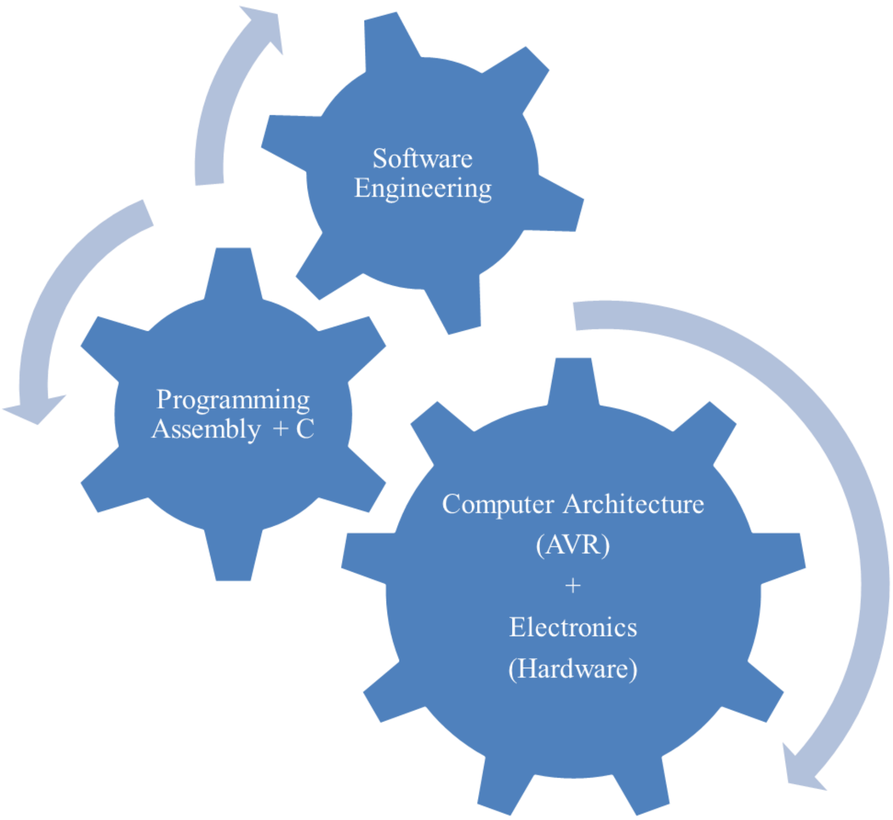
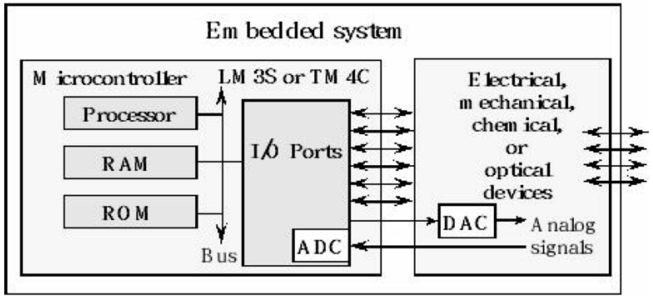
NCS 362: Embedded Systems



- Software Engineering.
- C Programming

Introduction to Embedded Systems

- Cyber-Physical Systems is another name for embedded systems, introduced in 2006 because these systems combine the intelligence of a computer with the physical objects of our world.
- Component is very broad including software, hardware - digital hardware, analog circuits, mechanical hardware, power supply and distribution, sensors, and actuators.
- Behavior is embodied by the responses of its outputs to changes in its inputs. Both time and state are important factors.
- Microcontrollers which are microcomputers incorporating the processor, RAM, ROM and I/O ports into a single package, are often employed in an embedded system because of their low cost, small size, and low power requirements.



Good Enough Software, Soon Enough

- 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

Good Enough Software, Soon Enough

- Follow a good plan (Cont…)
  - 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

What happens when plan meets reality?

- We want a robust plan which considers likely risks
  - What if the code turns out to be a lot more complex than we expected?
  - What if there is a bug in our code (or a library)?
  - What if the system doesn't have enough memory or throughput?
  - What if the system is too expensive?
  - What if the lead developer quits?
  - What if the lead developer is incompetent, lazy, or both (and *won't* quit!)?
  - What if the rest of the team gets sick?
  - What if the customer adds new requirements?
  - What if the customer wants the product two months early?

- **Successful software engineering depends on balancing many factors, many of which are non-technical!**

