12.5 Years Teaching Distributed Embedded System Design

Philip Koopman

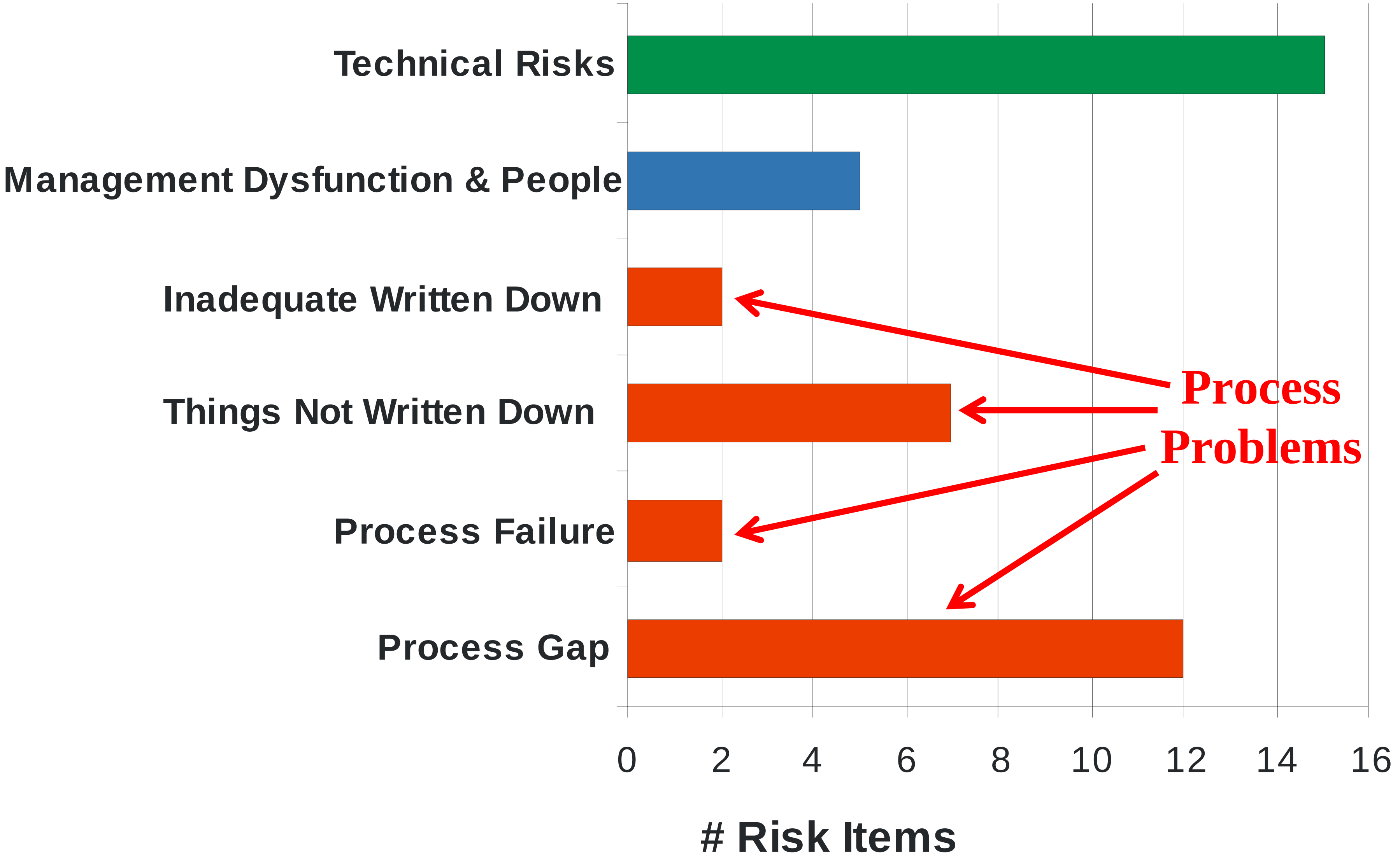
October 24, 2012

http://www.ece.cmu.edu/~ece649

Electrical & Computer ENGINEERING

Carnegie Mellon

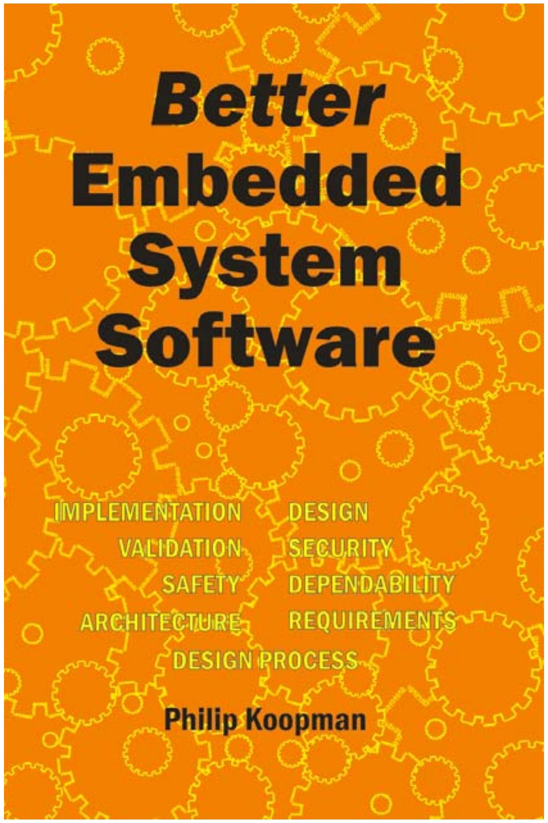
Experienced Teams: ~90 Industry Design Reviews



