Building Whole Systems: an brief overview

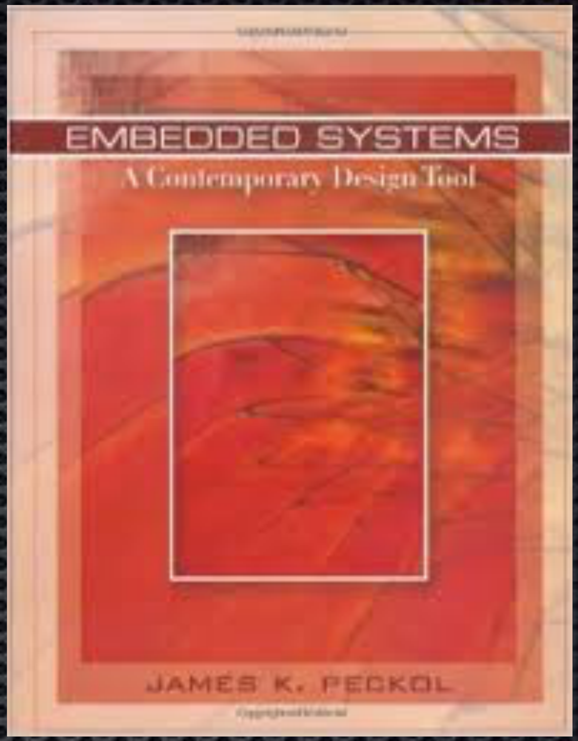
EDAN85 Embedded Systems Design - Continuation (Advanced) Course, Lecture 2