Risk Reduction

- Plan to the work to accommodate risks

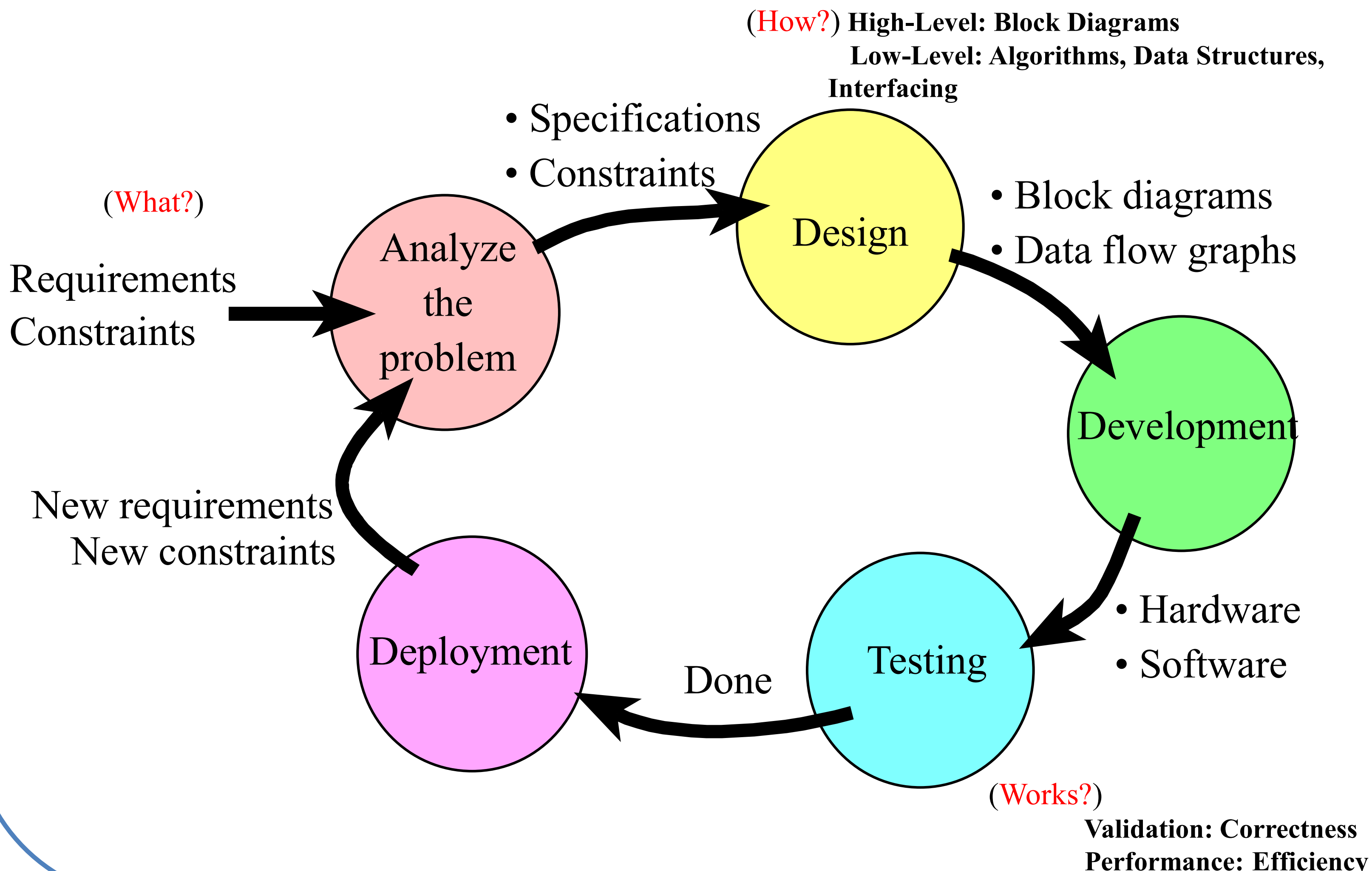
- Identify likely risks up front
  - Historical problem areas
  - New implementation technologies
  - New product features
  - New product line

- Severity of risk is combination of likelihood and impact of failure

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 - *Extensive downstream modification and maintenance!*
  - Corrections, adaptations, enhancements & preventive maintenance

Product life Cycle



Product life Cycle (Cont…)

- Requirement: a specific parameter that the system must satisfy.
  (informal description of what customer wants)
- Specifications: detailed parameters describing how the system should work. (size, weight, position, etc…)
  (precise description of what design team should deliver)
- Constraint: a limitation, within which the system must operate. (Cost).
- Safety: The risk to humans or the environment.
- Accuracy: The difference between the expected truth and the actual parameter.
- Precision: The number of distinguishable measurements. (Quantity)
- Resolution: The smallest change that can be reliably detected. (Quality)
- Response time: The time between a triggering event and the resulting action.
- Bandwidth: The amount of information processed per time.

Product life Cycle (Cont…)

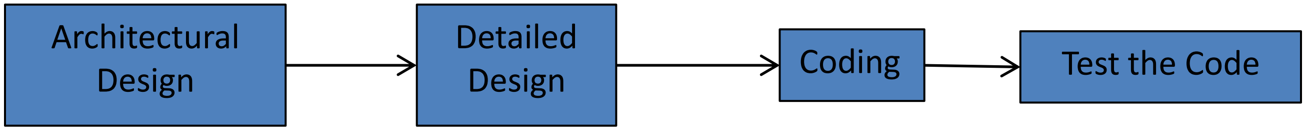
- Maintainability: The flexibility with which the device can be modified.
- Testability: The ease with which proper operation of the device can be verified.
- Compatibility: The conformance of the device to existing standards.
- Mean time between failure: The reliability of the device, the life of a product.
- Size and weight: The physical space required by the system.
- Power: The amount of energy it takes to operate the system.
- Unit cost: The cost required to manufacture one additional product.
- Time-to-prototype: The time required to design, build, and test an example system.
- Time-to-market: The time required to deliver the product to the customer.
- Human factors: The degree to which our customers like/appreciate the product.

Design Before Coding

- Ganssle's reason #9: ***Starting coding too soon***
- Underestimating the complexity of the needed software is a very common risk
- Writing code locks you in to specific implementations
  - Starting too early may paint you into a corner
- Benefits of **designing** system before **coding**
  - Get early insight into system's complexity, allowing more accurate effort estimation and scheduling
  - Can use design diagrams rather than code to discuss what system should do and how.
  - Can use design diagrams in documentation to simplify code maintenance and reduce risks of staff turnover

