

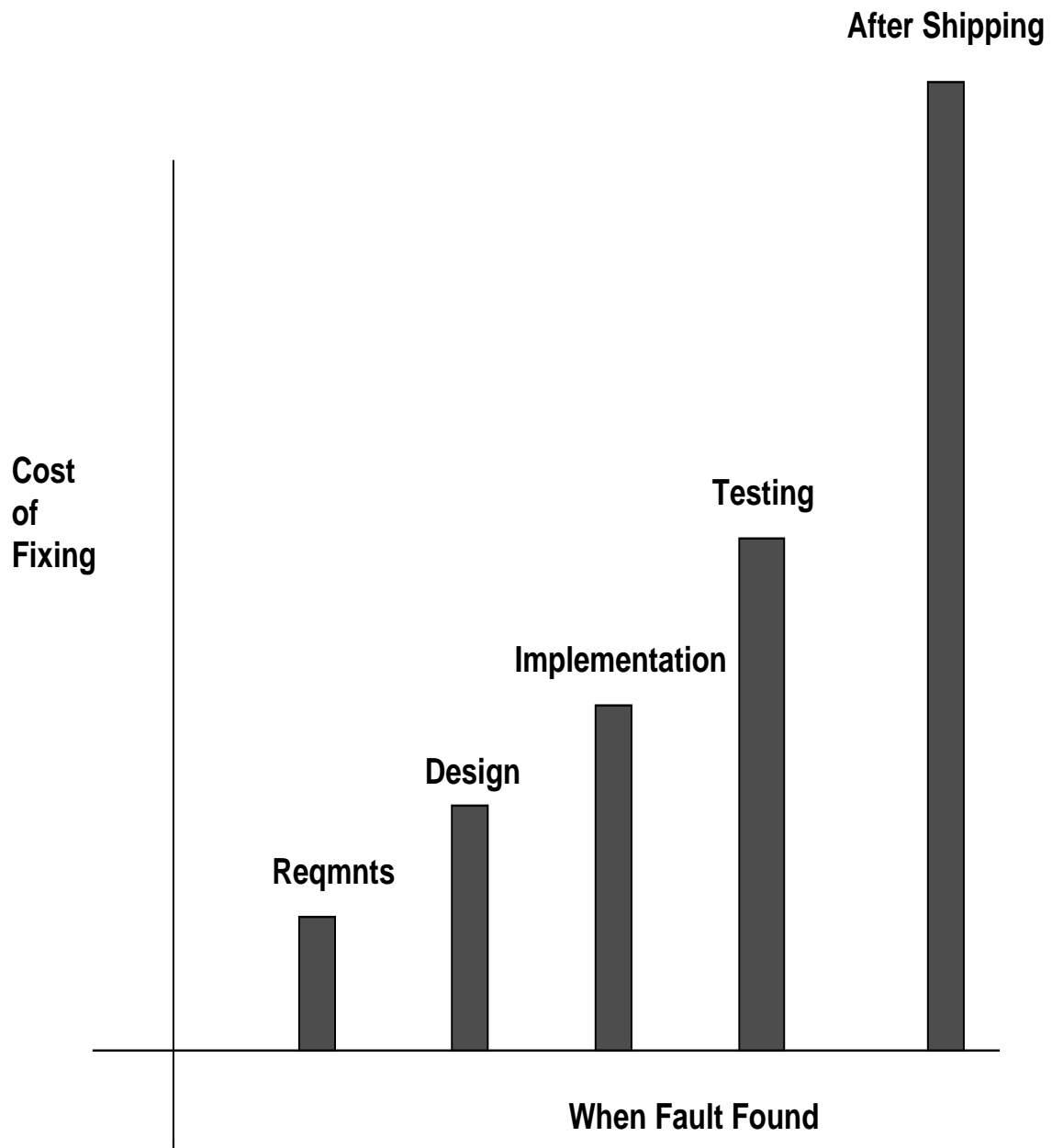
Requirements—Formal Approach

- Motivation: why get Requirements right.
- The A7E Case Study, and why
- **Lessons:** How to royally screw up on requirements.
- Getting it Right—Approach 2 (A7E Method *today*).

Reference:

HENNINGER, KATHRYN, L, “Specifying Software Requirements for Complex Systems, New Techniques and their applications”, IEEE Transactions on Software Engineering, January 1980. Vol SE6, No 1.

Why



The A7E Document

Context:

- Late 1970's: 120,000 Assembly instructions, IBM Minicomputer, 16K bytes of main memory. Second implementation. Guess what lifecycle?
- Embedded Control System: 22 Devices connected; safety-critical, with real-time deadlines.
 - *Input*: sensors, cockpit switches, keyboard
 - *Output*: heads-up disp., targeting, Steering Cues, etc
- Two types of functions: *periodic*, and *event-response*. Analogy to biological systems.
- Re-design and rebuilt of existing system.
 - Functionally identical to old program!
 - Must pass same tests, have same “footprint”, etc

Why study this?

- One of the earliest attempts to identify requirements analysis as a separate activity and study it.
- The results are widely acknowledged to be very successful.
- Good documentation of goals, methodology, results, and main lessons learned.
- A simple introduction to formal approaches.

Question: Why waterfall? Here?

Reminder: What was in the document

- Requirements document help communication between stakeholders.
 - *Depending on which process: waterfall, prototype, etc.*
- The document contains:
 - *Functional Requirements:* The system must control targeting on the main gun using the pilot's helmet
 - *Non-Functional Requirements:* The system must respond to pilot's command in under 200 ms
 - *Platform/Architecture Requirements:* The system must run on PowerPC 750 and be CORBA-compatible

How to write a bad requirements doc?

