

# 3

# Requirements & Methodical Engineering

**Distributed Embedded Systems**

**Philip Koopman**

**September 4, 2015**

**Carnegie  
Mellon**

# Course Update

---

- ◆ **Enrollment: will update verbally in class**
- ◆ **Please be sure to initial or add your name to an attendance list today**
  - We will contact missing students to see if they are going to drop
  - We will recommend dept. admit wait listed students who are here today according to dept. priority order – SPACE AVAILABLE
- ◆ **Fridays TA meetings & recitations:**
  - RECITATION for all students is 1:30-2:00 PM.
  - RESPOND to e-mail request for time conflicts or we will assume you are free
  - TA meetings will be announced on blackboard
  - Some groups might be at 12 or 2:30 depending upon class size
    - We will work with conflicts outside class time.
- ◆ **Labs due on Thursday nights**
  - First lab due on Thursday
  - Primary goal is to exercise things; much simpler than later labs
  - Time and submission mechanism on course web pages – go read them!
  - We'll post on blackboard & e-mail when hand-in folders are available

# **Important – Check Blackboard Announcements**

---

- ◆ **Blackboard announcements are the official way to publish course information**
  - Including required assignments (e.g., “send e-mail with following info”)
  - Including class cancellations, changes, group assignments, etc.
  - Including late-breaking bug fix announcements, etc.
- ◆ **Check Blackboard announcements daily**
  - Read them completely and please follow instructions precisely
    - If we say do something a certain way, we say it for an excellent reason
  - Web site will be updated as appropriate, but no reasonable way to “push” changes out to you that doesn’t cause more problems than it solves
  - E-mail alerts may be sent as backup in selected instances, but have proven too unreliable to be the primary communication mechanism
- ◆ **Expect announcements on:**
  - Sending us e-mail with group selections
  - Recitation & group meeting plans (ideally all happens 12:30-2:20 Fridays)

# Course Notes

---

## ◆ Group selections happening by ~Wednesday

## ◆ Office hours

- Prof. Koopman office hours are:
    - In my office (HH A-308)
    - Immediately after class every Monday & Wednesday starting today.
  
    - Please come up to my office for extended questions
    - We need to vacate this classroom by 2:25 PM
  
  - TAs office hours announced on blackboard/course web site
    - In general, expect to have at least 1 hour every week-day
- 
- ## ◆ Readings require a CMU IP address to access
- **Use CMU VPN** (course web page has a pointer) or other VPN service
    - <http://vpn.cmu.edu>

# Where Are We Now?

---

## ◆ Where we've been:

- General embedded/distributed background info
- Elevator domain information

## ◆ Where we're going today:

- Overview of methodical engineering material in book chs. 2, 3, 5, 6, 8
- A specific approach to embedded behavioral requirements
  - The template used for project software requirement specifications

## ◆ Where we're going next:

- Example end-to-end UML based design process

# Preview

---

## ◆ A brief look at software process models

- Creating software isn't just hacking out some code
- Knowing how to solder doesn't make someone an electronics engineer
- Knowing how to code doesn't make someone a software engineer
- Simply knowing how to solder + code doesn't make you an embedded system engineer – there is a lot more to it!

## ◆ Creating requirements

- A template that works for distributed embedded systems
- Discussion of more general requirements issues
- Specific coverage of requirements for the course project

## ◆ A brief mention of IEEE standards

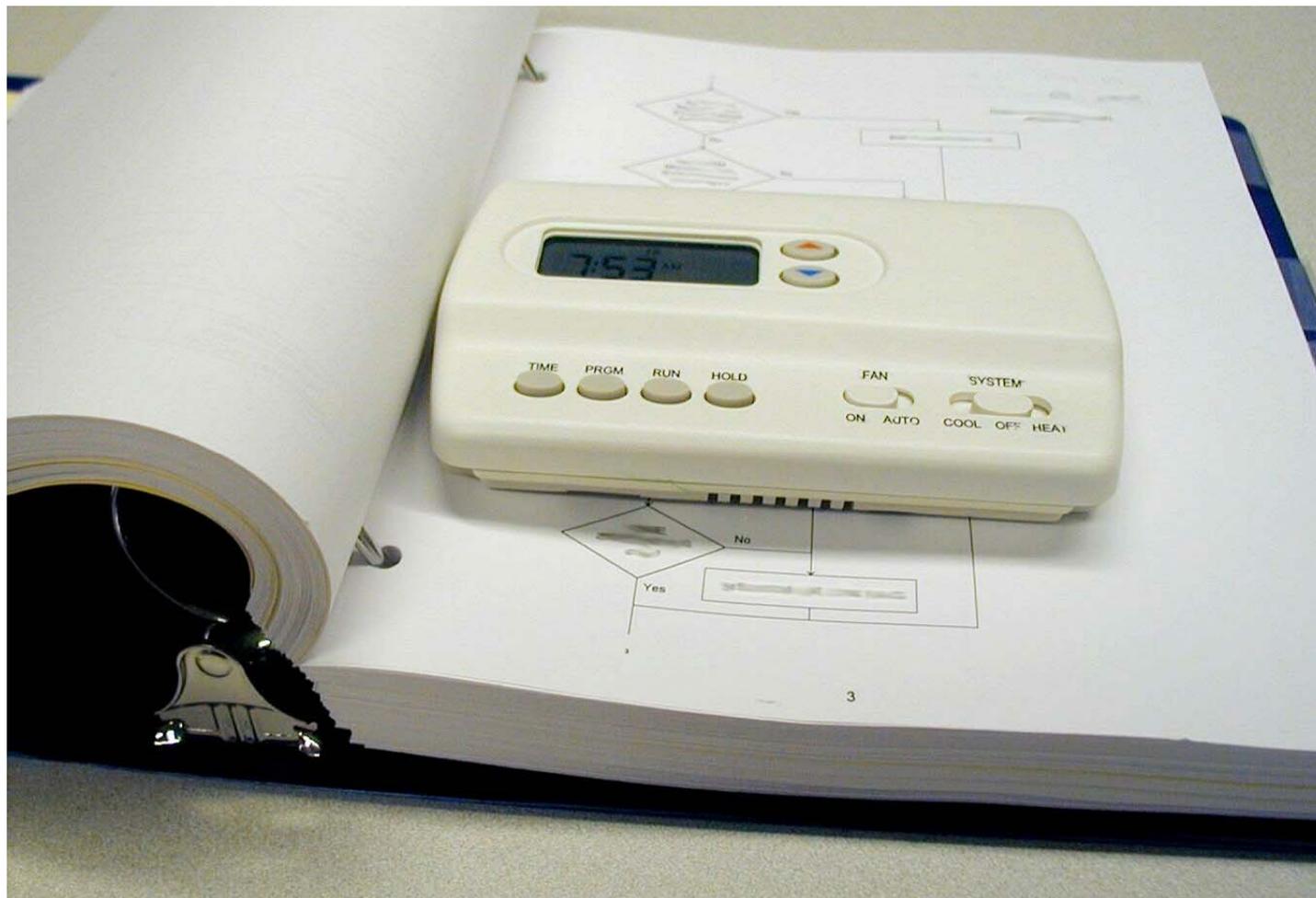
## ◆ Note: this lecture is simply one way to do requirements

- Requirements elicitation and analysis is a big topic
- This course teaches survival skills