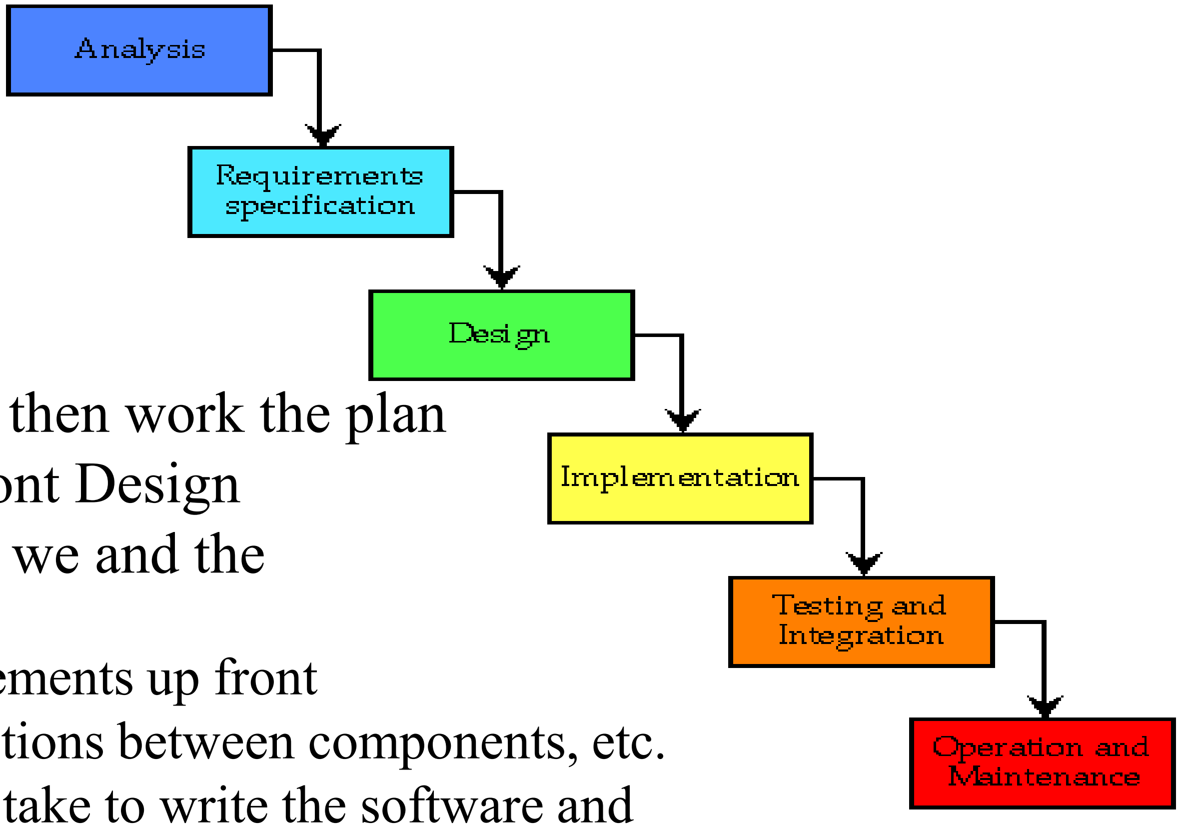
*Jack Ganssle in an internationally-recognized embedded systems engineer, author and speaker. See http://www.ganssle.com/articles/jackstoptenlist.htm for more on this.

Development Models



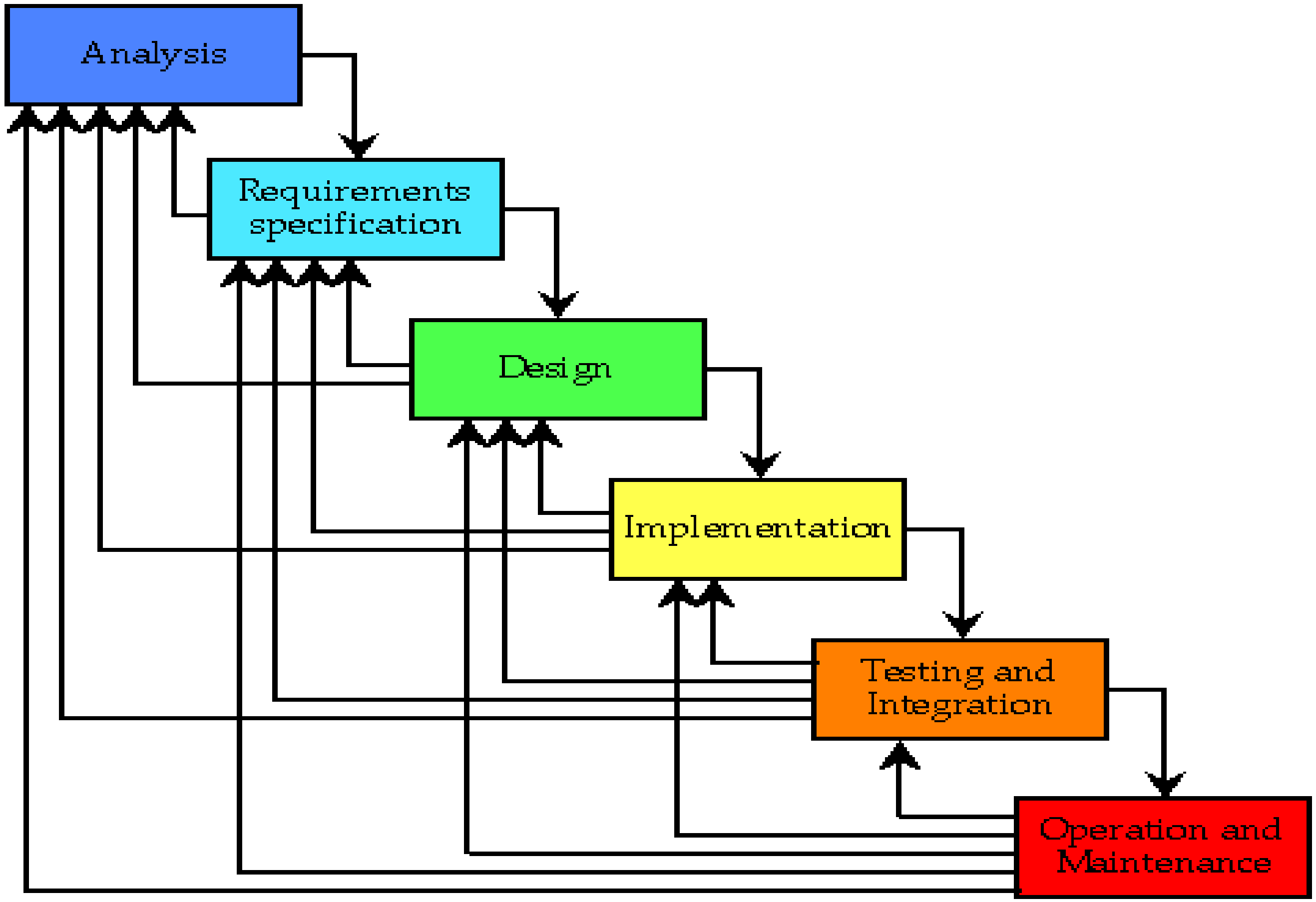
- How do we schedule these pieces?
- Consider amount of development risk
  - New MCU?
  - Exceptional requirements (throughput, power, safety certification, etc.)
  - New product?
  - New customer?
  - Changing requirements?
- Choose model based on risk
  - Low: Can create detailed plan. Big-up-front design, waterfall
  - High: Use iterative or Agile development method, spiral. Prototype high-risk parts first