Process Problems

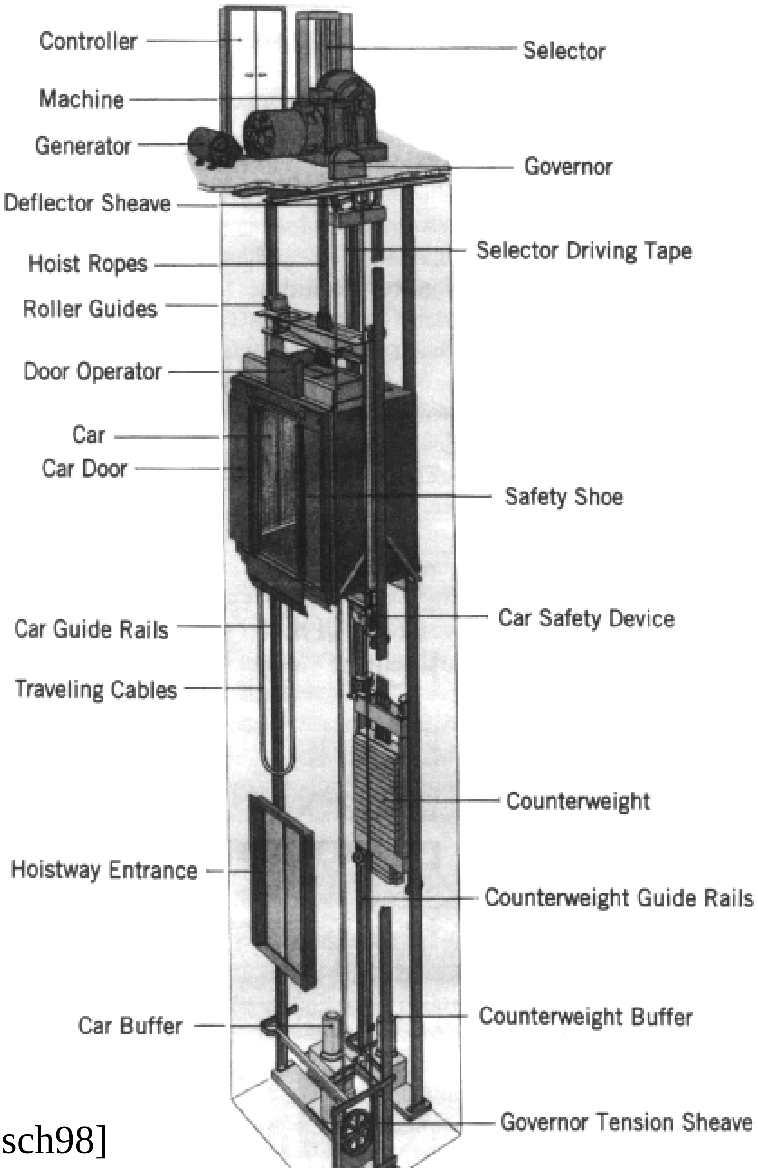
Only about 1/3 of risk areas are technical

