (may need other form of review)
  - Takes a lot of time (two whole programmers, full-time)
  - Doesn't work well with remote developers

# Testing: Integrate Test Definition with Designing

We now revisit specification and design documents to insert testing plan.

In practice, we would do them concurrently.

Levels

- Requirements Specification and Validation Plan
- High-Level Design and Test Plan
- Detailed Design and Test Plan

# Software Testing Challenge

Complete testing is impossible

- There are too many possible inputs
  - Valid inputs
  - Invalid inputs
  - Different timing on inputs
- There are too many possible control flow paths
  - Conditionals, loops, switches, interrupts…
  - And you would need to retest after every bug fix
  - Combinatorial explosion
- Some design errors can't be found through testing
  - Specifications may be wrong
- You can't prove programs correct using logic
  - Even if the program completely matches the specification, the spec may still be wrong
- User interface (and design) issues are too complex

# So Why Test Software?

- Testing IS NOT "the process of verifying the program works correctly"
  - You can't verify the program works correctly
  - The program probably won't work correctly in all possible cases
    - Professional programmers have 1-3 bugs per 100 lines of code after it is "done"
  - Testers shouldn't try to prove the program works
    - If you want and expect your program to work, you'll unconsciously miss failure because human beings are inherently biased
- The purpose of testing is to find problems quickly
  - Does the software violate the specifications?
  - Does the software violate unstated requirements?
- The purpose of finding problems is to fix them
  - Then fix the most important problems, as there isn't enough time to fix all of them
  - The *Pareto Principle* defines "the vital few, the trivial many"
    - Bugs are uneven in frequency – a vital few contribute the majority of the program failures. Fix these first.

# Approaches to Testing

## Incremental Testing

- Code a function and then test it (*module/unit/element testing*)
- Then test a few working functions together (*integration testing)*
  - Continue enlarging the scope of tests as you write new functions
- Incremental testing requires extra code for the *test harness*
  - A *driver* function calls the function to be tested
  - A *stub* function might be needed to simulate a function called by the function under test, and which returns or modifies data.
  - The test harness can *automate* the testing of individual functions to detect later bugs

## Big Bang Testing

- Code up all of the functions to create the system
- Test the complete system
  - Plug and pray

# Why Test Incrementally?

Finding out what failed is much easier

- With Big Bang, since no function has been thoroughly tested, most probably have bugs
- Now the question is “Which bug in which module causes the failure I see?”
- Errors in one module can make it difficult to test another module
  - Errors in fundamental modules (e.g. kernel) can appear as bugs in other many other dependent modules