# ***Myth: Embedded Systems Are Trivial***

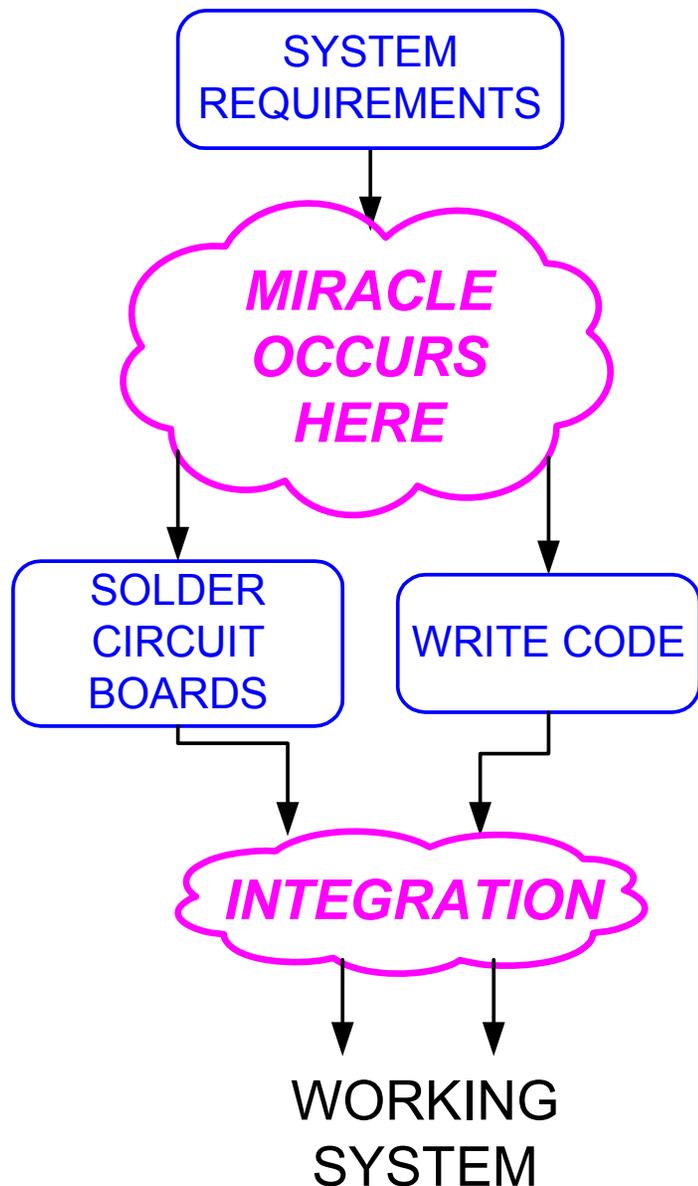
---

- ◆ ***Reality: Winning the game requires shoving 20 pounds into an 3 ounce sack***
  - Here's the design package for a household setback thermostat



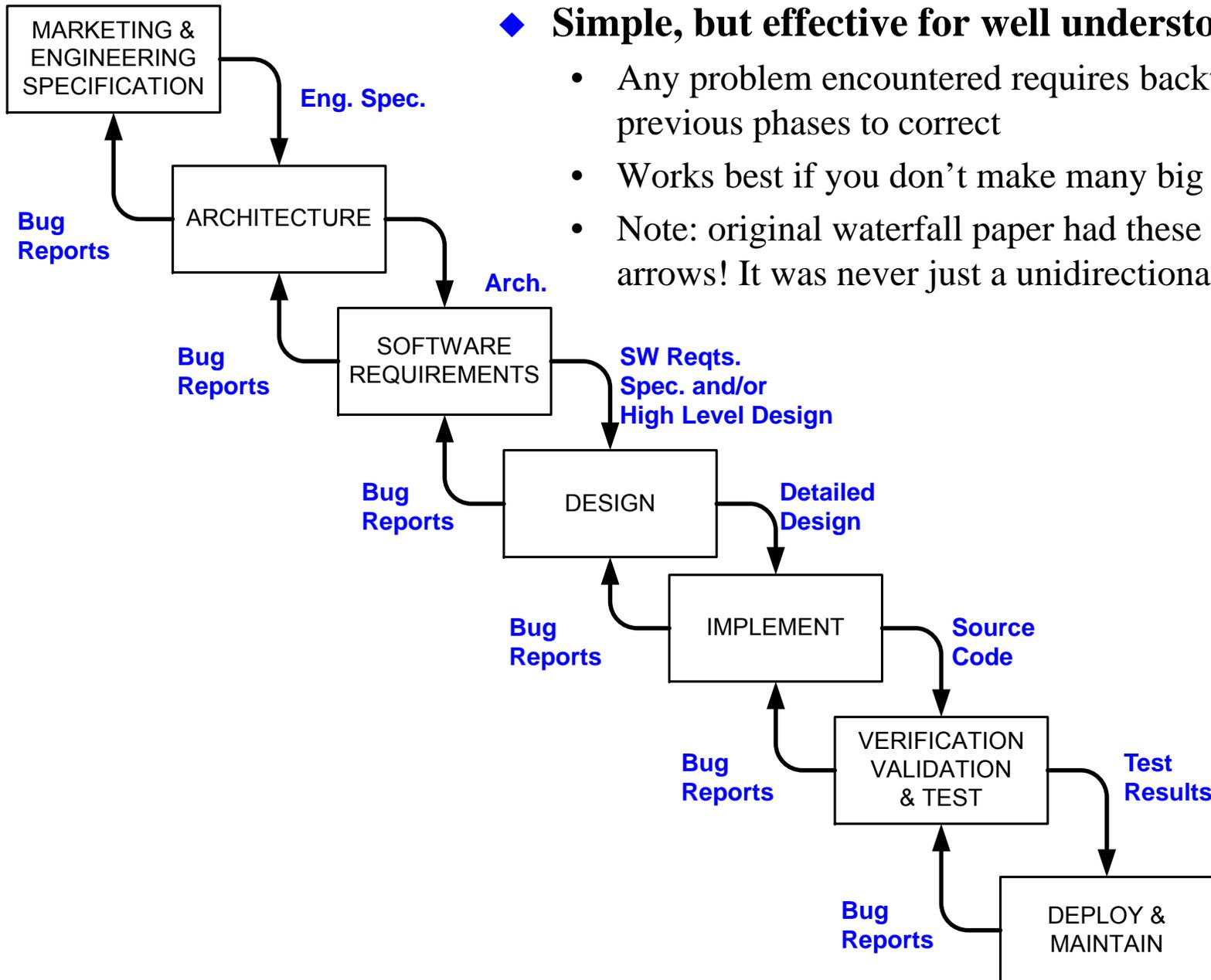
# An All-Too-Common Project Methodology

---



- ◆ **System requirements in form of project assignment; assumed perfect**
  - In the real world you get to write requirements
  - Requirements are *never* complete; *never* perfect at start of project
- ◆ **Many steps between requirements and implementation**
  - Would you start soldering/wire wrapping without a schematic?
  - Would you write code or VHDL without a design?
- ◆ **Most projects spend 50% of their schedule in integration**
  - Mostly because they are paying for shortcuts taken earlier