The Course: 18-649 Distributed Embedded Systems

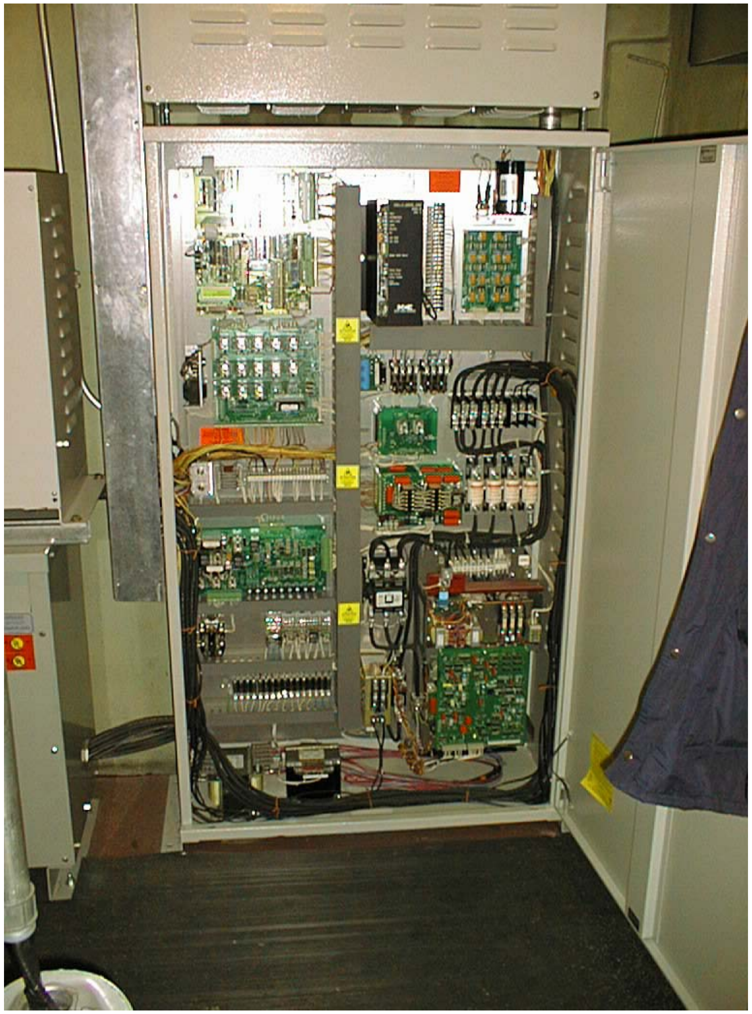


- **The “build an elevator” course**
  - 1/3 Software engineering skills
  - 1/3 Distributed embedded systems (e.g., CAN)
  - 1/3 Safety + Reliability + Validation
  - Semester-long software project
- **Informed by:**
  - Book based on industry design reviews
  - Lots of trial and error
- **What it IS and IS NOT**
  - NOT tool-centric; uses some UML
  - NO hardware; teaches discrete event simulation / Java
  - IS highly distributed; simulated CAN; prohibits use of a “brain node”
  - HAS rigorous traceability from requirements to design to tests
  - NOT “heavy” process, but strives for lightest weight that teaches concepts
  - NOT a capstone design project – process more than whizzy product