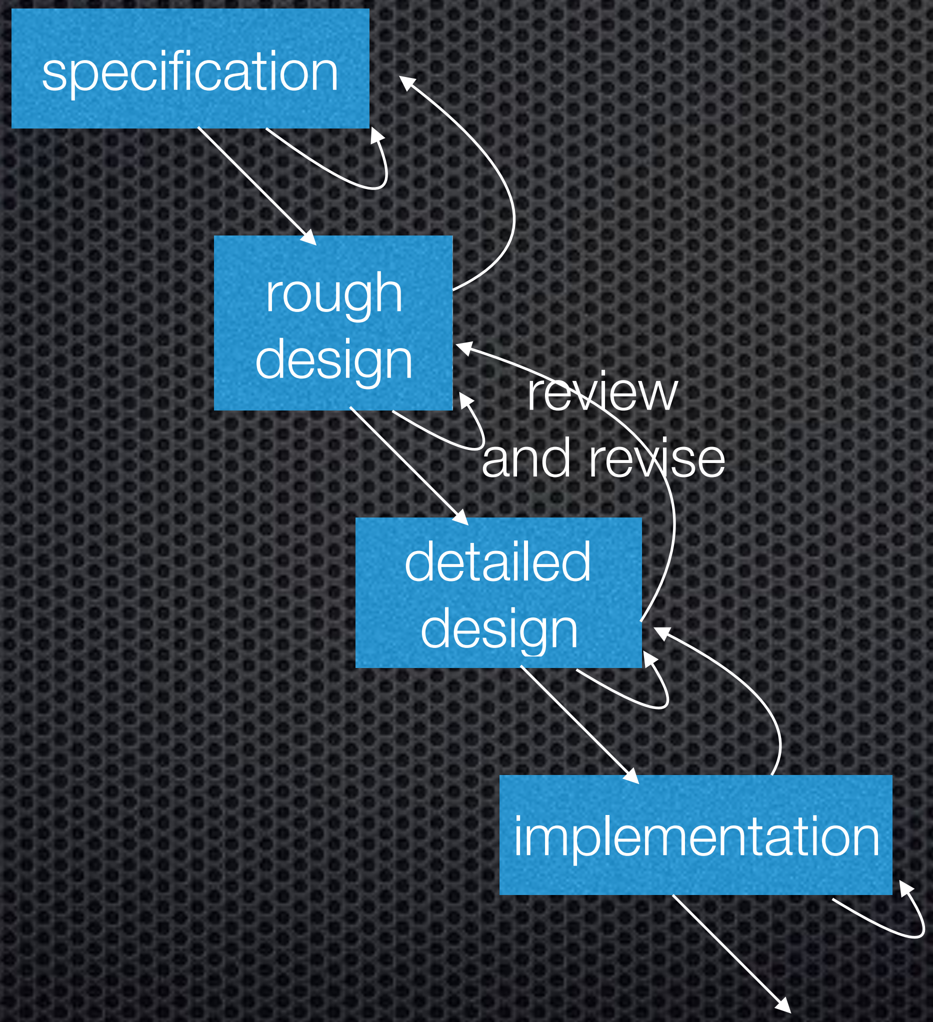
Lecture 2 Contents

- Life-cycle models or the five steps to design
    1. requirements definition
    2. system specification
    3. functional design
    4. architectural design
    5. prototyping
- Typical Issues and Solutions
- Working Tips

Life-cycle models

- ways to divide the design and development of a system into smaller steps/phases

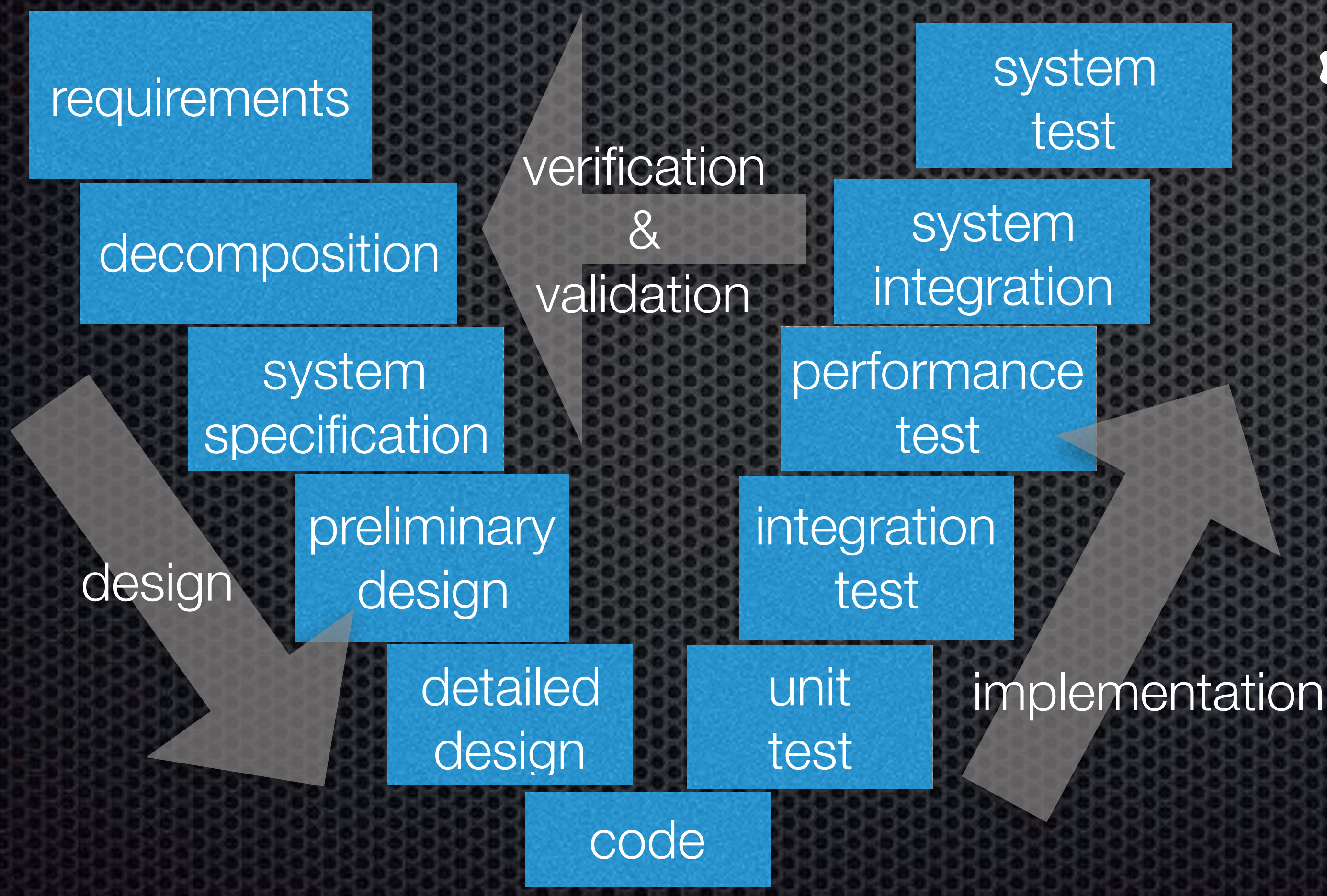




- **Waterfall**
  - complete a phase, then move on
  - unrealistic (real-world design is iterative)