# Old-School Waterfall Development Cycle

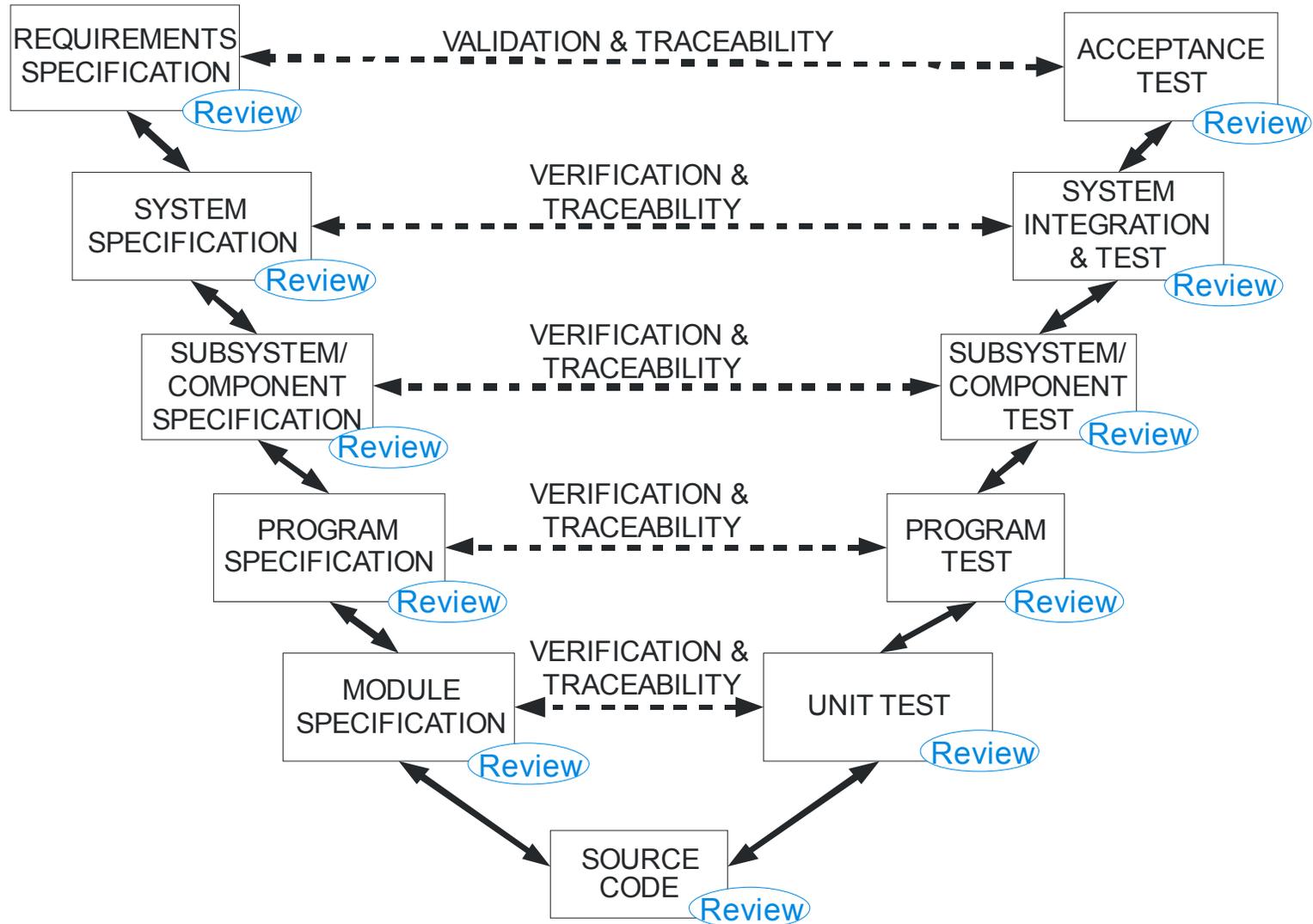


## ◆ Simple, but effective for well understood situations

- Any problem encountered requires backtracking to previous phases to correct
- Works best if you don't make many big mistakes
- Note: original waterfall paper had these backward arrows! It was never just a unidirectional process

# V (or “Vee”) Development Cycle

- ◆ Waterfall model modified to exploit subsystem modularity
  - Popular for automotive system design



# Software Development Plans

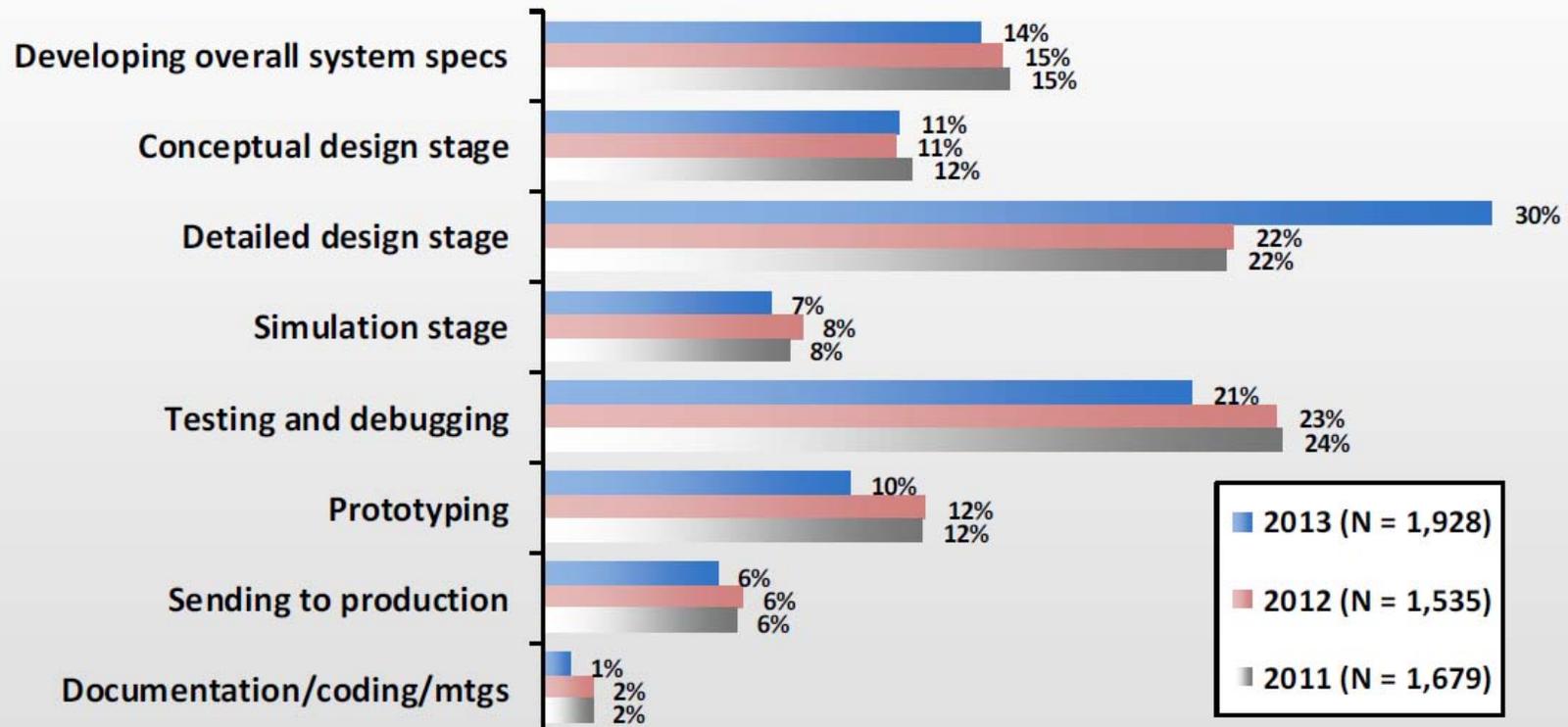
---

- ◆ **A written plan that spells out how development project runs:**
  - Development approach
  - How requirements are created and managed
  - How architecture is defined and evolved
  - How testing is planned and executed
  - How design & implementation are performed and documented
  - How reviews are conducted
  - Other project aspects: maintenance, project management, staffing, ...
  