The Project: Highly Distributed Elevator

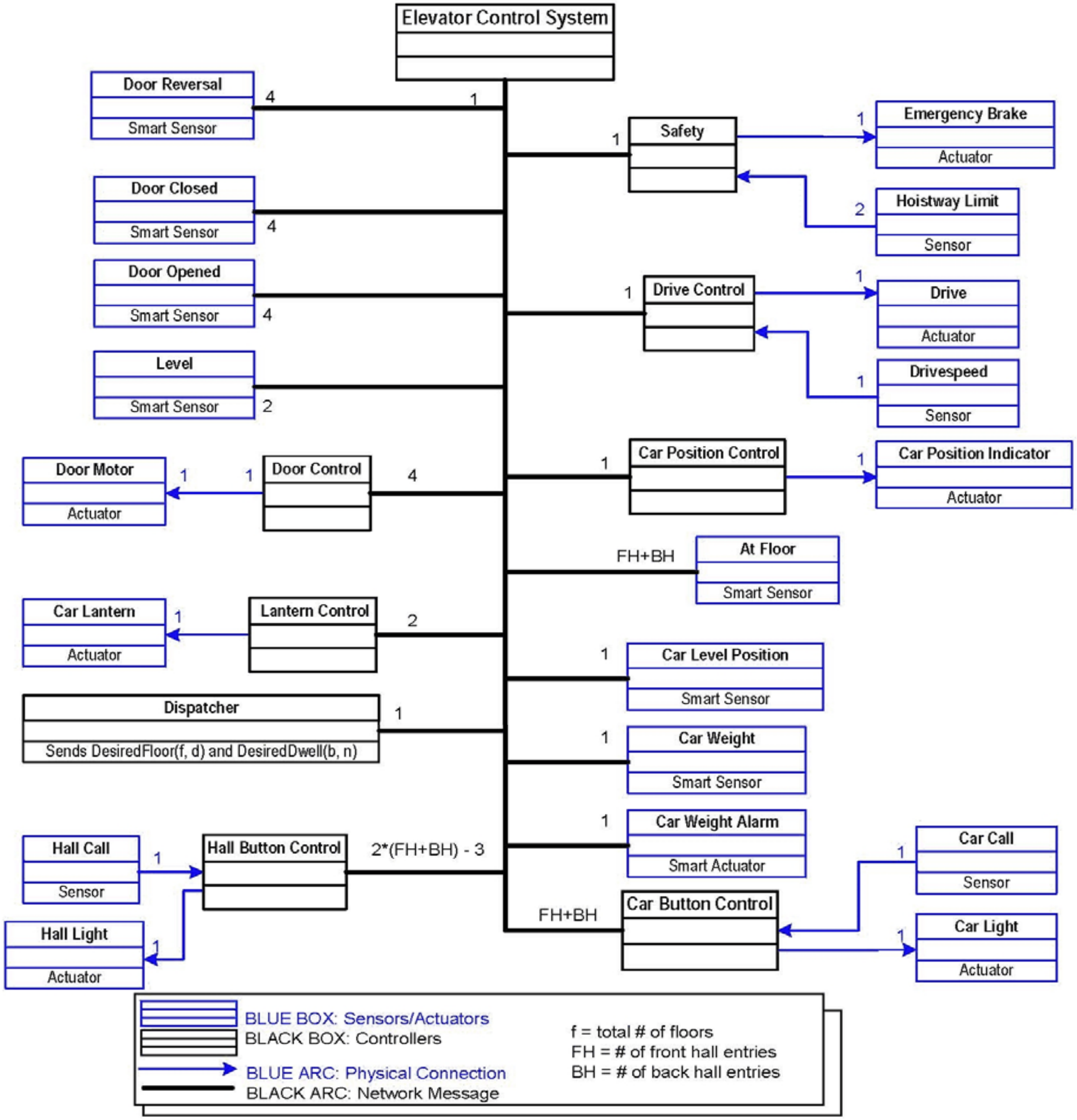


[Strakosch98]



Elevator Architecture Diagram

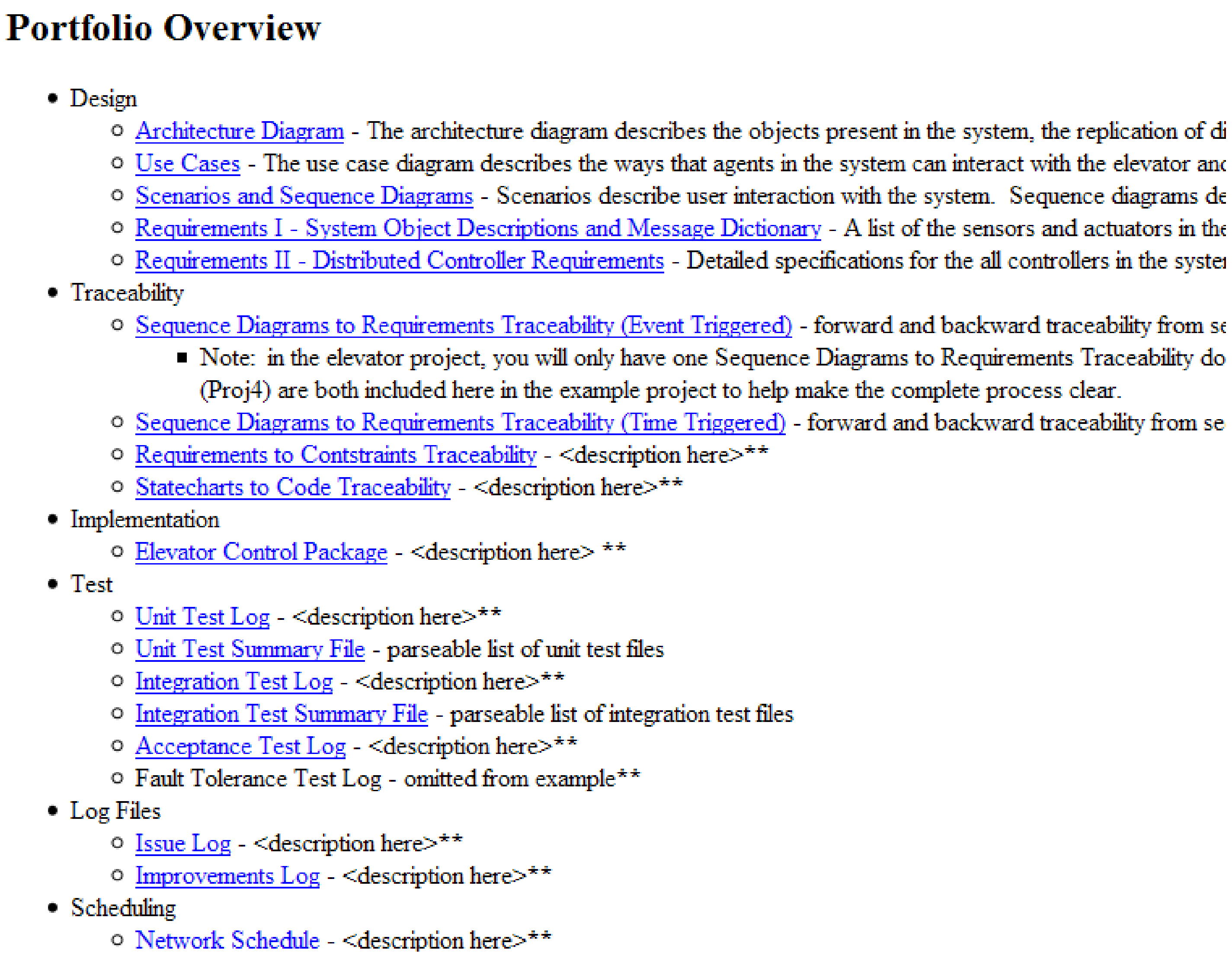
(revised 1/24/2011)



+ Multiple Simulated Passengers

Project Hand-In Via Portfolio

- ◆ **Printable HTML to keep things sane**
  - We provide the format; they fill it in; end-to-end updates every week