Waterfall (Idealized)

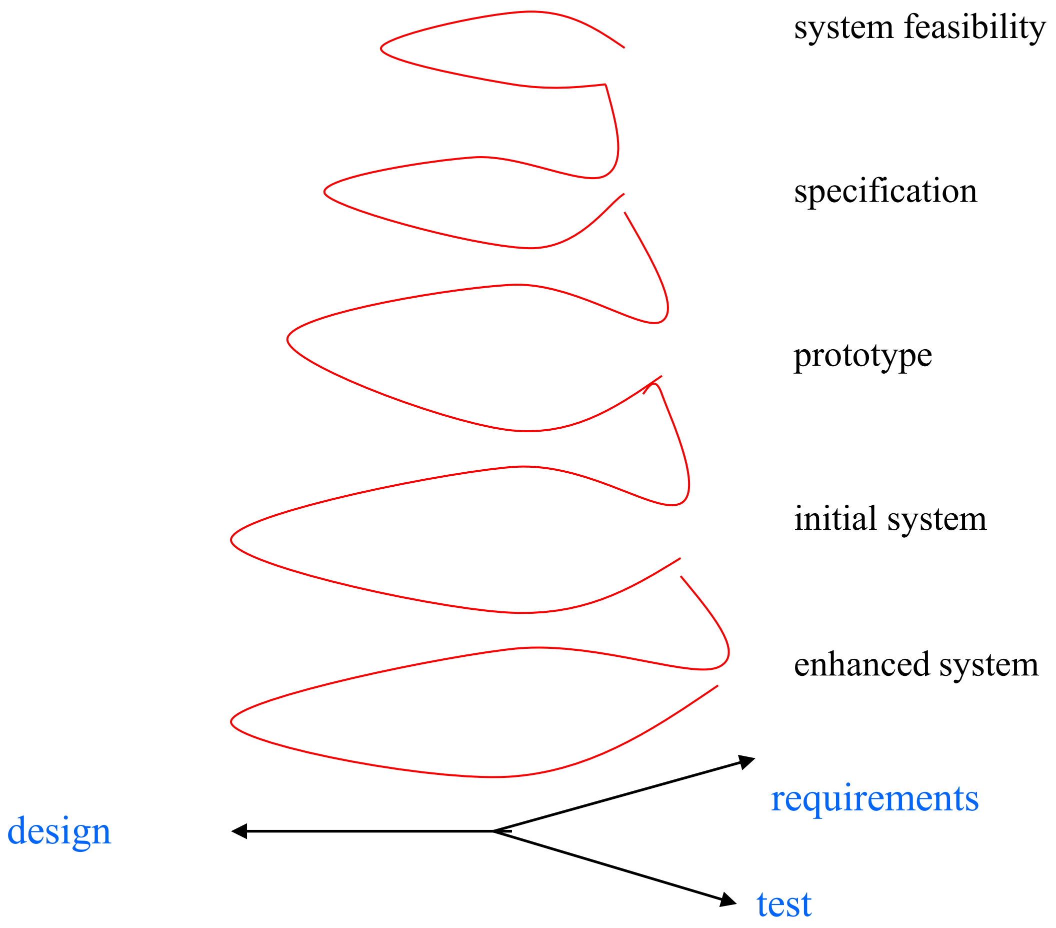


- Plan the work, and then work the plan
- BUFD: Big Up-Front Design
- Model implies that we and the customers know
  - All of the requirements up front
  - All of the interactions between components, etc.
  - How long it will take to write the software and debug it

