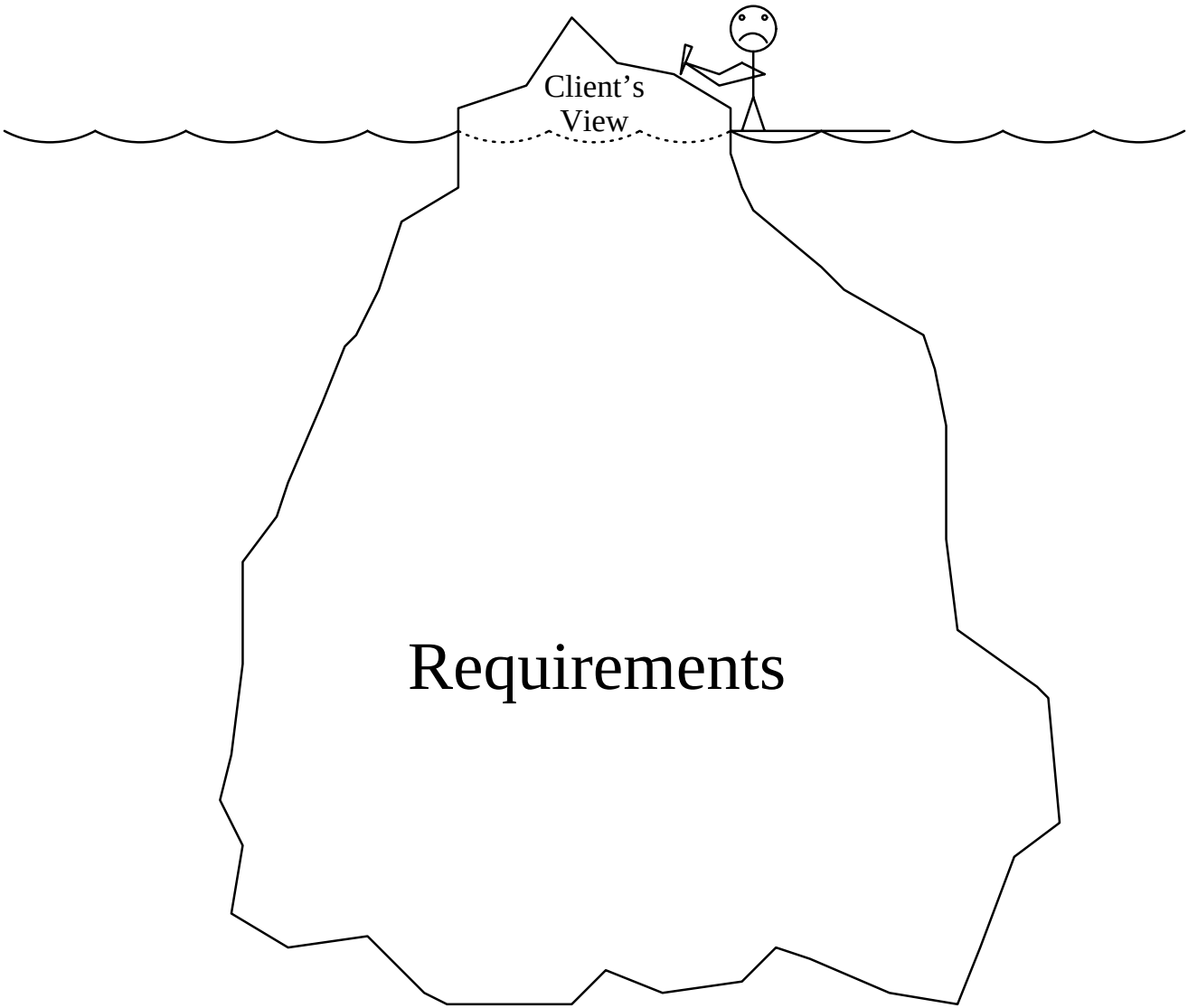
The Requirements Iceberg and Various Icepicks Chipping at It