Portfolio Overview

- Design
  - Architecture Diagram - The architecture diagram describes the objects present in the system, the replication of di
  - Use Cases - The use case diagram describes the ways that agents in the system can interact with the elevator anc
  - Scenarios and Sequence Diagrams - Scenarios describe user interaction with the system. Sequence diagrams de
  - Requirements I - System Object Descriptions and Message Dictionary - A list of the sensors and actuators in the
  - Requirements II - Distributed Controller Requirements - Detailed specifications for the all controllers in the syster
- Traceability
  - Sequence Diagrams to Requirements Traceability (Event Triggered) - forward and backward traceability from se
    - Note: in the elevator project, you will only have one Sequence Diagrams to Requirements Traceability do (Proj4) are both included here in the example project to help make the complete process clear.
  - Sequence Diagrams to Requirements Traceability (Time Triggered) - forward and backward traceability from se
  - Requirements to Contstraints Traceability - <description here>**
  - Statecharts to Code Traceability - <description here>**
- Implementation
  - Elevator Control Package - <description here> **
- Test
  - Unit Test Log - <description here>**
  - Unit Test Summary File - parseable list of unit test files
  - Integration Test Log - <description here>**
  - Integration Test Summary File - parseable list of integration test files
  - Acceptance Test Log - <description here>**
  - Fault Tolerance Test Log - omitted from example**
- Log Files
  - Issue Log - <description here>**
  - Improvements Log - <description here>**
- Scheduling
  - Network Schedule - <description here>**

Brief History of Project Evolution

- **Domain Expertise: I spent industry time working with Otis on elevators**
  - Including time on a next-generation architecture team
  - Including embedded network protocol tradeoff study
  - *Challenge:* creating a gritty, realistic project requires domain expertise

- **1999 – First Project**
  - Developed requirements, simulator, and simulated passengers
  - One cycle through a guided, somewhat ad hoc, waterfall process
  - Seven project phases @ 2 weeks each

- **Current Project**
  - More Elevator control functions
    - Main motor dynamics (build-in): requires commit point calculation
    - Doors on both side; door nudge behavior
    - Random cable slip: requires low-speed leveling
    - Time-accurate CAN network (deadline monotic scheduling); but no CPU real time
  - Three iteration design process ("dumb" to "smart") for requirements changes
  - Thirteen projects, mostly weekly; end-to-end traceability for each hand-in

Main Project Goals

- **Solo and group development (and time management)**
  - Many “simple” modules that must work in concert with others
  - Dispatcher (where does elevator stop next) can get complex
- **Technical aspects**
  - Basic UML literacy – all designs are state charts, not flow charts
  - Deadline monotonic scheduling for CAN bus
  - Inherent race condition in elevator (door re-open vs. main drive turns on)
- **Complex design process**
  - Flexibility within fixed constraints (high level reqs; fixed interface defns)
  - Concurrent design, implementation, unit test of different modules
  - Multi-iteration; requirements changes from “dumb” to “smart” elevator
  - Tons of info (examples; source code); learn how to sort through it
- **Realistic but lightweight process, including quality**
  - Including SQA, traceability – seeing how things fit together
  - Doing enough peer review to experience the benefits
  - Doing enough testing to understand how to create effective tests