- ◆ **Each step has defined activities, inputs outputs**
  - Boxes-and-arrows diagram with each box and arrow labeled
  - Arrow labels must correspond to an “artifact” (e.g., document) – or there is no way to tell if you actually are ready to follow the arrow to the next box

# Writing Software Is Zero Percent Writing Code

## 2013 Embedded Market Study

What percentage of your design time is spent on each of the following stages?



# Software Engineering Perspective

---

## ◆ Why software engineering?

- Creating software is becoming a substantial cost
- Special methods & approaches are necessary to manage software complexity
  - To reduce costs
  - To reduce schedule variability
  - To improve quality so that software defects don't eventually ruin company
- Embedded applications are held to higher standards than desktop software
  - People don't expect to have to reboot their car at every stoplight!

## ◆ Some high level lessons to remember from an embedded perspective

- Good process is vital for getting good software (and you need good people too!)
  - Design reviews are the highest impact activity you can start doing right away
- You need to hire (or train) software professionals
  - Just knowing how to solder does not make someone an electronics engineer
  - Just knowing how to write code does not make someone a software engineer
  - Knowing both how to solder and write code does not make someone an embedded software engineer

# Good Systems Start With Good Requirements

---

- ◆ **Make sure system does the right things**
- ◆ **Make sure system doesn't do the wrong things**
- ◆ **List all constraints the product must meet**

- ◆ To the optimist, the glass is half full.
- ◆ To the pessimist, the glass is half empty.
- ◆ To the engineer,  
the glass is twice as large as it needs to be.

- ◆ **Make sure the system is complex enough,  
but no more complex than required**
  - Minimize cost/minimize complexity
  - By extension, simpler requirements are better

# Tiers of Requirements

---

## ◆ Marketing Requirements

- Functional: Elevator shall deliver all passengers with reasonable speed
  - Perhaps: Elevator shall be faster than competitor's elevator
- Nonfunctional: Elevator shall increase maintenance contract profits
- Constraint: "Product shall be compatible with building existing maintenance networks"

## ◆ Product Requirements

- Functional: elevator shall have peak speed of 11.3 meters/sec (if fastest competitor is 11.2 meters/sec)
- Nonfunctional: elevator shall have a patented diagnostic interface
- Constraint: elevator shall be compatible with BACnet, but need not use it for internal functions

## ◆ Software Requirements

- How do we build the software to get it done?
- For example, multi-tasking system with per-task functionality
- Functional: main motor shall use PID control with 10 msec loop rate
- Many requirements deal with the details

# Marketing Requirements

---

## ◆ Marketing Requirement:

- Business needs / “Voice of Customer”
- It is a requirement that describes a specific feature (as opposed to a goal)
- 1-2 sentences, perhaps with supporting rationale:
  - Customer shall receive prompt feedback that elevator call has been registered  
Rationale: without feedback, customers beat on buttons, causing early wearout

## ◆ It should be (but almost never is):

- Measurable.
  - Acceptance test should be able to measure it
- Conflict-free
- Concrete, precise, unambiguous
  - Use “shall”, “should”, etc.
- Ranked according to desirability and cost/performance acceptability threshold

## ◆ It is often likely to be:

- Emergent – not something that is easy to specifically design
  - E.g., “has to fit in palm/in a shirt pocket” is easy to measure; difficult to build
- A “feature” on a feature checklist, given from a customer point of view

# Product Requirements

---

- ◆ **This is where engineering makes acceptance criteria less squishy**
- ◆ **Functional Requirements**
  - “Depressing button X shall illuminate light Y within 200 msec.”  
Rationale: human time constants for button presses are 200-500 msec
- ◆ **Non-functional Requirements & Emergent Properties**
  - “Mean time between unsafe operating situations for each rail signal shall be greater than 250,000 years” (example from a subway system)
  - “Mean time to repair a single component failure shall be less than 30 minutes after arrival of mechanic bringing standard tool set.” (typical jet aircraft)
  - “Vehicle shall exceed 25 mpg”
  - “Elevator shall deliver at least 1000 passenger\*floors/minute at up-peak”
- ◆ **Constraints**
  - Specifies a required technical or other approach
    - “.NET technology shall be used”
  - Specifies regulatory or other constraints on solution space/design process
    - “System shall conform to requirements of DO-178b” (FAA software process)
- ◆ **Assumptions/operating conditions**
  - “Assume no power outage lasts more than 30 minutes”

# A “Cookbook” Embedded Requirements Process

---

- ◆ **This is a common method – not the “best”, nor even “right” way**
  - It’s easy to take an entire course to teach heavy-duty UML methodology
  - The purpose of this lecture is to give you a basic toolkit that’s better than just hacking out the code
  - (Current “best” ways usually break when used on a *real* embedded system!)
  
- ◆ **Requirements Creation**
  1. Elicitation: Identify business/system requirements
  2. Create architecture / allocate functions to subsystems
  3. Create scenarios/use cases
  4. Create detailed behavioral requirements
  
- ◆ **Requirements Traceability & Risk Management**
  1. Create traceability matrices to trace:
    - System req.  behavioral req.  implementations  integration tests
    - System req.  acceptance tests
  2. Simulate system to check global/emergent behaviors
  3. Prototype system to check allocation-based properties

# A Word On Traceability

## ◆ Traceability is checking to ensure that steps of the process fit together

- Market Reqts ⇔ Customer focus groups, etc.
- Product Reqts ⇔ System Acceptance Test
- Software Reqts ⇔ Software Integration Test
- Module Reqts ⇔ Unit test

## ◆ Forward Traceability:

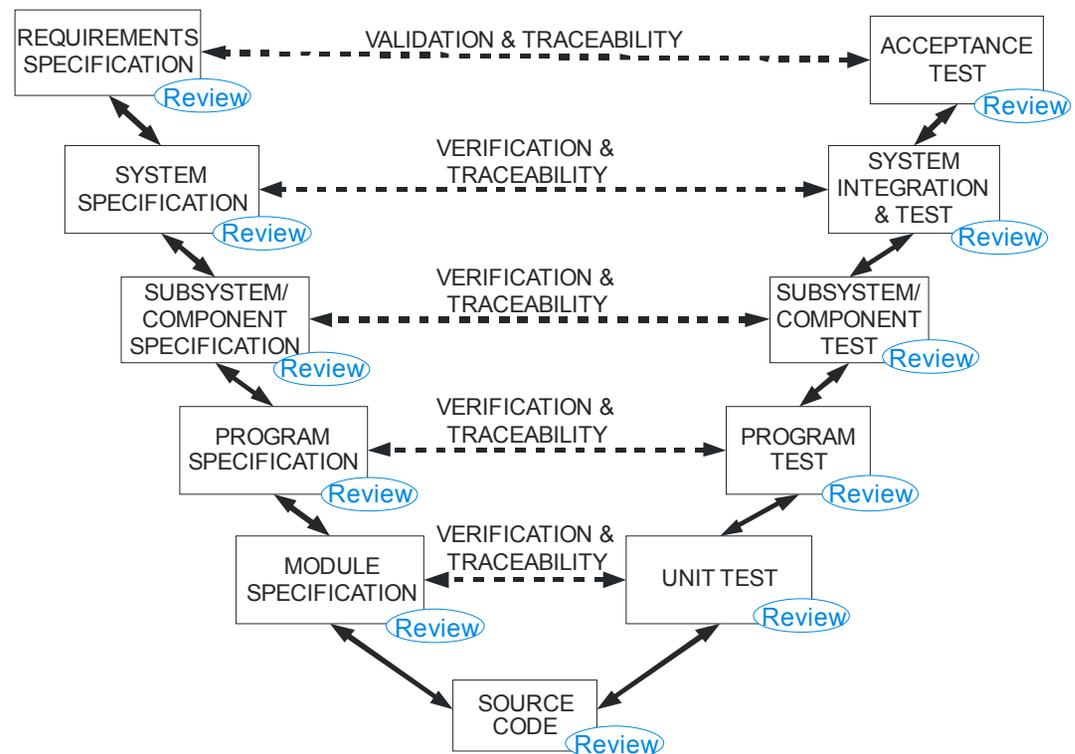
- Next step in process has everything in current step
- “Nothing got left out”