Daniel M. Berry
dberry@uwaterloo.ca


Client's
View
Requirements

Outline

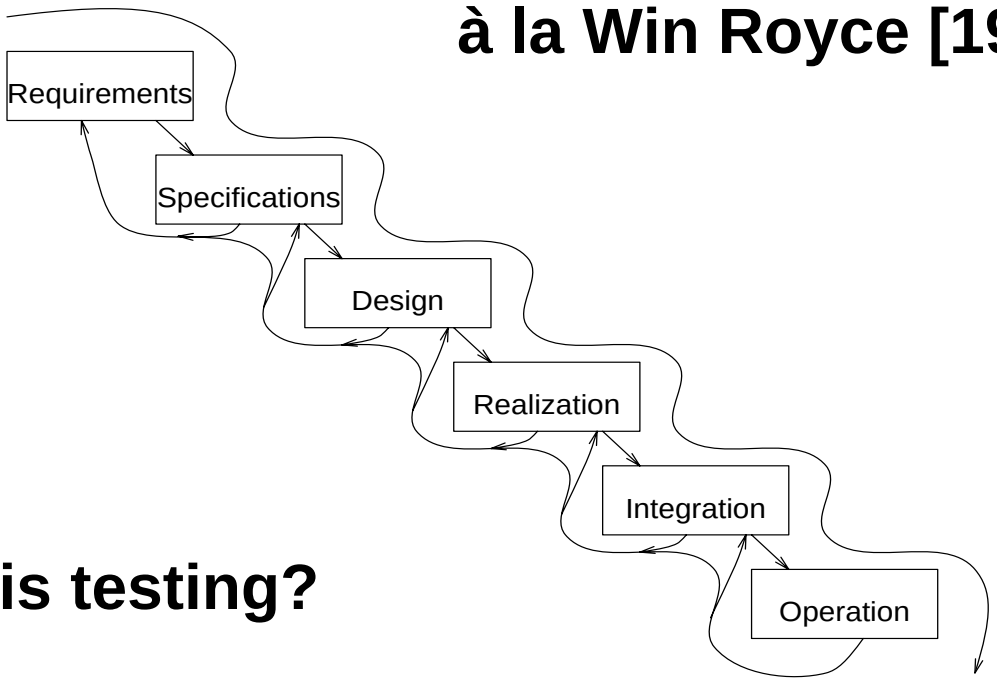
Lifecycle Models
RE is Hard
Why Important to Do RE Early
Myths and Realities
Where Do Requirements Come From?
Formal Methods Needed?
Requirements and Other Engineering
Bottom Line
RE Lifecycle

Outline, Cont'd

- **Overview of Research**
  - **Earlier and Later**
  - **Elicitation**
  - **Analysis**
  - **Natural Language Processing**
  - **Tools**
  - **Changes**
  - **Empirical Studies**
- **Future**

Traditional Waterfall Lifecycle

à la Win Royce [1970]



Where is testing?

Only one slight problem: It does not work!

Fred Brooks about Waterfall

In ICSE ’95 Keynote, Brooks [1995a] says “The Waterfall Model is Wrong!”

- **The hardest part of design is deciding *what* to design.**
- **Good design takes upstream jumping at every cascade, sometimes back more than one step.**

ICSE ’95 was in Seattle, Washington!

Fred Brooks also says:

“There’s no silver bullet!” [Brooks 1987]

- **Accidents**
  - **process**
  - **implementation**

  i.e., details

- **Essence**
  - **Requirements**

“No Silver Bullet” (NSB)

- **The *essence* of building software is devising the conceptual construct itself.**

- **This is very hard.**

  - **arbitrary complexity**
  - **conformity to given world**
  - **changes and changeability**
  - **invisibility**

NSB, Cont'd

- **Most productivity gain came from fixing *accidents***

  - **really awkward assembly language**
  - **severe time and space constraints**
  - **long batch turnaround time**
  - **clerical tasks for which tools are helpful**

NSB, Cont'd

- **However, the essence has resisted attack!**

  We have the same sense of being overwhelmed by the immensity of the programming problem and the seemingly endless details to take care of,

  and we produce the same kind of poorly designed software that makes the same kind of stupid mistakes

  as 40 years ago!

Brooks, Cont'd

Brooks adds, "The hardest single part of building a software system is deciding precisely what to build.... No other part of the work so cripples the resulting system if it is done wrong. No other part is more difficult to rectify later."

Real Life

We see similar requirement problems in real-life situations not at all related to software.

Contracts

We all know how hard it is to get a contract just right ...

to cover all possible unanticipated situations.

Houses

We all know how hard it is to get a house plan just right before starting to build the house.

Contractors even *plan* on this; they underbid on the basic plan, expecting to be able to overcharge on the inevitable changes the client thinks of later [Berry 1998].

Homework Assignments

We all know how hard it is to get the specification of a programming homework assignment right, especially when the instructor must invent new ones for every run of the course.

There is a continual stream of updates to the assignment.

Errors and Requirements

According to Barry Boehm [1981] and others, around 65–75% of all errors found in SW can be traced back to the requirements and design phases.

Errors and Requirements, Cont'd

Ken Jackson in a 2003 Tutorial on Requirements Management and Modeling with UML2, cites data from a year 2000 survey of 500 major projects' maintenance costs concluding that 70–85% of total project costs are rework due to requirements errors and new requirements.

In the table, the *d lines include requirements issues and add to 84%, but not all their instances are requirements related.

Errors and Requirements, Cont'd

Tom Gilb [1988] says that approximately 60% of all defects in software exist by design time.