The Design Process We Teach

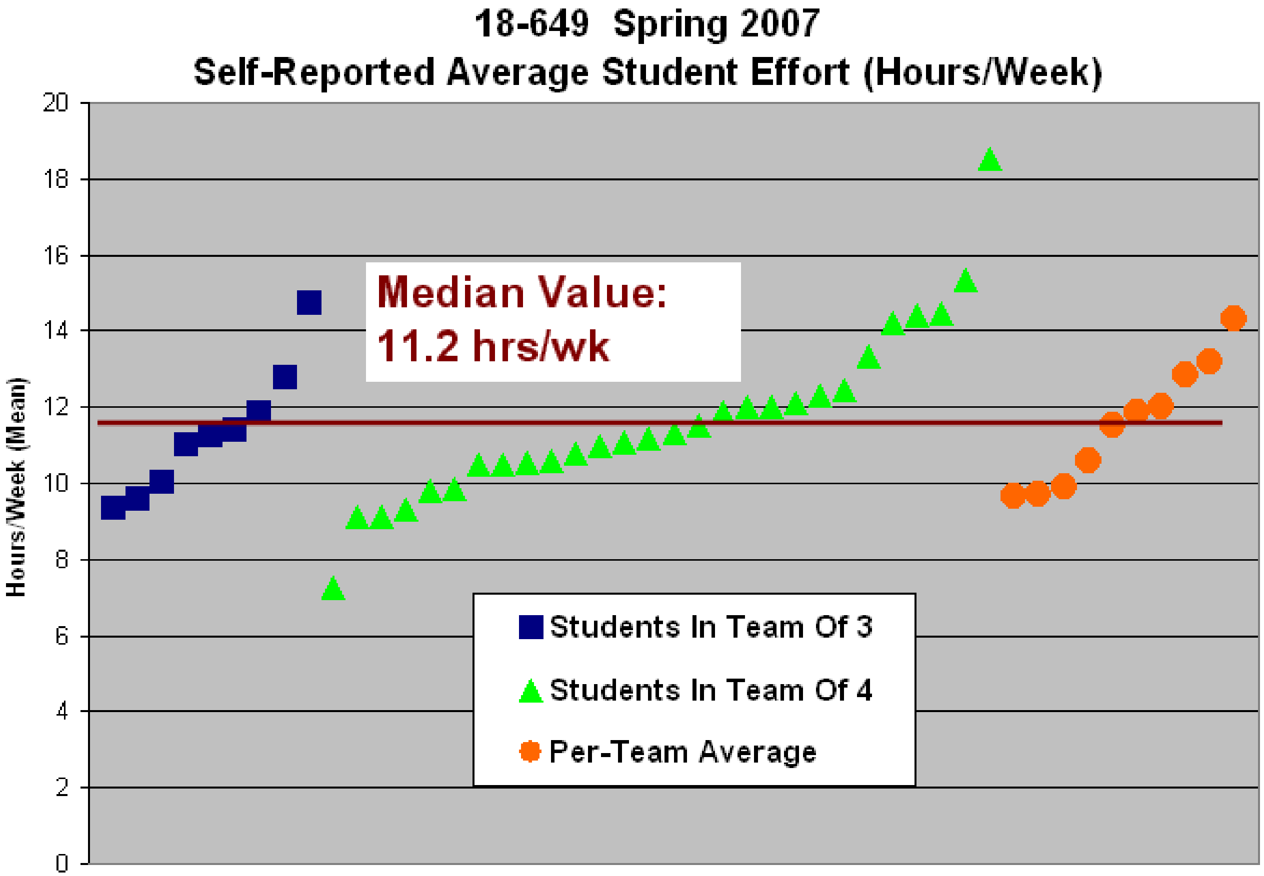
- **Need to address the entire process (not just the tool chain part)**
    - Teach some domain expertise
    - Requirements – provide high level requirements
    - Architecture – takes about 6 weeks to understand distributed approach
    - Use cases – provided as a kickstart
    - Sequence diagrams – provide some; they do the rest (Visio or Dia)
    - “Behaviors” – intermediate step to avoid exhaustive sequence diagram creation
    - Statecharts – by hand (Visio or Dia)
    - Code generation – by hand (Java, simulation framework provided)
    - Unit Test – test framework provided; traces to statecharts
    - Integration test – test framework provided; traces to sequence diagrams
    - System test – simulated passengers
    - Acceptance test – high level requirements run-time monitor; safety brake
    - Traceability required between big steps; must be updated

- **Challenge: tool support that doesn’t hide the fundamentals**
    - In our case, little automation but careful attention to make project “simple”
    - Downside – only works if everyone does the same high level project