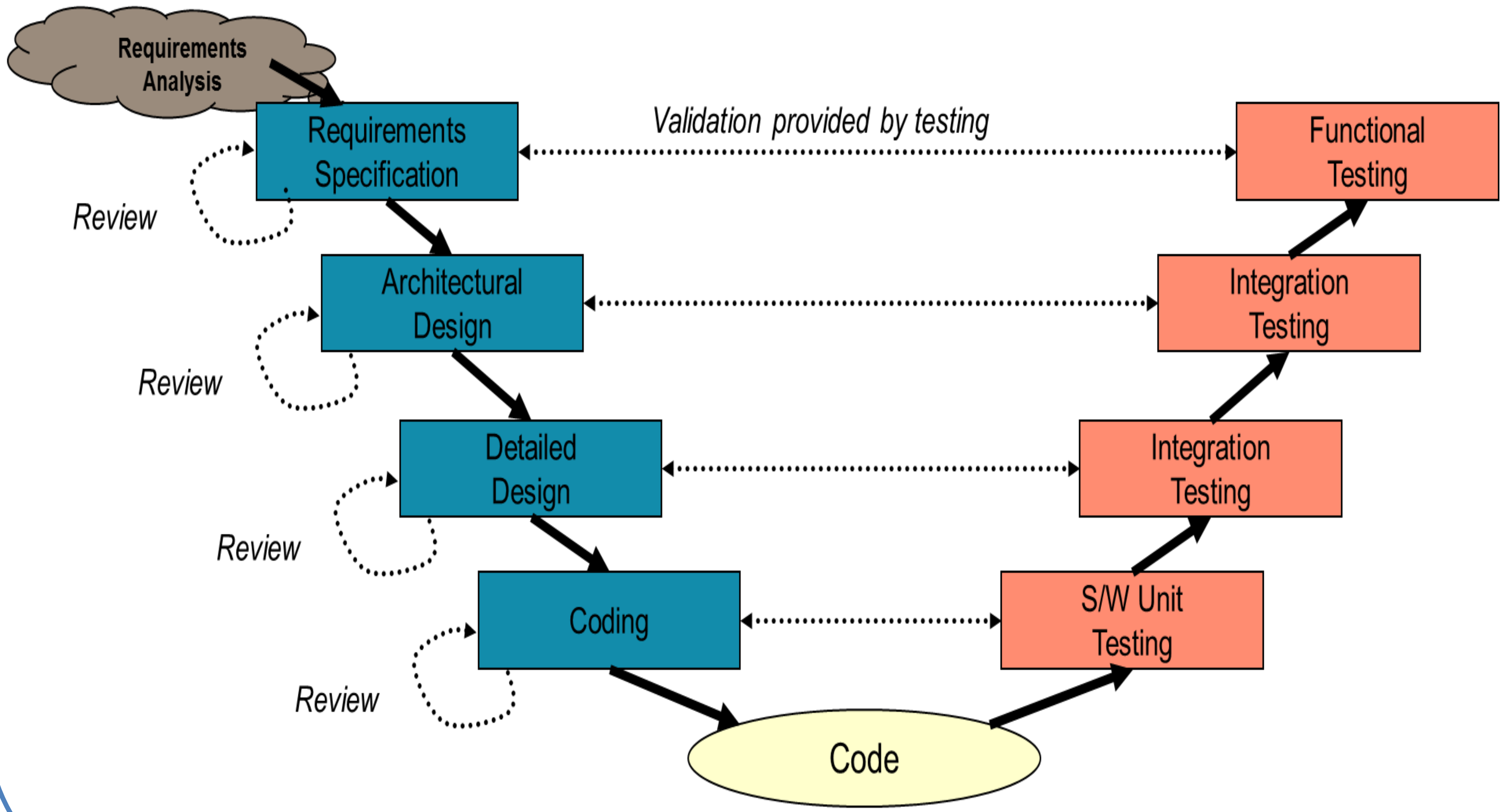
Waterfall (As Implemented)



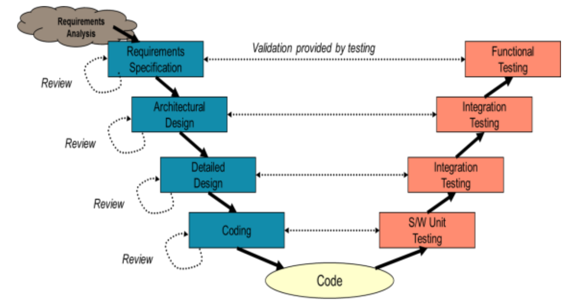
Spiral


system feasibility
specification
prototype
initial system
enhanced system
requirements
design
test

V Model Overview



1. Requirements Specification and Validation Plan



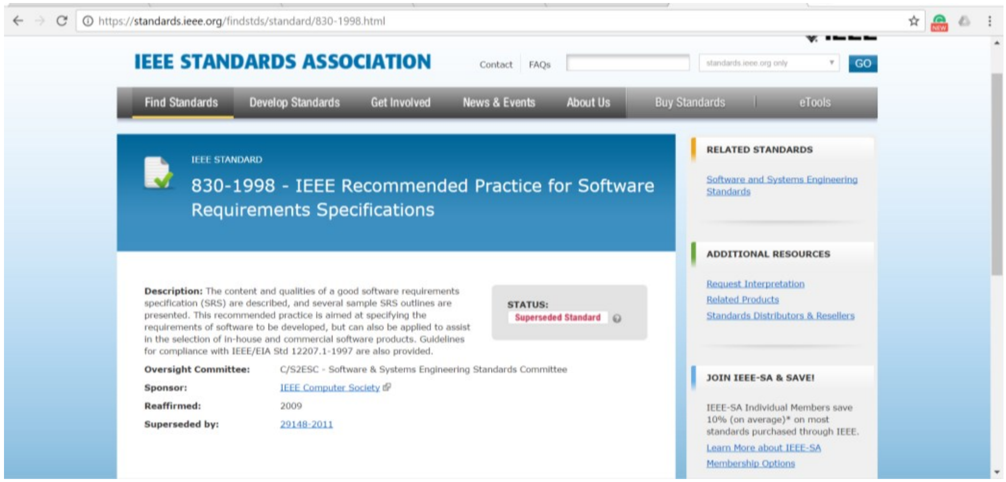
- Should contain:
  - *Introduction* with goals and objectives of system
  - *Description of problem* to solve
  - *Functional description*
    - provides a “processing narrative” per function
    - lists and justifies design constraints
    - explains performance requirements
  - *Behavioral description* 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 done yet?*)
    - is the foundation for a validation test plan
  - *Bibliography and Appendix* refer to all documents related to project and provide supplementary information

Good requirements

- Correct.
- Clear. Unambiguous.
- Complete.
- Verifiable: is each requirement satisfied in the final system ?
- Consistent: requirements do not contradict each other.
- Modifiable: can update requirements easily.
- Traceable:
  - know why each requirement exists;
  - go from source documents to requirements;
  - go from requirement to implementation;
  - back from implementation to requirement.

Requirements Document

- The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract.
- Write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable.
- IEEE templates can be used to define a project (IEEE STD 830-1998).



Requirements Document (Cont…)

1. Overview
   1.1. Objectives: Why are we doing this project? What is the purpose?
   1.2. Process: How will the project be developed?
   1.3. Roles and Responsibilities: Who will do what? Who are the clients?
   1.4. Interactions with Existing Systems: How will it fit in?
   1.5. Terminology: Define terms used in the document.
   1.6. Security: How will intellectual property be managed?

2. Function Description
   2.1. Functionality: What will the system do precisely?
   2.2. Scope: List the phases and what will be delivered in each phase.
   2.3. Prototypes: How will intermediate progress be demonstrated?
   2.4. Performance: Define the measures and describe how they will be determined.