Life-cycle models

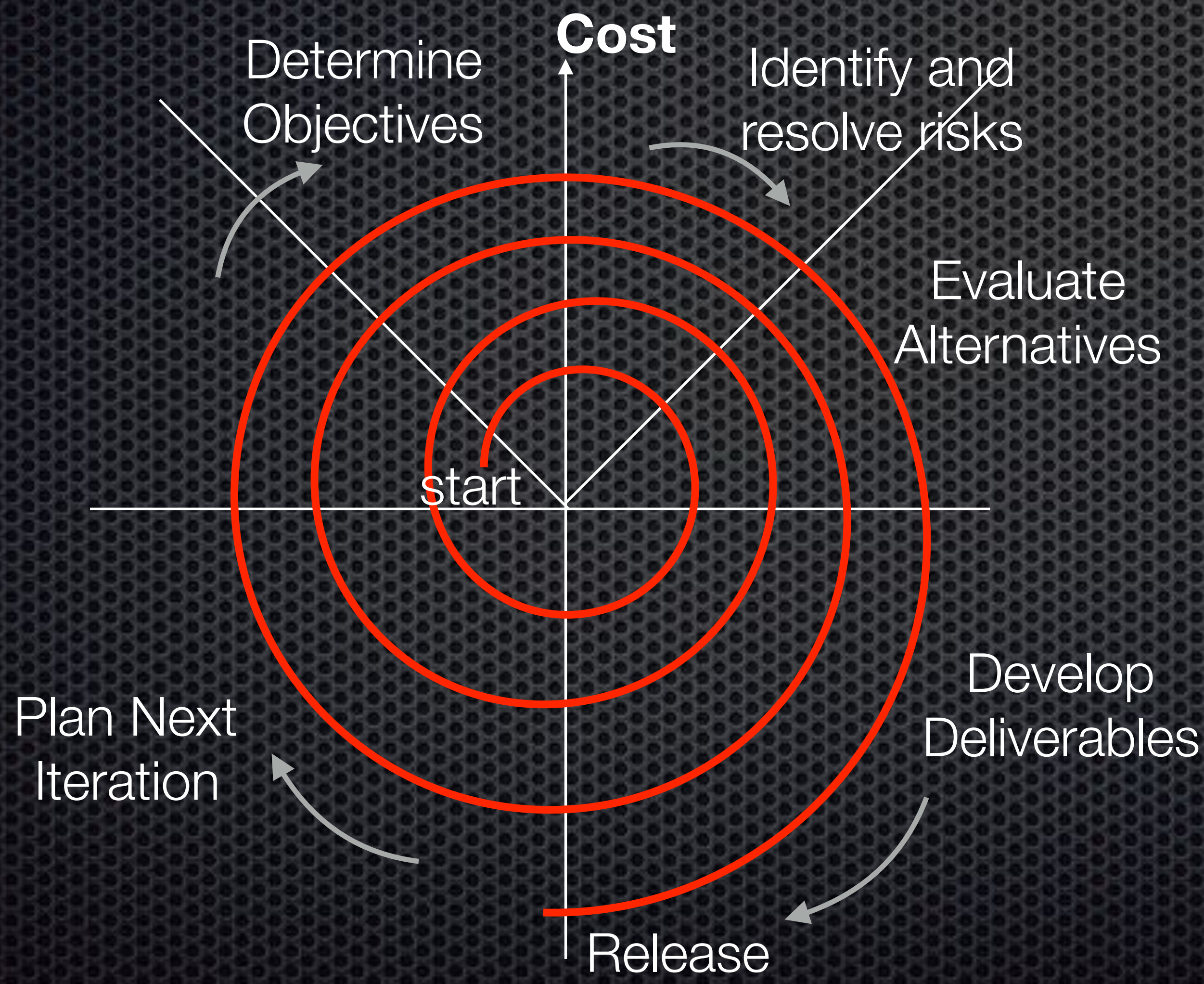
- ways to divide the design and development of a system into smaller steps/phases