## ◆ Backward Traceability

- Previous step in process provoked everything in current step
- “Nothing spurious included /no gold plating”

## ◆ More on this in later talks

- Lots of examples in end-to-end UML lecture



# 1) Requirements Elicitation

---

- ◆ **Customer may provide requirements in request for quote (RFQ)**
- ◆ **Vendor may need to interview customer and extract requirements**
  - Requirements phase may precede design phase under a separate contract
- ◆ **You might have engineering judgment ( “guessing” )**
  - “I’ll know it when I see it”
  - “Same as last time except better”
- ◆ **This is one place where being a good writer + clear thinker really helps!**
  - There are various cookbook methods to do requirements elicitation, but that’s beyond the scope of this course
  - It can help a lot to have a professional technical writer on the requirements team

# Important Requirement Key Words

---

(Shall/Should are universal; the rest are merely suggestions to show how tricky this stuff can get)

## ◆ Behaviors/Constraints:

- **Shall** = system has to do it 100% of the time unless specifically excepted
- **Should** = desirable that system does it whenever reasonable to do so
- **Can** = system can do something, but no particular incentive to implement

## ◆ User Actions:

- **Must** = user has to do this (same as “shall”, except for user, not computer)
- **May** = user can exhibit this behavior, but does not have to

## ◆ Environment words:

- **Will** = designer can count on environment being this way
- **Might** = designer has to accommodate situation, but can't count on it

## ◆ Change risk:

- **Expected to change** = this area is likely to be changed
- **Could change** = this area is something that could change, but might not

# Elevator Example: High Level Requirements

---

1. (*What does it do?*): **All passengers shall be delivered to desired destination floor in a timely manner.**
  
  2. (*Safety*): **Any unsafe condition shall cause an emergency stop**
    - An emergency stop should never occur
  
  3. (*What metrics matter?*): **Performance should be optimized to extent possible, including customer-specified weightings for:**
    - Delivery times:
      - Maximum end-to-end passenger delivery time
      - Maximum passenger waiting time
      - Average end-to-end passenger delivery time
      - Average passenger waiting time
    - ...
- ◆ *Note:*        **SHALL = mandatory;    SHOULD = desirable**

## 2) Create Architecture

---

- ◆ **For now think of this as a class diagram**

- However, architecture is both objects AND interfaces, so there is more to it
- We'll explore that a bit in later lectures

- ◆ **For now, assume you have a list of objects in the system**

- Elevator car
- Doors
- Hall call buttons
- Car call buttons
- Directional lanterns

# 3) Create Scenarios/Use Cases

---

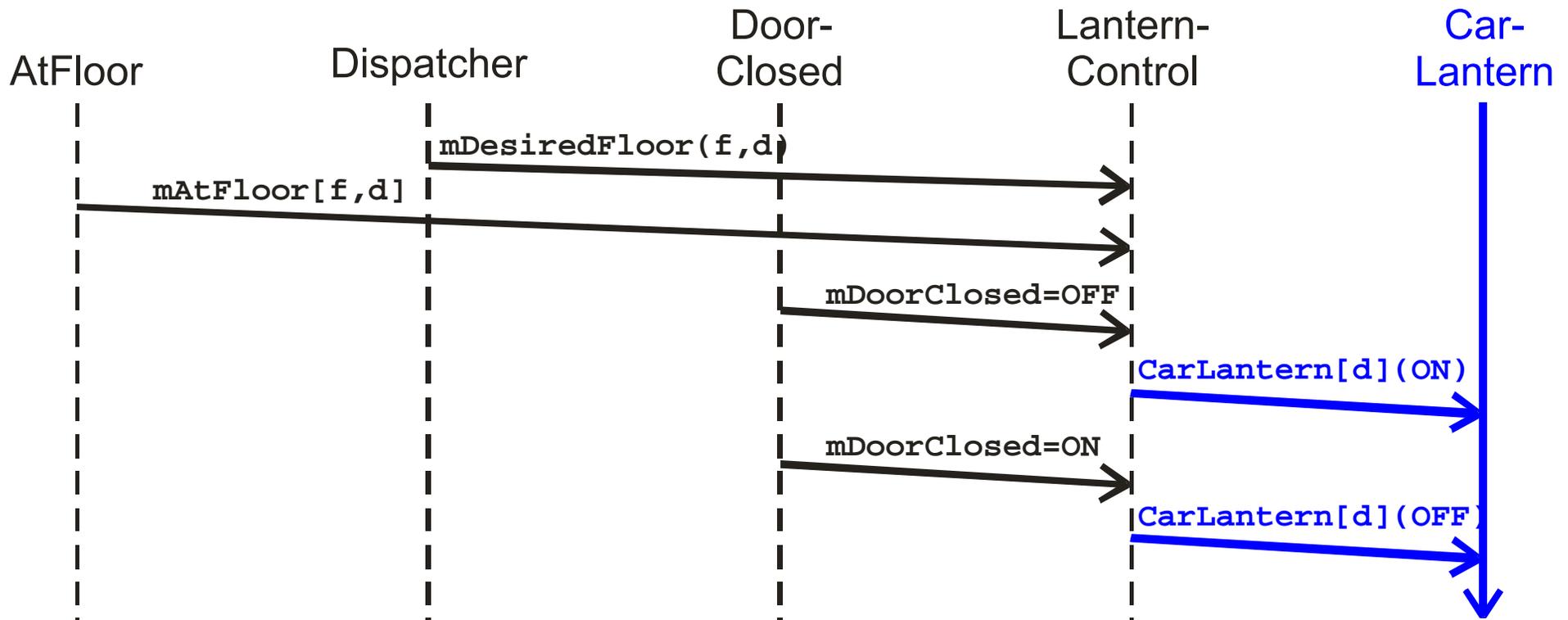
## ◆ First: high level flows through the system

- Idea is to create a flow chart of actions experienced by actors / “object lifecycles”
  - Elevator passenger from initial entry into system until final delivery
  - Driver from entering car to leaving car
- Each block of flow chart yields one or more use cases
  - (Object-oriented/other approaches possible, but flow charts are usual practice)

## ◆ Second: “building-block” use cases catching snippets of functionality

- Multiple scenarios possible for each use case (elevator example):
  - Passenger calls car
    - » Scenario: Presses button once
    - » Scenario: Button has already been pressed by someone else
  - Passenger enters car
    - » Scenario: Successful first try
    - » Scenario: Gets bumped by door and exits instead of entering, then retries
  - Passenger calls destination
    - » ...