Requirements Document (Cont…)

2.5. Usability: Describe the interfaces. Be quantitative if possible.

2.6. Safety: Explain any safety requirements and how they will be measured.

3. Deliverables

3.1. Reports: How will the system be described?

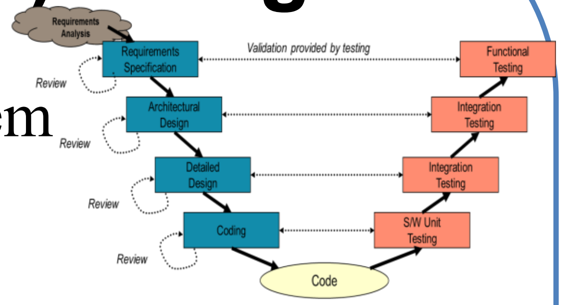
3.2. Audits: How will the clients evaluate progress?

3.3. Outcomes: What are the deliverables? How do we know when it is done?

Types of requirements

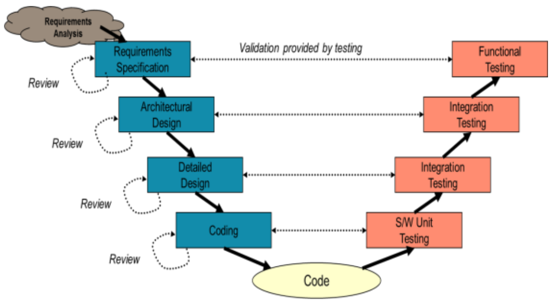
- Functional
  - input/output relationships. (what the system needs to do)
- Non-functional:
  - Timing.
  - Power consumption,
  - Manufacturing cost.
  - Physical size.
  - Time-to-market.
  - Reliability. (emergent system behaviors)
- Constraints
  - what limits the design choices

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

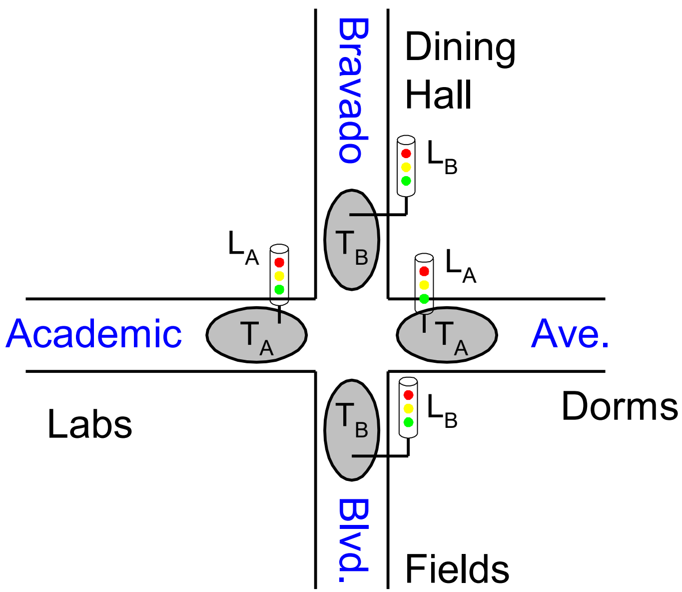
3. Detailed Design

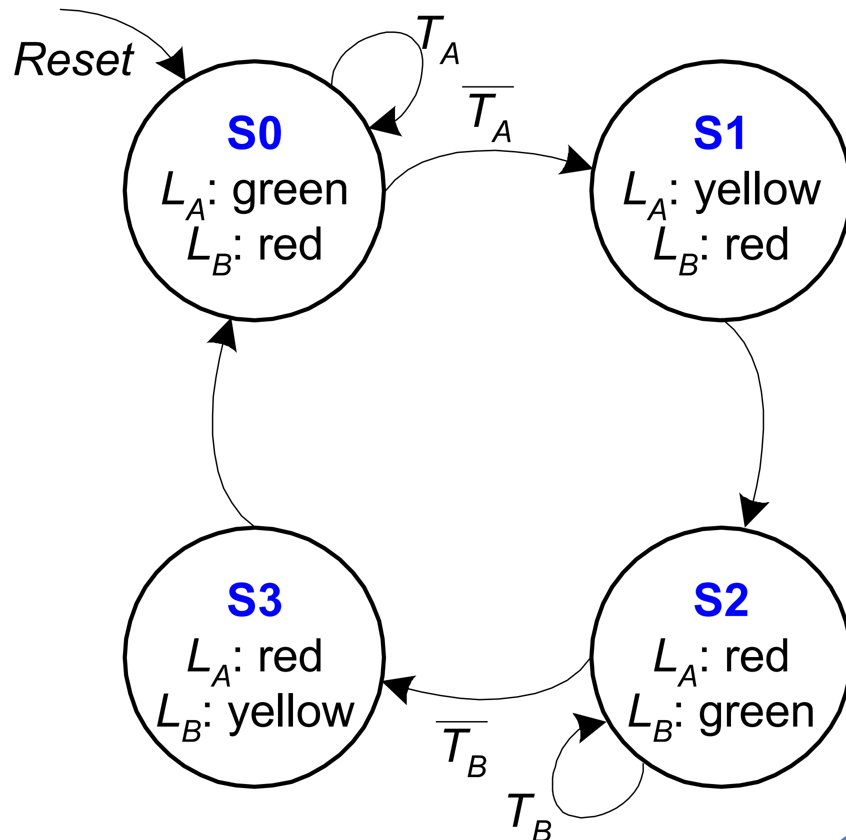


- Describe aspects of how system behaves
  - Flow charts for control or data
  - State machine diagram
  - Event sequences
- Graphical representations very helpful
  - Can provide clear, single-page visualization of what system component should do
- Unified Modeling Language (UML)
  - Provides many types of diagrams
  - Some are useful for embedded system design to describe structure or behavior