- **Waterfall**
- **V Cycle**
  - somewhat similar to Waterfall
  - emphasis on testing

Life-cycle models

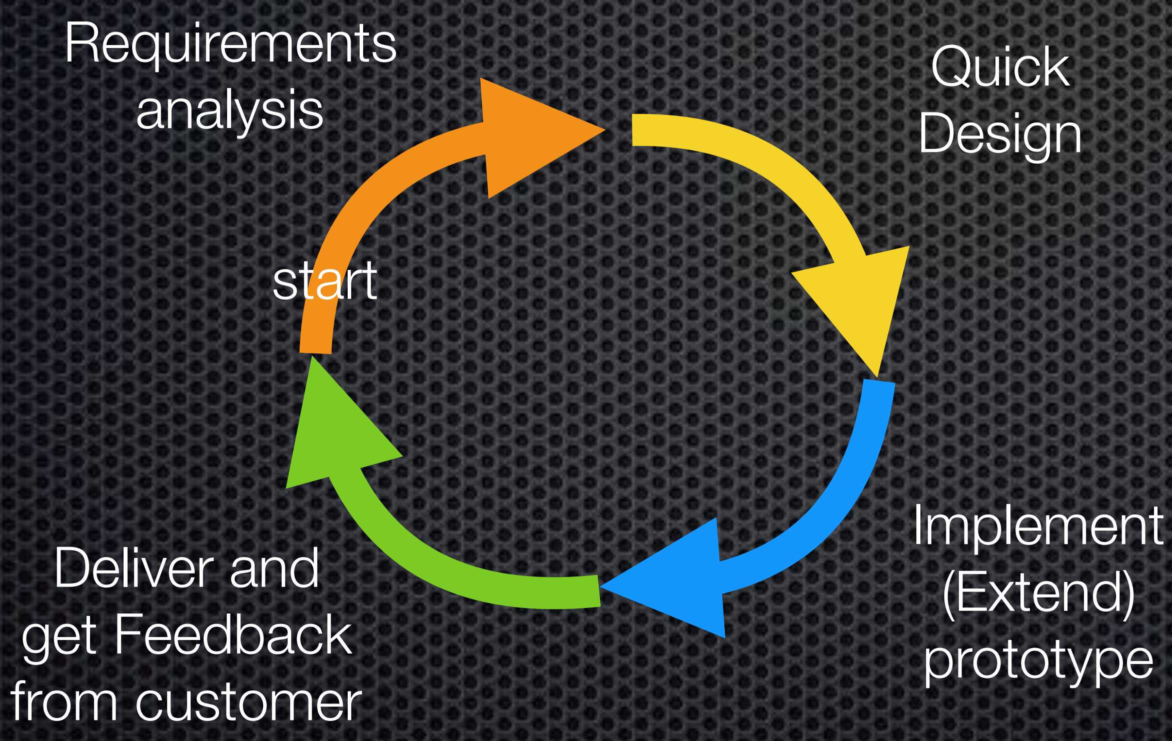
- ways to divide the design and development of a system into smaller steps/phases



- Waterfall
- V Cycle
- **Spiral**
  - risk oriented view
  - start small, iterate

Life-cycle models

- ways to divide the design and development of a system into smaller steps/phases