# Example Elevator Lantern Sequence Diagram



◆ **Note: CarLantern[d] messages are physical actuations (power applied to wires), not literally messages on the network.**

- But, we consider them “messages” for design purposes
- The project simulation framework passes them as a special class of message that takes zero time and consumes no network bandwidth
- More on this in recitation and later lectures

## 4) More Detailed Software Requirements

---

Below are generic embedded examples; we'll have a more precise style than this:

### **Behavioral Requirement:**

2.3.1 When the first probe temperature exceeds 80 degrees C +/- 1 degree C, the heating element shall turn off within 100 msec.

Rationale: this is a software safety mechanism to avoid actuating safety relief valve

....

### **Constraint:**

5.7.1 Program memory shall be no more than 60% full at initial product release.

Rationale: this leaves room for software expansion and avoids software costs caused by nearly-full memory.

# Our Detailed Behavioral/Module Requirements

---

## 1. List subsystems

- In a fine-grain distributed system, this is sensors+actuators+objects
  - “Other objects” are usually compute-nodes such as dispatcher
- Actors included to provide environmental model and interface

## 2. Template for behaviors requirements for each subsystem:

- Replication
- Instantiation
  - Initial conditions
  - When are dynamic objects are created?
- Assumptions about how other modules behave
- I/O interface (list of sensor/actuator interfaces)
- State (private variables useful for describing behavioral memory)
- Constraints
  - Non-functional requirements; safety interlocks
- Behaviors
  - Functional requirements

# Example Elevator Behavioral Requirements – 1

---

## *LanternControl[d]*

### ◆ Replication:

- *(How many are there and where are they?)*
- Two controllers, one for each lantern {Up, Down} mounted in the Car. Each controller controls two lightbulbs in parallel (one by each of the Car's front and back doors), and actuates each front/back pair of bulbs as a single actuator.

### ◆ Instantiation:

- *(What are settings at initialization; when are they created (default is permanent))*
- Lanterns are Off at initialization.

### ◆ Assumptions:

- *(What do you need to assume to meet constraints given listed behaviors?)*
- CarLanterns[d] are never commanded to be On at the same time.

### ◆ Input Interface:

- *(What inputs are available?)*
- mDoorClosed[b, r] -- “m” prefix means network message; “r” is right/left
- mDesiredFloor
- mAtFloor[f, b] -- “f” means one set per floor; “b” is front/back

# Example Elevator Behavioral Requirements – 2

---

## *LanternControl[d] (continued)*

### ◆ **Output Interface:**

- *(What outputs are available?)*
- CarLantern[d] (physical interface to light bulbs!)
- mCarLantern[d] (network message to tell other modules what's happening)

### ◆ **State:**

- *(What private state variables are maintained? What notational macros are used?)*
- **DesiredDirection** = {Up, Down, Stop} computed desired direction based on comparing CurrentFloor with Floor desired by Dispatcher. This is implicitly computed and used as a macro in the behavior descriptions.

### ◆ **Constraints:**

- *(What invariants must hold? – “passive” requirements.)*
- 7.1 Both CarLanterns[d] shall not be On at the same time.

# Example Elevator Behavioral Requirements – 3

---

## *LanternControl[d] (continued)*

### ◆ BEHAVIORS:

- *(What active behaviors must be implemented?)*

7.2 When all mDoorClosed[b,r] are False, CarLantern[DesiredDirection] shall turn On.

7.2.1 If DesiredDirection is Stop, neither lantern shall illuminate.

7.3 When all mDoorClosed[b,r] are True, all CarLantern[d] shall be turned Off.

### ◆ Notes:

- These may not be exactly the right behaviors you want, but they are a start
  - For example, they may seem intuitive but “when” is ambiguous
- These requirements are really half-way to implementation (but that’s useful)
- You need detailed object interfaces & message dictionary to do this

# Magic Formula for Behavioral Requirements

---

## ◆ Event-driven system (think “interrupts”):

- *(#ID) <message received>* shall result in *<messages transmitted>* and/or *<variable values assigned>*
  - Account for all possible messages received
  - Account for all possible messages that need to be transmitted
  - Make sure all variables are set as required
  - OK to transmit multiple messages; OK to set multiple variables
- OK to also use:  
*<message\_received>* and *variable == X* on left hand side of “shall” statement
  - OK to use multiple variables on left hand side
- ***ONLY ONE*** received message per requirement (network serializes messages; simultaneous reception of multiple messages is *impossible*)

## ◆ Time-triggered system is different (think “polled I/O”):

- Keep copies of last value received for each message and periodically set outgoing message values for next periodic transmission
- Permits using multiple received messages
  - We’ll see this later in more detail.

## ◆ ***EVERY VERB GETS A NUMBER***

- Numbers are used to tracing requirements to implementation & tests later on

# Additional Characteristics of a Good Requirement

---

## ◆ Understandable and unambiguous

- Understandable to system designers who have to implement it
- Understandable to marketing/sales/customers who know the needs
- Understandable to outside vendors (non-domain experts) who have to build it
- Concise (few words) and simple
- Consistent – same word used to mean same thing in all places

## ◆ Implementation-neutral

- Doesn't say how to do it, just what the desired outcome/action/side-effect is

## ◆ *Testable!!!*

- If you can't test it, how do you know the system meets the requirement?
  - Constraints can be difficult – how do you prove something is impossible?
- In general, this implies measurable
  - Includes tolerance for measurement – most measurements are not exact

## ◆ *Traceable!*

- In practice, this means every requirement has a unique number or identifier
- Must be possible to trace that requirement is satisfied by final system
  - Sometimes done by tracing to features/system component functions
  - More often done in practice by tracing to system tests

# What About Robustness?

---

- ◆ **This is largely an unexplored area, but matters in real life**
  - More robust systems tend to make few assumptions about environment
  - Therefore they can withstand design errors/problems elsewhere
  
- ◆ **The previous example is not very robust**
  - Constraint: can't turn on both lanterns
  - Current solution: assume nobody tries to do that(!)
  
  - More robust solution:
    - Good: if one lantern is on, ignore commands to other lantern  
(but what if the defect forgets to turn the first lantern off? It stays stuck on)
    - Probably better: if one lantern is commanded on, turn the other lantern off
    - Probably even better: after a predetermined time of no messages, turn both lanterns off

# Distributed Embedded Requirements Issues

---

- ◆ **Cost/weight/power/fuel economy/emergent properties**
  - Usually determined by an allocation budget
  - Budgets vary in accuracy, but are seldom perfect
  - Sometimes a certifying authority/test is invoked (*e.g.*, “Meets spec. UL-1998”)
- ◆ **Real time deadlines**
  - Some derived from estimated physical time constants of machines & people
  - Sometimes computing deadlines are used to apportion hardware resources
- ◆ **Safety/Security**
  - Safety is usually based on interlocks and hardware reliability
    - Requirements specifically address unsafe operations
    - “Single-point failure shall not cause unsafe situation”
  - Usually it is assumed that software is perfect and therefore not a safety problem
    - This is a bad idea for safety, and a worse idea for security
  - Security, if addressed at all, usually depends on isolation/firewall

# A Fable: The Headlights and the Tunnel

---



[Fractured Fairy Tales  
(A Pittsburgh Original)]

Recently, when there was a highway department testing a new safety proposal. They asked motorists to turn on their headlights as they drove through a tunnel. However, shortly after exiting the tunnel the motorists encountered a scenic-view overlook. Many of them pulled off the road to look at the reflections of wildflowers in pristine mountain streams and snow-covered mountain peaks 50 miles away. When the motorists returned to their cars, they found that their car batteries were dead, because they had left their headlights on.