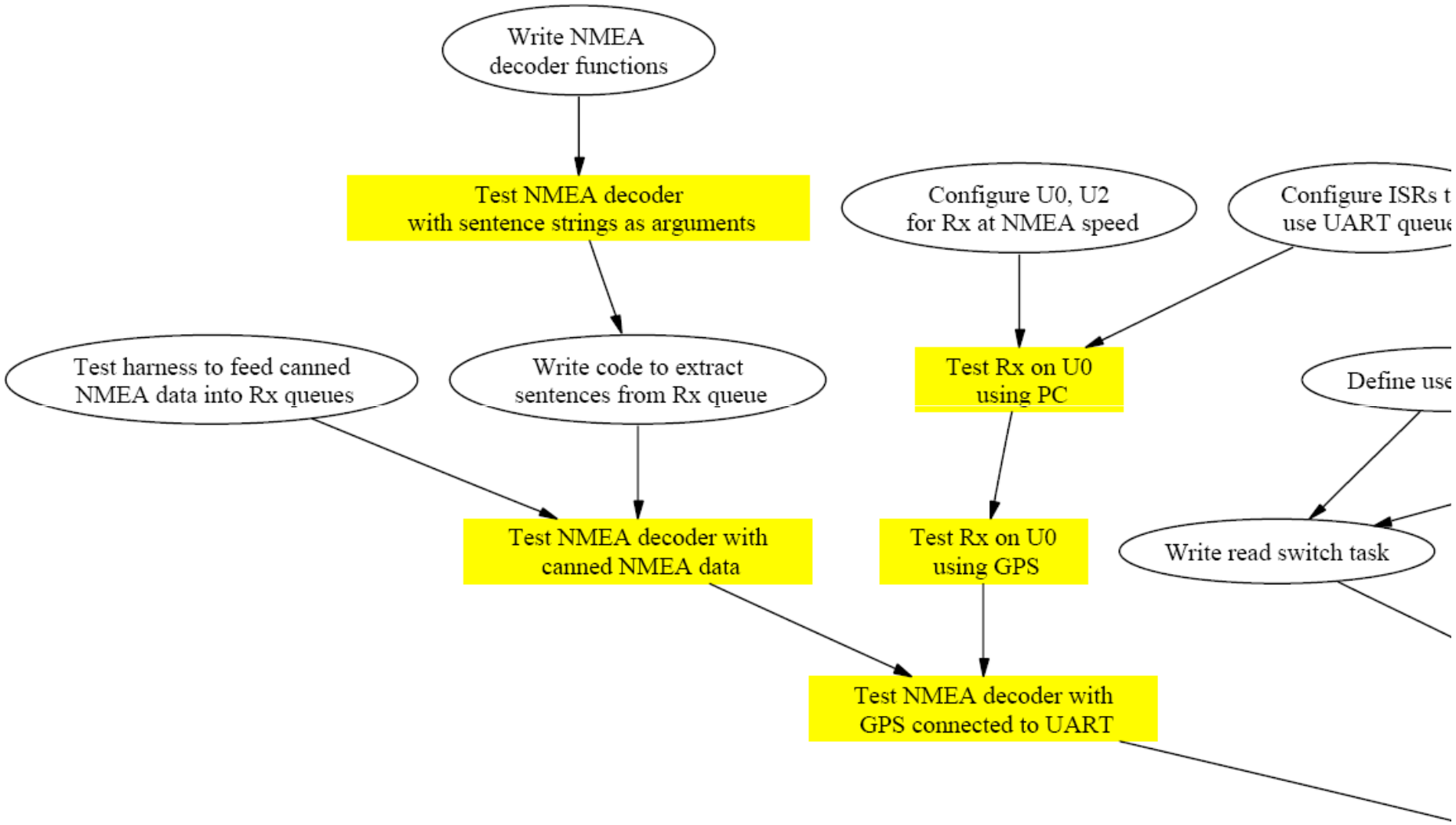
Less finger pointing = happier SW development team

- It’s clear who made the mistake, and it’s clear who needs to fix it

Better automation

- Drivers and stubs initially require time to develop, but save time for future testing

# Example


Write NMEA decoder functions
Test NMEA decoder with sentence strings as arguments
Configure U0, U2 for Rx at NMEA speed
Configure ISRs t use UART queue
Test harness to feed canned NMEA data into Rx queues
Write code to extract sentences from Rx queue
Test Rx on U0 using PC
Define use
Test NMEA decoder with canned NMEA data
Test Rx on U0 using GPS
Write read switch task
Test NMEA decoder with GPS connected to UART

# Clear Box (White Box) Testing

## How?

- Exercise code based on *knowledge of how program is written*
- Some tools enable automatic branch/code coverage analysis
  - If requirements state that all code be executed, this coverage information is essential

## Subcategories

- Condition Testing
  - Test a variation of each condition in a function
    - True/False condition requires two tests
    - Comparison condition requires three tests
      - » A>B? A < B, A == B, A > B
  - Compound conditions
    - E.g. (n>3) && (n != 343)
- Loop Testing
  - Ensure code works regardless of number of loop iterations
  - Design test cases so loop executes 0, 1 or maximum number of times
  - Loop nesting or dependence requires more cases

# Black Box Testing

Complement to white box testing

Goal is to find

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavioral or performance errors
- Initialization and termination errors

Want each test to

- Reduce the number of additional tests needed for reasonable testing
- Tell us about presence or absence of a class of errors

# In-Class Exercise: Test Plan Definition

Create test plans for Chart Maker

- Requirements + acceptance tests
- High-level design + integration test plans
- Detailed design + unit tests

# Perform Project Postmortems

Goals – improve your engineering processes

- extract all useful information learned from the just-completed project – provide “virtual experience” to others
- provide positive non-confrontational feedback
- document problems and solutions clearly and concisely for future use

Basic rule: problems need solutions

- A postmortem is not…
  - a forum for assigning blame to engineers
  - a way to complain about management
  - a place to whine

Often small changes improve performance, but are easy to forget

# Example Postmortem Structure

- Product
  - Bugs
  - Software design
  - Hardware design
- Process
  - Code standards
  - Code interfacing
  - Change control
  - How we did it
  - Team coordination
- Support
  - Tools
  - Team burnout
  - Change orders
  - Personnel availability