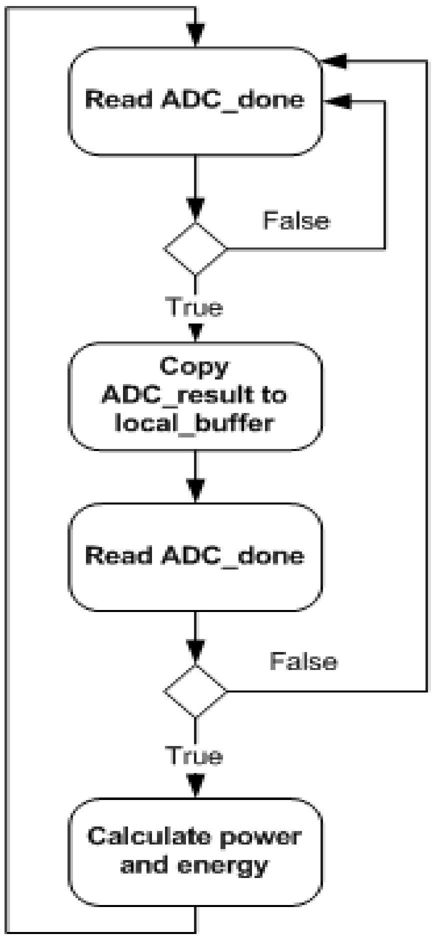
State Machine

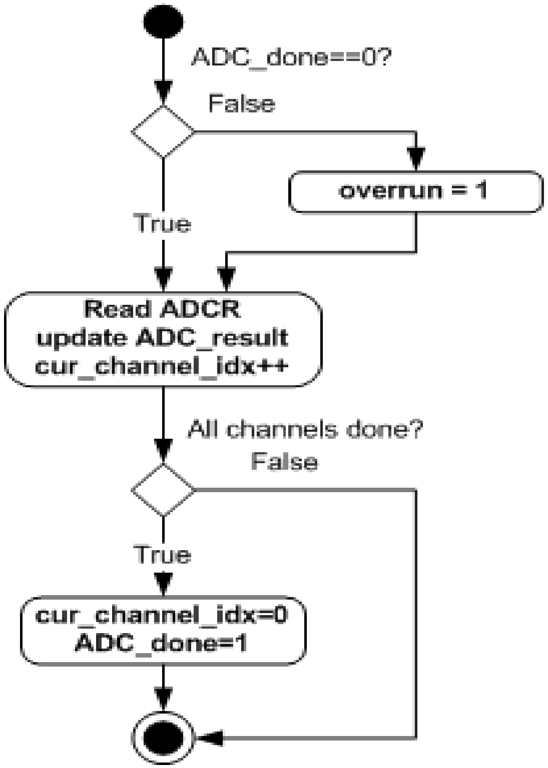
- **Moore / Mealy FSM**
- **States:** Circles
- **Transitions:** Arcs