So, the highway department decided to erect signs to get the drivers to turn off their head-lights.

[Based on Gause and Weinberg, 1990]

- 
- ◆ **First they tried:**



**TURN YOUR LIGHTS OFF**

- ◆ **But someone said that not everyone would heed the request to turn their headlights on, and they couldn't turn their headlights off.**

- 
- ◆ **So they tried:**



**IF YOUR HEADLIGHTS ARE  
ON TURN THEM OFF**

- ◆ **But someone objected that would be inappropriate if it were night time.**

- 
- ◆ So they tried:



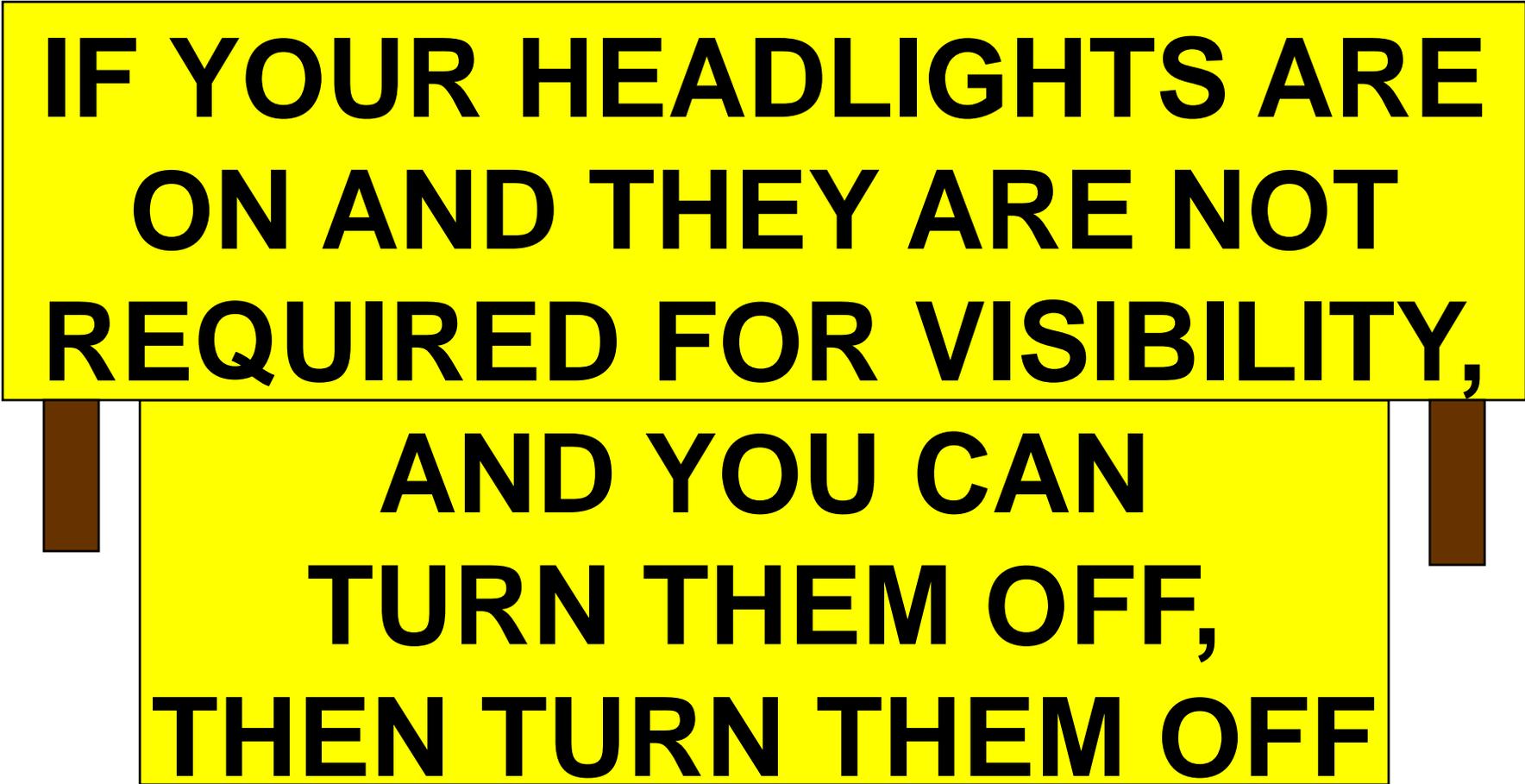
- ◆ But someone objected that would be inappropriate if it were overcast and visibility was greatly reduced.

- 
- ◆ So they tried:

**IF YOUR HEADLIGHTS ARE  
ON AND THEY ARE NOT  
REQUIRED FOR VISIBILITY,  
THEN TURN THEM OFF**

- ◆ But someone objected that many new cars are built so that their headlights are on whenever the motor is running, so they couldn't be turned off.

- 
- ◆ So they tried:



**IF YOUR HEADLIGHTS ARE  
ON AND THEY ARE NOT  
REQUIRED FOR VISIBILITY,  
AND YOU CAN  
TURN THEM OFF,  
THEN TURN THEM OFF**

- ◆ But someone objected....

- 
- ◆ **They decided to stop trying to identify applicable states – just alert the drivers and let them take appropriate action**



**ARE YOUR LIGHTS ON?**

- 
- ◆ **But, shorter is even better:**



# Morals Of That Fable For Requirements

---

- ◆ **“Simple” situations can lead to complex requirements**
  - The final sign might work, but it is not a precise specification of behavior
    - In fact, it is merely identifying a situation that might trigger behaviors (exiting a tunnel)
  - “Common sense” is not so common after all, and often not simple
- ◆ **You have to sufficiently understand the domain to understand the requirements**
  - “Turn headlights on when appropriate” might be a good starting point to tell an engineer... → This is a marketing requirement
  - ... but not a good ending point for detailed behavioral specifications.
    - This is an engineering requirement
  - ... and probably isn't useful when trying to communicate with a synthesis tool!
    - This is a high level design
- ◆ **Perennial tradeoffs:**
  - Complexity vs. coverage
  - Specification brevity vs. domain expertise required for understanding

# IEEE Standard 830-1998

---

- ◆ **“IEEE Recommended Practice for Software Requirements Specifications”**
  - “SRS” = Software requirements specification
  - In the embedded world, it should be “System Requirements Specification”!
- ◆ **Areas addressed:**
  - Functionality (what does it do)
  - External interfaces (this is really architecture, but is important to have in SRS)
  - Performance (speed, real-time issues)
  - Attributes (non-functional requirements such as maintainability, reliability)
  - Design constraints (applicable standards, policies, *etc.*)
- ◆ **Much of it is really a different presentation of material in this lecture**

# IEEE 830-1998 Outline for SRS

---

## Table of Contents

### 1. Introduction

#### 1.1 Purpose

#### 1.2 Scope

#### 1.3 Definitions, acronyms, and abbreviations

#### 1.4 References

#### 1.5 Overview

### 2. Overall description

#### 2.1 Product perspective

#### 2.2 Product functions

#### 2.3 User characteristics

#### 2.4 Constraints

#### 2.5 Assumptions and dependencies

### 3. Specific requirements (See 5.3.1 through 5.3.8 for explanations of possible specific requirements. See also Annex A for several different ways of organizing this section of the SRS.)