1. Say different things in different places *e.g....use same words with different meanings*
2. Make the document hard to change. (*e.g.*, “when speed is between 0 and 5 knots. . .” all over the document)
3. Leave important (non-direct) functional details out (*e.g.*, *what happens when the gun jams?*)
4. Leave out important external non-functional requirements: like cost, performance, etc; characteristics of environment (engines, guns, control surfaces etc).
5. Talk about internal details of the system *Internal architecture, not what it does—(How about CORBA compatibility?)*.
6. Don't maintain it as lifecycle proceeds *e.g.*, *developers fix a particular feature interaction that was noticed, but don't modify document*
7. Don't get all the right people involved.

Consistency & Maintainability Techniques

- Use the same terminology everywhere: this promotes understandability, consistency, and makes the document easy to read, write, and check.
 - *Data Items*: `/input-data-item/`, `//output-data-item//`, `$string-value$`. Each one described using a standard document template.
 - *Events*:
Described as changes in values, using conditions:
`@T(/Fuel/ ≤ !Min-Cap! && $Targeting$ = "engaged")`
`@F(...)`
- Use Text Macros: this makes the document easier to maintain.
 - *Example*: `!Stall! = /Air-Speed/ ≤ 85 knots`

Dealing with Complexity

- Too many variables;
- Too many “functions” from inputs to outputs
- Too many “internal states”.

Solution: Simplify using “Modes”. Use significant mission states: cruising, terrain-following, engaged, stealth, etc to organize requirements.

- Modes defined in terms of “Mode Condition Tables”: which modes are active in which states of *input variable* values?
- Recognized two types of functions: *periodic* and *event-driven*.
- Computation of *periodic* (“homeostasis”) functions defined in “Function Condition Tables”
- Responses to *events* defined in “Event Condition Tables”.

Completeness Techniques

Different types of Tables were used:

Mode Condition Tables:

MODE	/\$Terrain Follow\$/	/\$Autopilot\$/	/Altitude/	/Latitude/
Manual	OFF	OFF	—x—	—x—
Cruise	OFF	ON	$\succ 100\text{m}$	$\prec 70^0$
Terrain Following	ON	—x—	$\prec 100\text{m}$	$\prec 70^0$
...

The above says: row 1) In manual mode, the “terrain follow” and “auto pilot” switches are off, and it doesn’t matter where or how high the plane is. 2) In cruise mode, the “terrain follow” switch is off, and the altitude is over 100m and the latitude is less than 70 degrees. etc.

Note:

- Some entries may have “don’t-care” conditions.
- Can help spot inconsistencies and errors
- Can help find missing data conditions that are applicable to a mode.

Function Condition Table

Output Value Condition Tables: Specifies an output value for a given function in different mode conditions

Terrain Following	/Altitud/ \leq 100	/Altitud/ \succ 100	...
MODE2	Condition 1.1	Condition 1.2	...
MODE3	Condition 2.1	Condition 2.2	...
MODE4	Condition 3.1	Condition 3.2	...
//CompDisp//	Value 1	Value 2	Value 3

- Bottom row indicates different ways the function computes the output.
- Each column is a mutually exclusive condition for which the function works the same way.
- Each row is a different Major mode. Only one mode is active at a time.

Event Condition Table

Responses to events. Specifies state changes or other actions responding to input events.

Assume: !AltHigh! \leftrightarrow (/Altitude/ \leq 100m))

Terrain Following	@T(!AltHigh!)	@F(!AltHigh!)	...
MODE2	Condition 1.1	Condition 2.1	...
MODE3	Condition 2.1	Condition 2.2	...
MODE4	Condition 3.1	Condition 3.2	...
<i>Action</i>	//buzzer// := \$off\$	//buzzer// := \$on\$...

- Bottom row indicates the responses to events in various modes.
- Each column is a mutually exclusive condition which causes an identical response.
- Each row is a different Major mode (only one mode active at any time).

Why did this work with the A7E?

- System of moderate complexity.
 - 22 input/output variables, a small number of “modes”.
- Previous system and expertise available.
 - This was the second system of it’s kind.
 - Customers were experts in the domain.
- “Market” demand for high-quality system.
 - Demand for Quality, and the willingness to pay, and willingness to wait.
- Support for a “formal” requirements document.

Applicability

- The terminology/macro stuff is pretty widely applicable.
- Applicable to: Small Control Systems, Medical Devices, public transit vehicles (well-established, high-reliability, well-understood, customers are experts, moderately complex, slow-moving markets)
- Less applicable to:

Space Shuttle	<i>Too many variables</i>
Chess Player	<i>Too many States</i>
Telephone Switches	<i>not well-understood, too complex</i>
Video Games	<i>market too fast-moving</i>

Questions

So How about:

Video Game Simulation of A7E?

Radiation Therapy Equipment?

A Home Burglar Alarm ?

A Home Robot ?

Tax Computation Software ?

*Next class—Use Case Method, and using them with
Rapid prototypes—a very different approach*

Burglar Alarm example

- Variables:
 - input: `/intruder/`, `/$card-reader$/`,
 - output: `//$Alarm$//`, `//$buzzer$//`, `//$door$//`,
- Modes: BurglarAlarm (BA) is On, or Off.
- If BA is On, and someone is at the door for more than 5 seconds, without putting a card in, sound the alarm; otherwise, open the door to let them in, and sound the buzzer while the door is open.
- If BA is On, and someone opens the door from the inside, then sound the alarm *and* the buzzer.