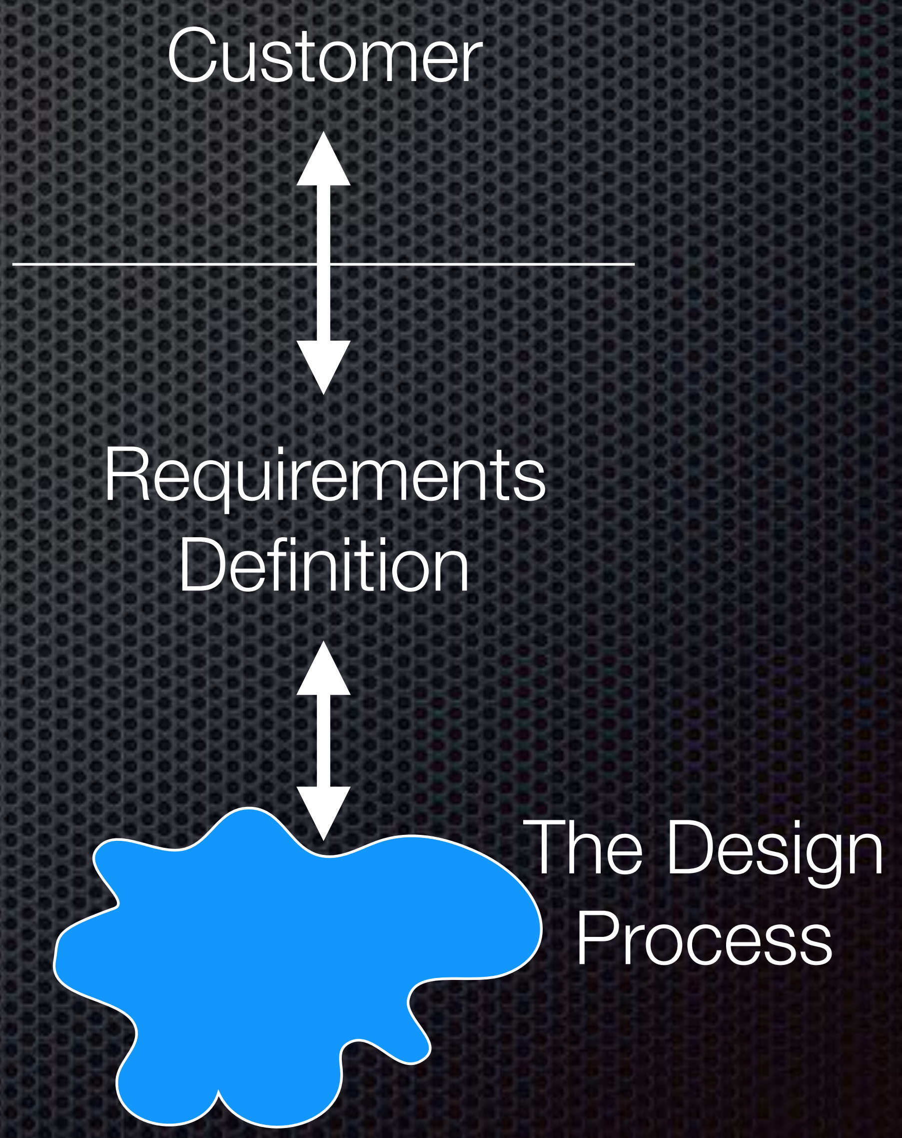
- **Waterfall**
- **V Cycle**
- **Spiral**
- **Rapid prototyping**
  - sometimes identified with "spiral"
  - the prototype should never turn into the final product

Five Steps

- Successful Design is based on:
  1. requirements definition
  2. system design specification
  3. functional design
  4. architectural design
  5. prototyping

1. Requirements

- identify **what** to do and **how well**, starting from the customer
- characterize the system and its role in the environment