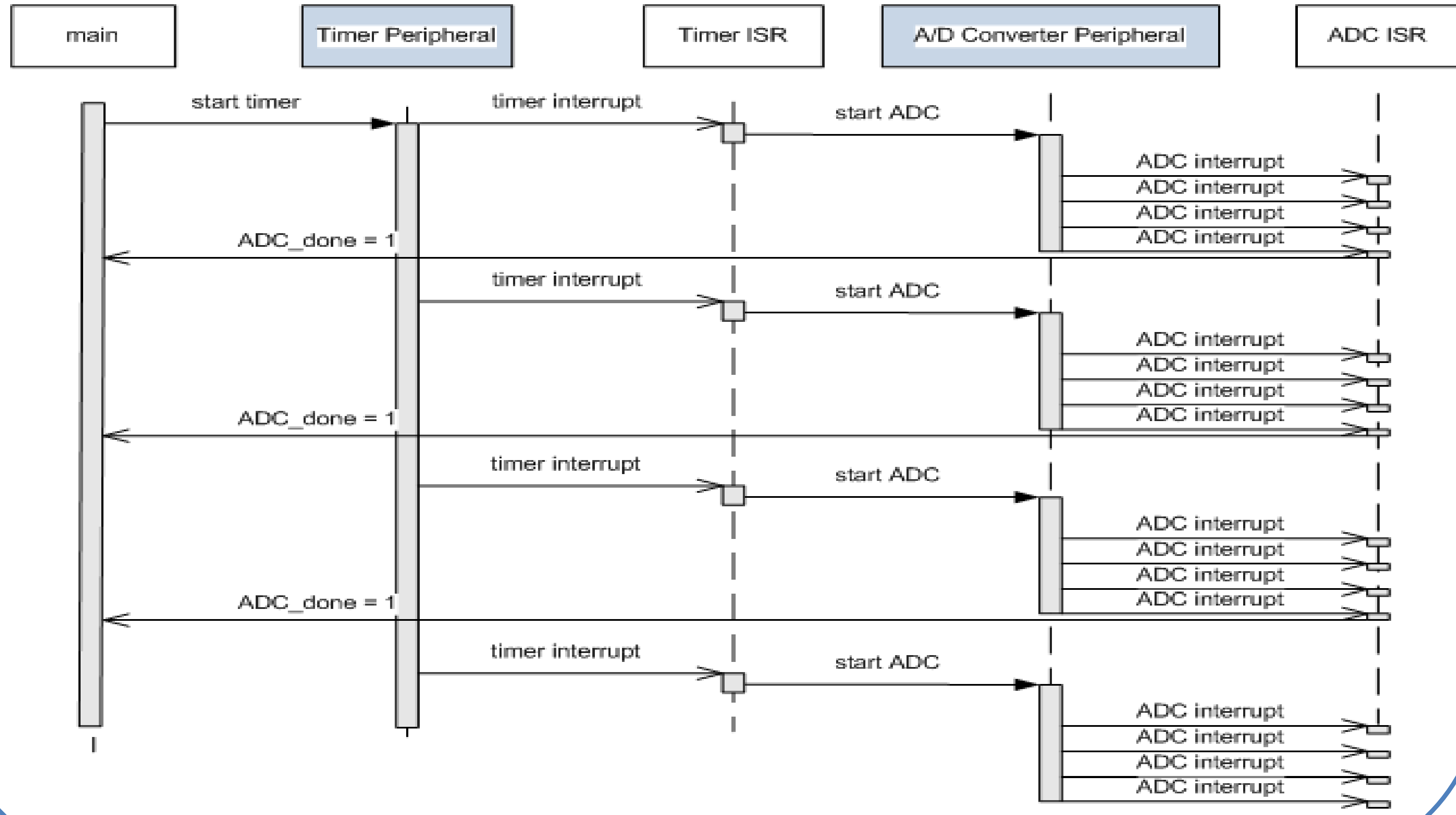


Flowcharts

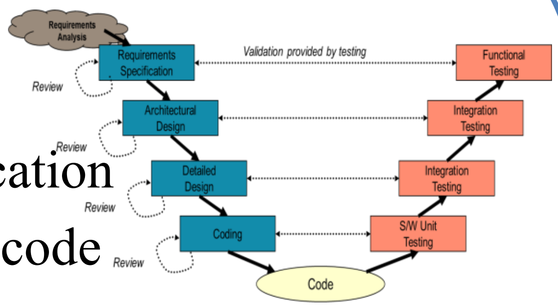

Read ADC_done
False
True
Copy ADC_result to local_buffer
Read ADC_done
False
True
Calculate power and energy


ADC_done==0?
False
overrun = 1
True
Read ADCR update ADC_result cur_channel_idx++
All channels done?
False
True
cur_channel_idx=0 ADC_done=1

Sequence of Interactions between Components

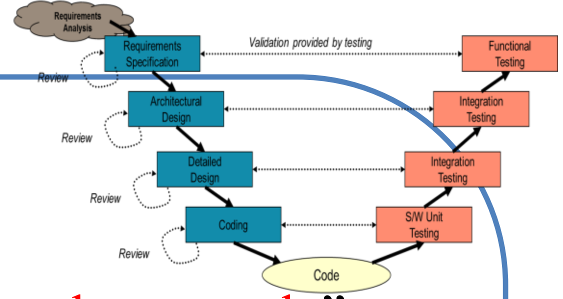


4. Coding and Code Inspections



- Coding driven directly by Detailed Design Specification
- Use a version control system while developing the code
- Follow a coding standard
- Perform code reviews
  - Peer Code Review
    - IBM removed 82% of bugs
    - 80% of the errors detected by HP's inspections were unlikely to be caught by testing
- Test effectively
- Automation
- Regression testing

5. Software Testing



- Testing IS NOT "the process of verifying 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 correctly (impossible)
    - 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 the ones which matter
  - 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.

5. Software Testing - 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

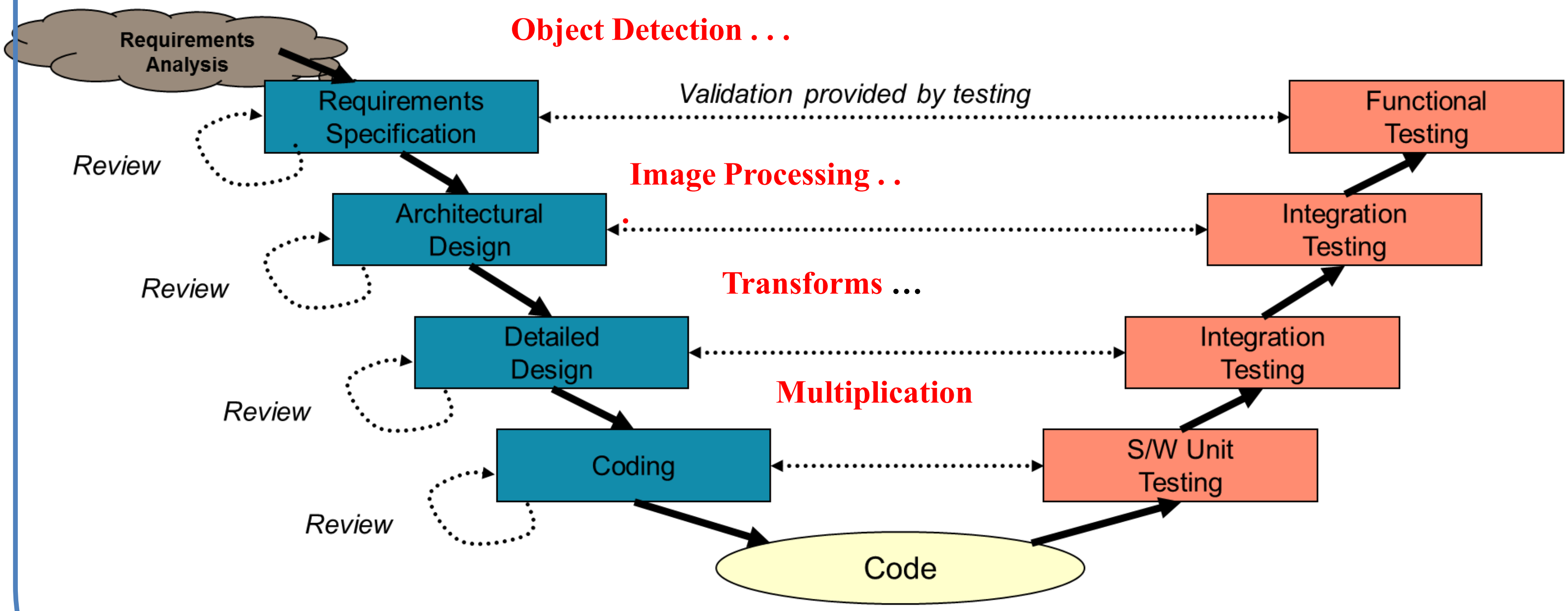
Tesla Crash

Tesla model s autopilot strikes again in dallas crash

Audi A8

..\AudiA8.mp4

V Model Overview



Assignment no. 3

kindly read the following paper [Software Engineering for Space Exploration]. In short, one paper only (2 pages), write an essay mention your opinion about the topic.

Notes:

- you will deliver your report at lecture time.
- you can work in a group but the group is only two students.
- you may need to read more - paper references or external resources.
- at lecture time, there will be a discussion regarding the topic, be ready to present the topic and discuss it.