## Appendixes

## Index

**Figure 1 – Prototype SRS outline**

# Other IEEE Standards

---

## ◆ They aren't the only way to do things

- But they are a way that can be standardized upon without a lot of writing work
- They represent a minimum best practice
- CMU has a license for full access to IEEE Standards (IEEE Xplore) on line

## ◆ Other relevant IEEE standards for embedded systems (partial listing):

- IEEE Std. 1223-1998. *IEEE Guide for Developing System Requirements Specifications.*
  - Much higher level look at the system level
- IEEE Std. 730-1998 *IEEE Standard for Software Quality Assurance Plans.*
  - Points to many of the software process specifications
- IEEE Std 1483-2000, *IEEE standard for verification of vital functions in processor-based systems used in rail transit control.*
  - The rail people have a lot of good detailed experience at critical software
- There are some nuclear specifications too, but they tend to rely on hardware as a safety net.

# Requirement Pitfalls & Good Ideas

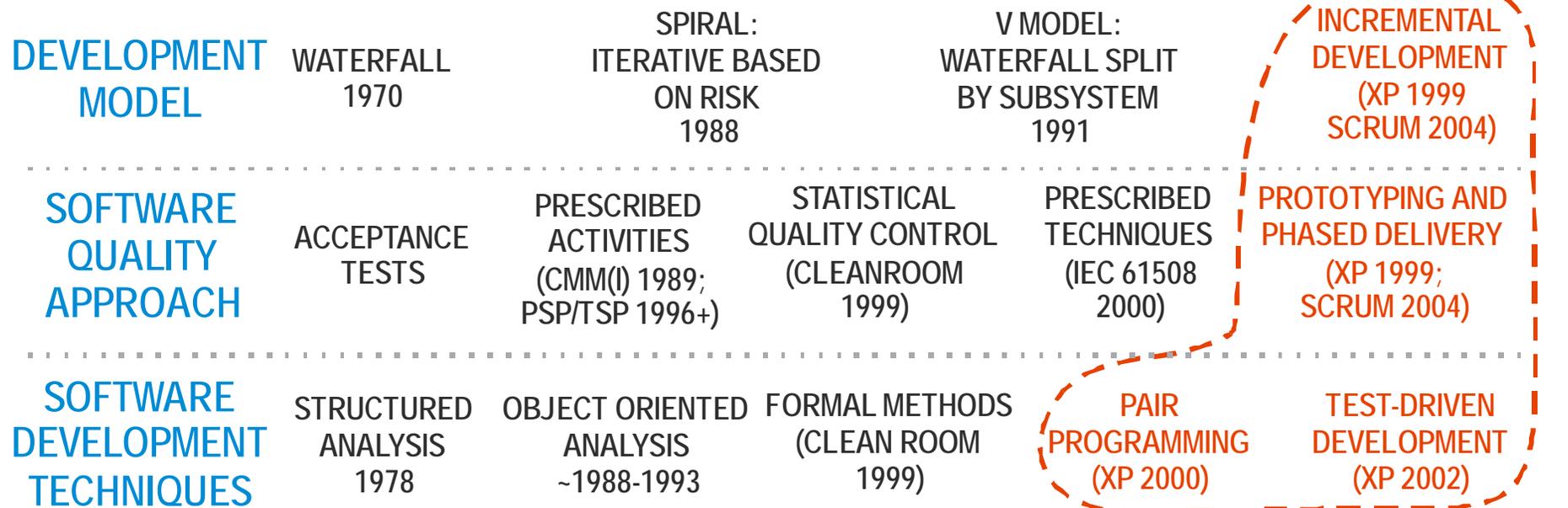
---

- ◆ **Don't require perfection**
  - “This product shall have no software defects” (yeah, right)
- ◆ **Don't make it hard to decide if something meets requirements**
  - Product shall be fast and user friendly
- ◆ **Measure something you can measure**
  - If you can't measure reliability, measure minimum up-time in stress test
- ◆ **Get the testers to help you**
  - They know what they can test – don't tell them to do something impossible!
- ◆ **Collect data during test and deployment**
  - If you have trouble measuring requirements, see what happens in the field
- ◆ **If you can't measure the product, measure the process**
  - If you need near-perfect software, you'll never measure it directly ....  
... but you *can* measure whether the development process was followed well.

# Agile Methods As A Recent SW Development Trend

## ◆ Combination of several ideas

- Some are old ideas: e.g., self-organizing teams; some are new ideas
- Development model + quality approach + techniques
- Emphasizes adaptation to change
- Every real project evolves, whether it is Agile or Waterfall



# Agile Methods + Embedded (?)

---

- ◆ **No reason Agile Methods can't work with embedded**
  - Primary benefit (in my opinion) is that it makes developers happier
    - Also can, if run properly, let development start while requirements evolve
    - But, but you need to manage and moderate the risks
- ◆ **Issue: “Agile” is sometimes is camouflage for cowboy coding**
  - Undefined, undisciplined processes are bad, regardless of what you call them
  - Yes, Agile teams actually should follow a rigorously defined process
- ◆ **Issue: “No-paper” Agile has problems with long-lived systems**
  - 10+ year old undocumented systems are a nightmare; Agile doesn't change that
- ◆ **Issue: Agile doesn't have an independent system acceptance test**
  - Implicitly assumes that defects are tolerable OR fixes are easy to deploy (neither of which is true of many embedded systems)
- ◆ **Issue: Agile doesn't have independent process monitoring (SQA)**
  - Software Quality Assurance (SQA) tells you if your process is working
  - Agile teams may be dysfunctional and have no idea this is happening
    - Or they may be fine – but who knows if they are really healthy or not?

# How Much “Paper” Is Enough?

---

- ◆ **Old military development saying:**
  - You can’t deploy until the weight of the paper exceeds the weight of the system.
- ◆ **Does all this mean you need to be buried in paper?**
  - You can be clever about *minimizing* the bulk of paper
  - But if you have zero paper, there is no way to know if your process is actually working
  - For example if you have no architecture diagram, then you have no way to know if various developers all have the *same* architecture in their heads
- ◆ **Our course project uses a medium-light weight of paper**
  - Common in industry to have heavier weight paper
  - Our purpose is to give you enough so you can have a better idea of cost/benefit ratio of medium-weight paper processes
  - Safety critical paper is a whole different world; we’re not doing that in our project



# Review

---

- ◆ **READ the advice from previous students**
- ◆ **Requirements through behavioral specification process**
  - High level requirements – what the product is supposed to do
  - (architecture step is in a future lecture)
  - Behavioral specifications
  - Specific engineering process to use
    - More on design methodology in coming weeks; this is just a jump-start
    - We’re teaching survival skill versions – not the ultimate answer.
- ◆ **IEEE standards are heavily used in some companies**
  - Why recreate something when you can use it as a starting point?
    - And if you’re sued, at least you followed “best practice”
  - Other industries have other standards, but the idea is the same
- ◆ **Test hint – look for red ink in these slides (on-line copy is in color)**
- ◆ **Next lectures:**
  - **Complete description and walk-through of course design process**