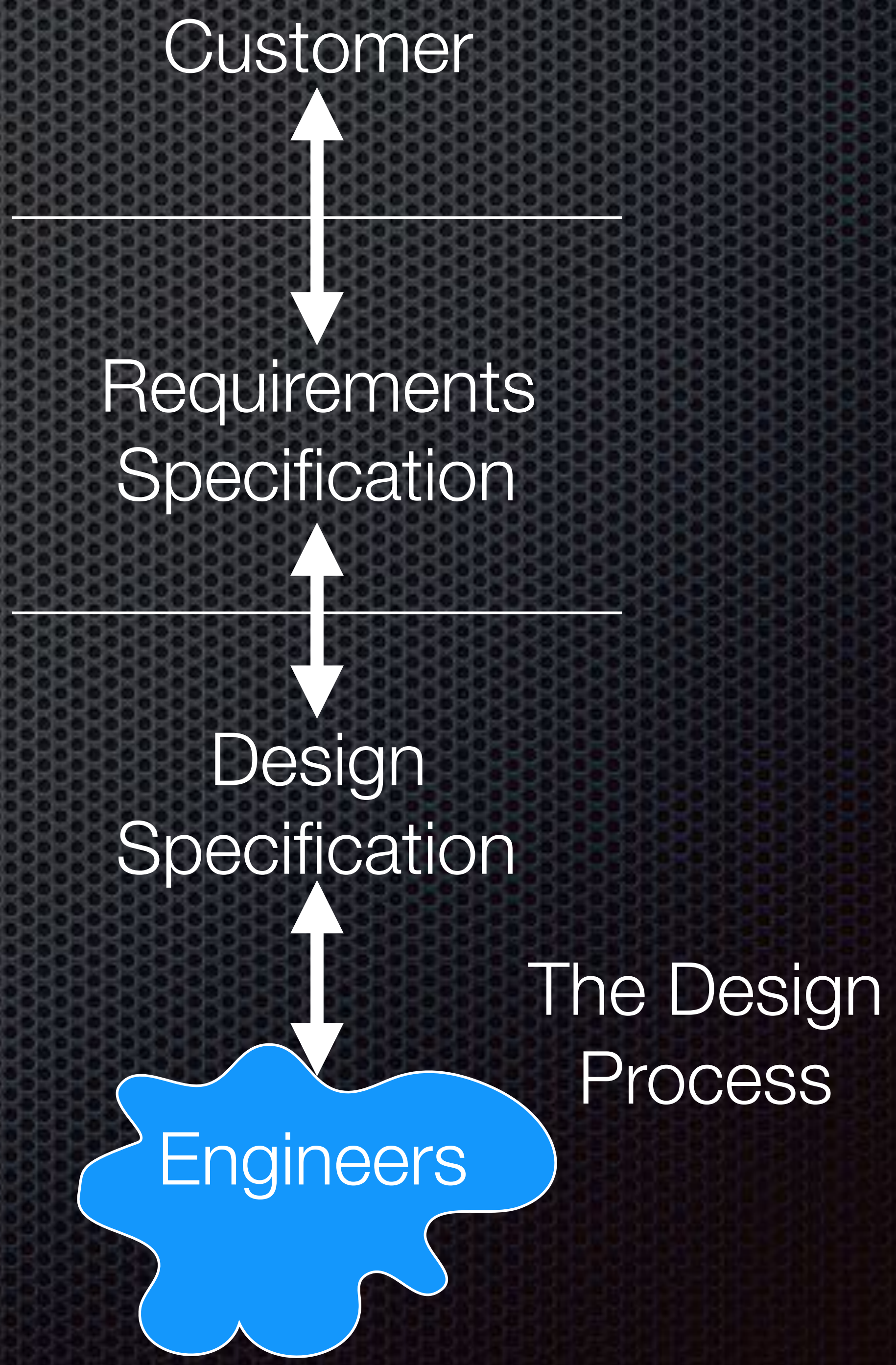


2. Design Specification

- Formalizes the requirements in a precise, unambiguous language

A. system's public interface (I/O) from inside the system

B. **how** are the I/O requirements met by internal functions



3. Functional Design

- find appropriate internal functional architecture for the system ☞ **a functional model**

Partition and decompose as necessary:

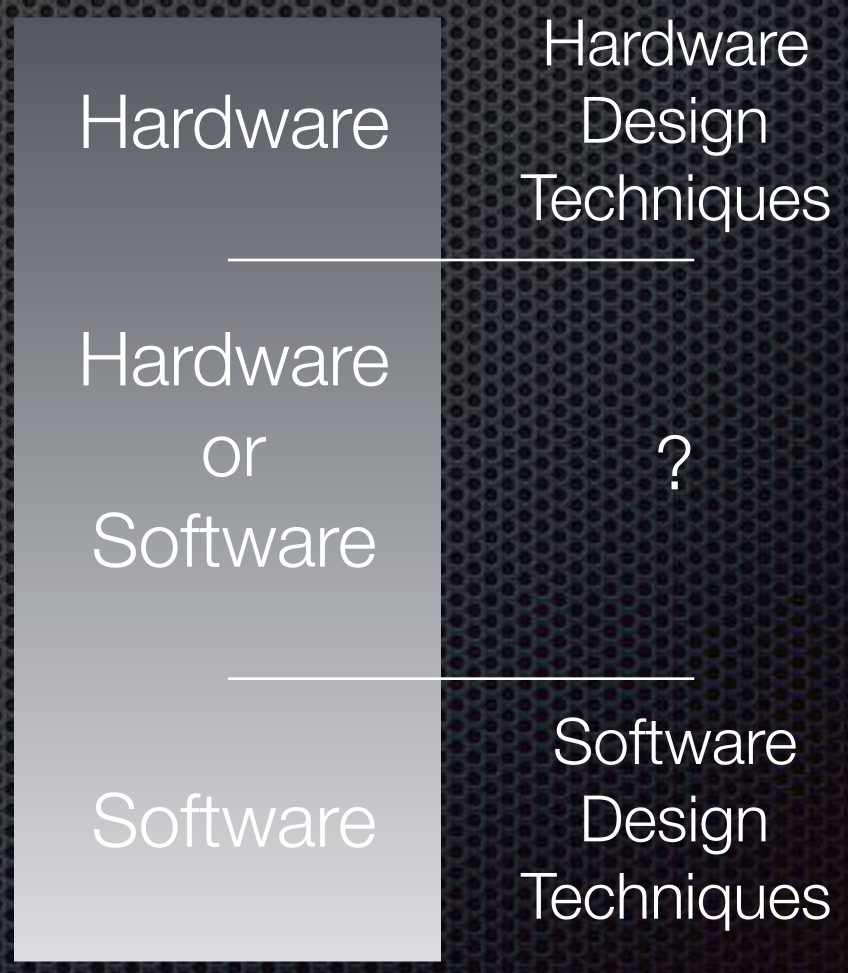
- minimize coupling (module interdependency)…or maximize cohesion
- progressively refine into smaller manageable modules (and interfaces)

4. Architectural Design

- map functions to hardware

constraints:

- geographical distribution
- physical and user interfaces
- legacy components and cost
- system performance needs
- timing and dependability needs
- power consumption



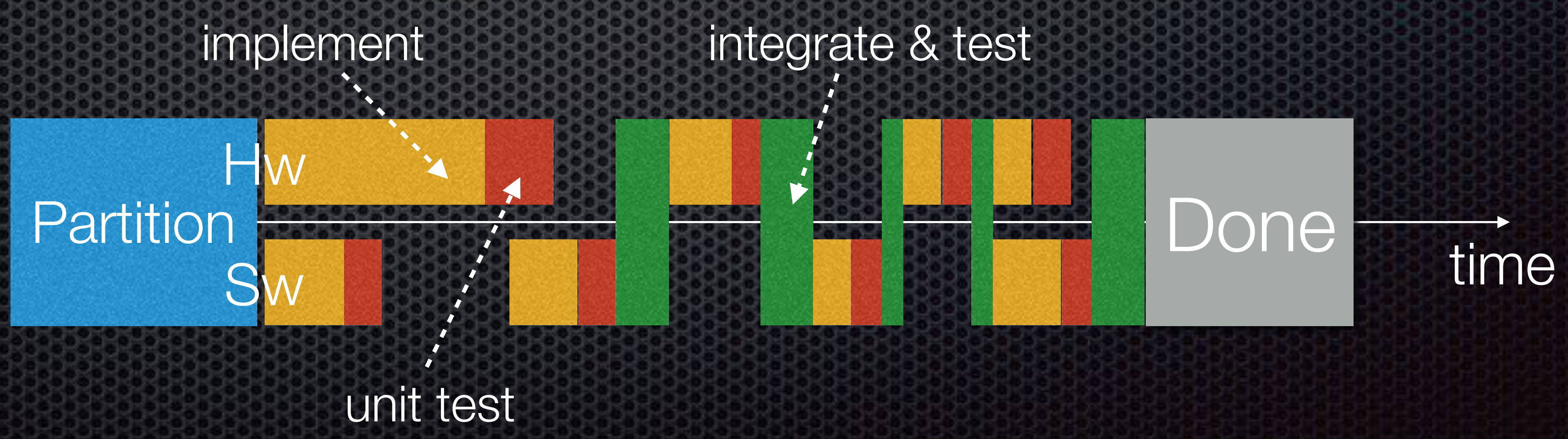
5. Prototyping

- bottom-up process: assemble parts, eliminate more and more of the abstract functionality
- purpose: understand/evaluate the system design
  - static analysis:
    coupling, cohesiveness, complexity
  - dynamic analysis:
    behavioral verification, performance, trade-offs