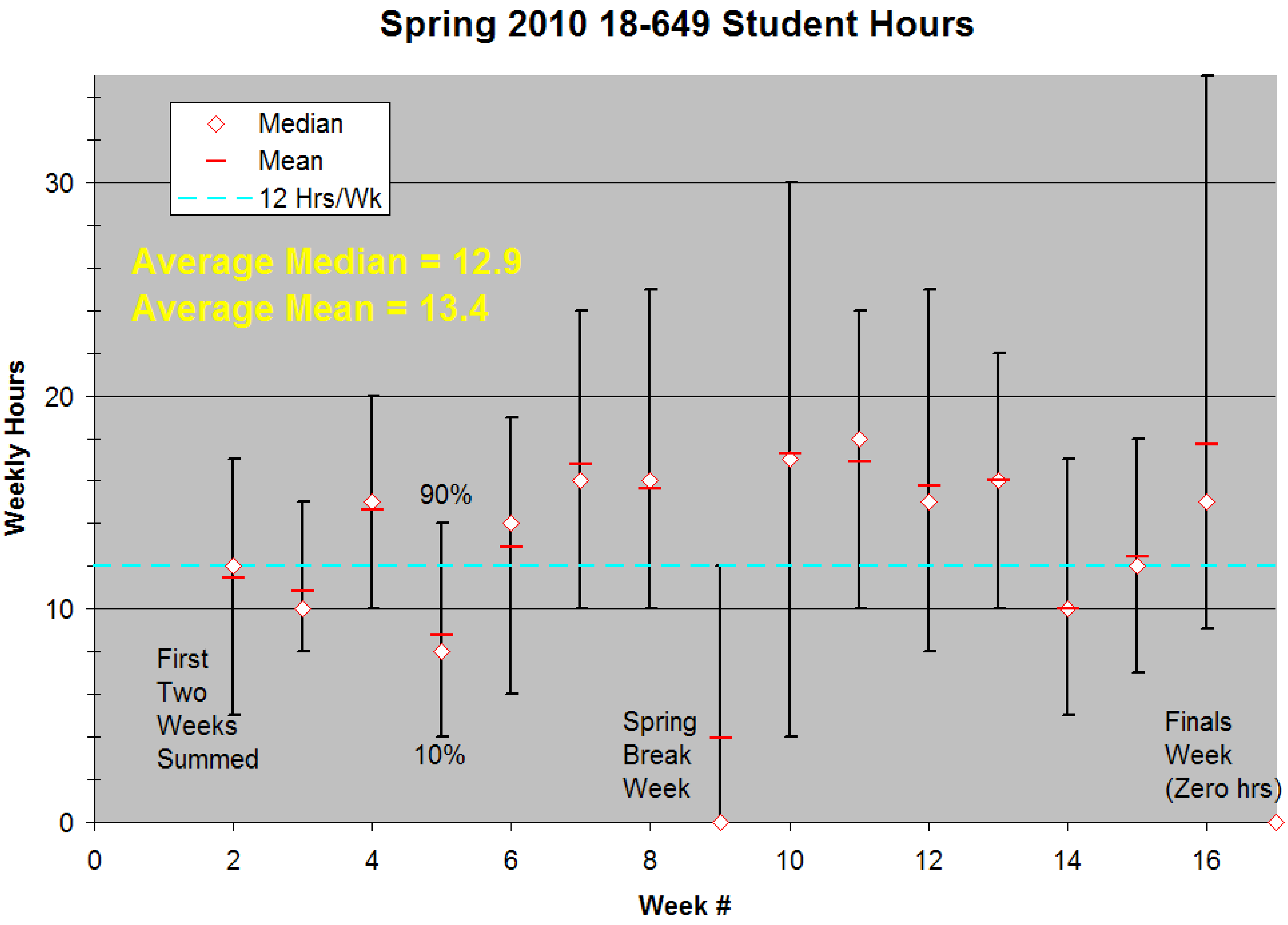
Project Teams

- **Essentially all teams are computer engineering students**
  - Some are mostly hardware; some are mostly software; a few with little background

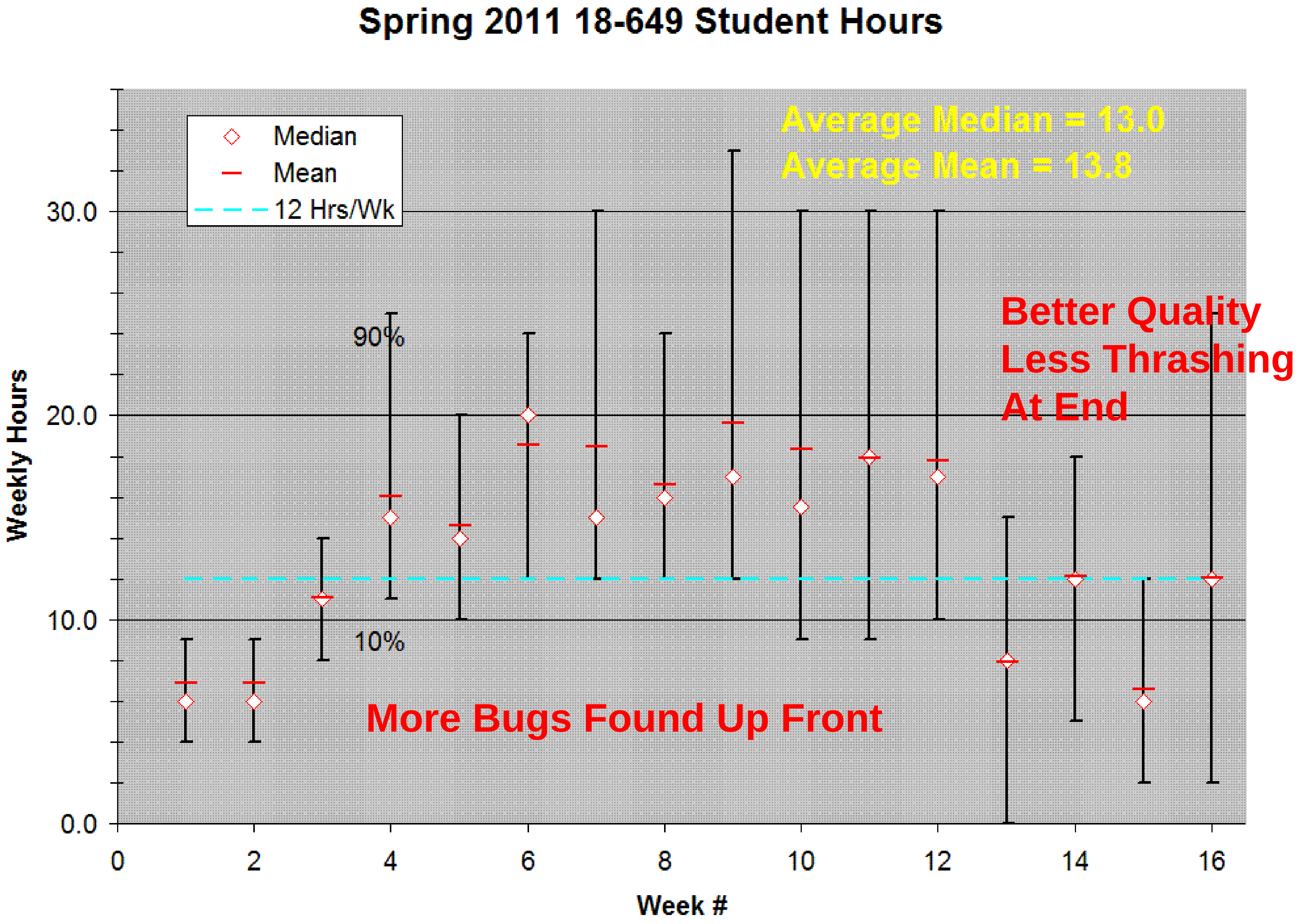
- **Teams of 3 or 4**
  - Team of 2 is too much work
  - I want to avoid teams of 5 – process isn't heavy enough
  - 3 vs. 4 makes no difference to weekly effort(!)