Typical
System Design Issues

- a mix of specifications at different levels of abstraction
- neither Hw or Sw are fully functional
- Sw needs a “Hw” for implementation

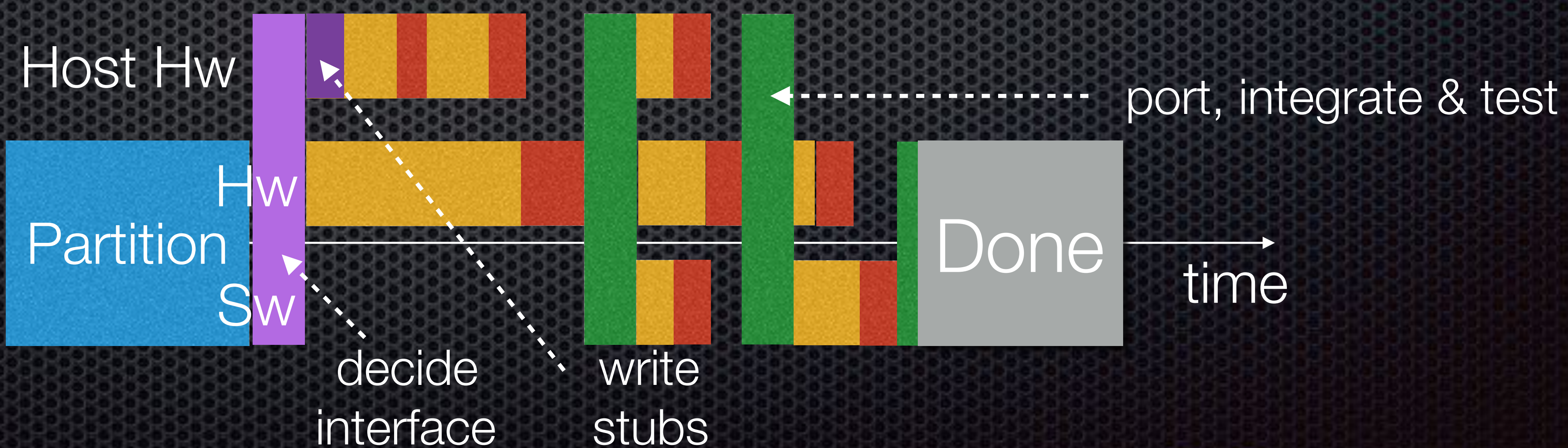
Target platform development (wait for Hw):



Improved System Development

Cross-platform development (after partitioning Hw/Sw):

- decide Hw/Sw interface and write stubs
- develop Sw on an available Hw + stubs
- develop Hw in parallel (along with tests in Sw)
- port stubs on Hw, integrate Sw and test again



An Example

- **Spec**: *…read a pair of integers (x,y) on the serial and display a pixel an the screen at (x,y)…*

“read integer from serial” and “put pixel” are Hw/Sw interfaces

Host Hw
desktop PC

Stubs (Interface)
int ReadInt() { scanf() }
void PutPixel(int,int) { printf() }

Target Hw
developed in parallel

(Host) Software
…
PutPixel(ReadInt(), ReadInt())
…

Target API
rewrite ReadInt and PutPixel for the target!

compile with the target API

Target Software

Concluding Tips (I)

- make an detailed initial specification: reduces confusion and other problems later on
- distribute the work in the team (Hw, Sw, Integration, Testing, etc.): more work gets done in parallel
- meet up/report/discuss your progress often (team, me): know who needs and who can give help before is too late

Concluding Tips (II)

- start from a working system (Hw+Sw) and build around it: a full re-design takes much more effort/time
- testing is essential: bugs are hard to detect and fix later
  - use unit tests, simulate modules thoroughly
  - use debuggers, printouts, leds, etc. to make sure your system works
  - write simple Sw tests for your Hw
- if time is short, go around problems: patch dodgy Hw in Sw