Before Peer Review Spreadsheet (Ineffective Reviews)



After Spreadsheet & Weekly Defect Reporting



CPS Challenges – Time Triggered Design

- **Students have trouble with time-triggered design**
  - Almost all think event-triggered at start of course
  - Jumping direct to time-triggered easily loses half of them
  - Approach:
    - Do simple design (not code) event-triggered at first….
… then re-do the same design as time-triggered
    - Have a really simple bright-line test for difference (event triggered takes one input message; time triggered can have multiple input messages)

- **Time triggered wrinkles**
  - Defining “any order is OK” for multiple arcs on sequence diagrams
  - Students take a while to get synchronous state charts vs. async.
  - Need to limit control loop speeds for realism (not doing CPU scheduling)
  - Students tend to play with timing to skirt (not cure) race conditions

Other CPS-Relevant Challenges

- **Control system dynamics**
  - Students need to understand dynamics for timing floor landings
    - Need to predict when to send “slow down” command to hit targer floor
  - But, discrete event simulator makes it painful for them to create controls

- **How do you know the system is working?**
  - “Seems to work” isn’t good enough for the real world – or realistic project
    - Students have usually been trained to hack away until it passes an easy test
    - It has to really work, all the time, for all test cases, for all timings
    - Need to instill and practice notions of “test coverage”
  - For example, “Elevator doors only open when call pending at that floor”
  - Our solutions:
    - Originally – use trace dump and grep/perl (ugly)
    - Currently – students build in system monitors for critical properties

- **Hard to do it all in one semester (12 unit course = 4 semester hours):**
  - Need significant lecture content, so project scope is limited
  - Light on: RTOS, security, control systems, hardware aspects

18-649 Lecture Topics

1. Overview
2. Elevator domain knowledge
3. Boeing 777 validation video
4. Requirements & methodical engineering
5. UML-based design
6. End-to-end project design example on soda vending machine
7. Distributed systems applied to embedded control
8. Reviews & software process
9. Testing
10. Communication protocols
11. CAN protocol case study
12. CAN performance

Test #1; mid-semester project demos

13. Economics (HW and SW)
14. Advanced elevator behavior
15. Verification/validation/certification
16. Distributed real time scheduling
17. Humans as a system component
18. Dependability
19. High integrity (critical) system design
20. Safety standards (e.g. IEC 61508)
21. Distributed time
22. Security & internet connectivity
23. FlexRay Protocol case study
24. Ethics & Societal impact

Test #2; final project demos

Topic Areas Covered In Text (omits networks)

- **Introduction**

  Software Development Process
  - Written development plan
  - How much paper is enough?
  - How much paper is too much?

- **Requirements & Architecture**
  - Written requirements
  - Measureable requirements
  - Tracing requirements to test
  - Non-functional requirements
  - Requirement churn
  - Software architecture
  - Modularity

- **Design**
  - Software design
  - Statecharts and modes
  - Real time
  - User interface design

- **Implementation**
  - How much assembly language is enough?
  - Coding style
  - The cost of nearly full resources
  - Global variables are evil
  - Mutexes and data access concurrency

- **Verification & Validation**
  - Static checking and compiler warnings
  - Peer reviews
  - Testing and test plans
  - Issue tracking & analysis
  - Run-time error logs

- **Critical System Properties**
  - Dependability
  - Security
  - Safety
  - Watchdog timers
  - System reset
  - Conclusions

Engineering Challenges Beyond Technical Stuff

- **Students often value technology more than engineering methods**
  - Students benefit from having followed a defined process
  - CAD-like tool chains implicitly tend to enforce a process…
    - … but students may not be able to extend that thinking beyond the tools
  - A good design project can teach most of them to understand value of process
    - Peer reviews that actually find defects
    - Design approaches that find bugs before the code is written
    - Tests that actually find problems early

- **Consider the right balance of what skills to teach**
  - We usually teach engineers to design cool new demos from scratch, but…

  - Many engineers spend their time modifying, not building from scratch
  - Many engineers spend their time testing, not designing from scratch
  - Many engineers have to make rock-solid systems, not flaky demos
  - Many engineers have to play together on a team with a defined process
  - Most engineers are de facto software engineers .. but are not trained that way