Errors and Requirements, Cont'd

Marandi and Khan [2014] cite studies by Kumaresh & Baskaran and by Suma & Gopalakrishnan that show that the

- **requirement phase introduces 50%–60%,**
- **design phase introduces 15%–30%, and**
- **implementation phase introduces 10%–20%**

of total defects to software.

Flip Side

Those data say that we are doing a pretty good job of implementing of what we *think* we want.

But, we are doing a lousy job of knowing what we want.

Source of Errors

Either

- **the erroneous behavior is required because the situation causing the error was not understood or expressed correctly, or**

- **the erroneous behavior happens because the requirements simply do not mention the situation causing the error, and something not planned and not appropriate happens.**

Requirements Always Change

In a Requirements Engineering ’94 Keynote, Michael Jackson says:

Two things are known about requirements:

1. ***They will change!***
2. **They will be misunderstood!**

Why will they *always* change?

E-Type Software

à la Meir Lehman [Lehman 1980]

An E-type system solves a problem or implements an application in some *real-world* domain.

Once installed, an E-type system becomes inextricably part of the application domain, so that it ends up altering its own requirements.

E-Type Software, Cont'd

Example:

- **Consider a bank that exercises an *option* to automate its process and then discovers that it can handle more customers.**
- **It promotes and gets new customers, easily handled by the new system but beyond the capacity of the manual way.**
- **It cannot back out of automation.**
- **The requirements of the system have changed!**

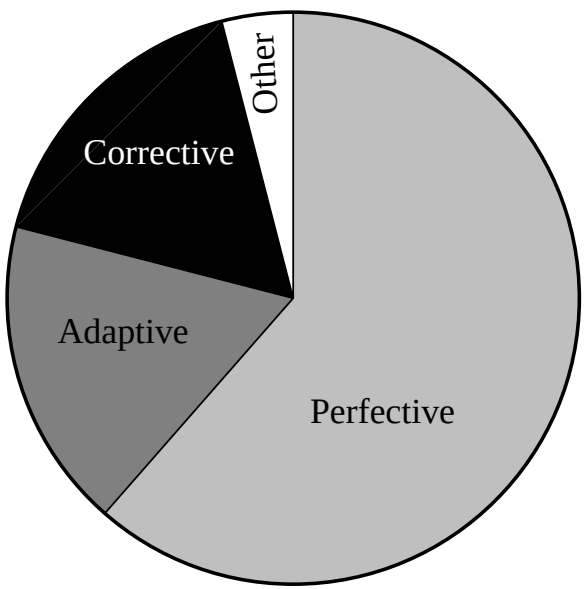
E-Type Software, Cont'd

Daily use of a system causes an irresistible ambition to improve it as users begin to suggest improvements.

Who is not familiar with that, from either end?

E-Type Software, Cont'd

In fact, data show that most maintenance is *not* corrective, but for dealing with E-type pressures!



Why Important to Do RE Early

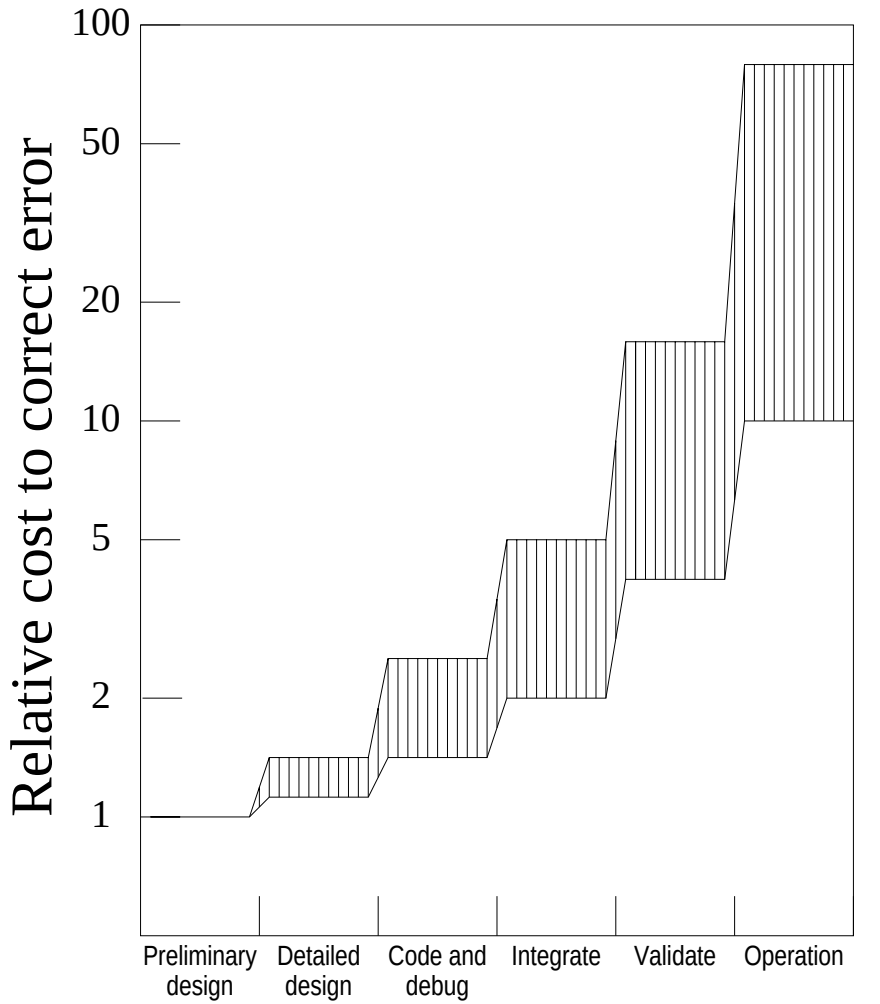
The BIG Question:

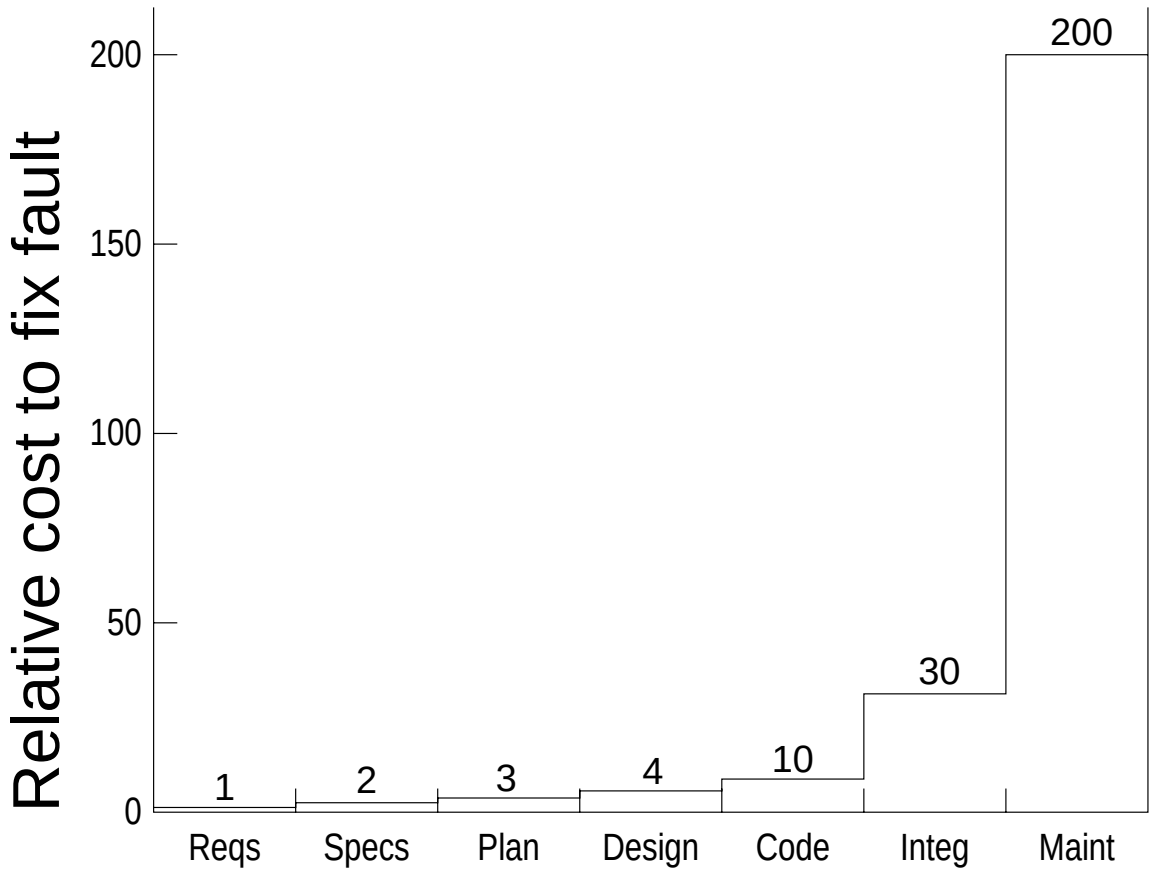
Why is it so important to get the requirements right early in the lifecycle? [Boehm 1981, Schach 1992]

We know that it is much cheaper to fix an error at requirements time than any time later in the lifecycle.

Cost to Fix Errors

Barry Boehm's (next slide) and Steve Schach's (slide after that) summaries of data over many application areas show that fixing an error after delivery costs two orders of magnitude more than fixing it it at RE time.

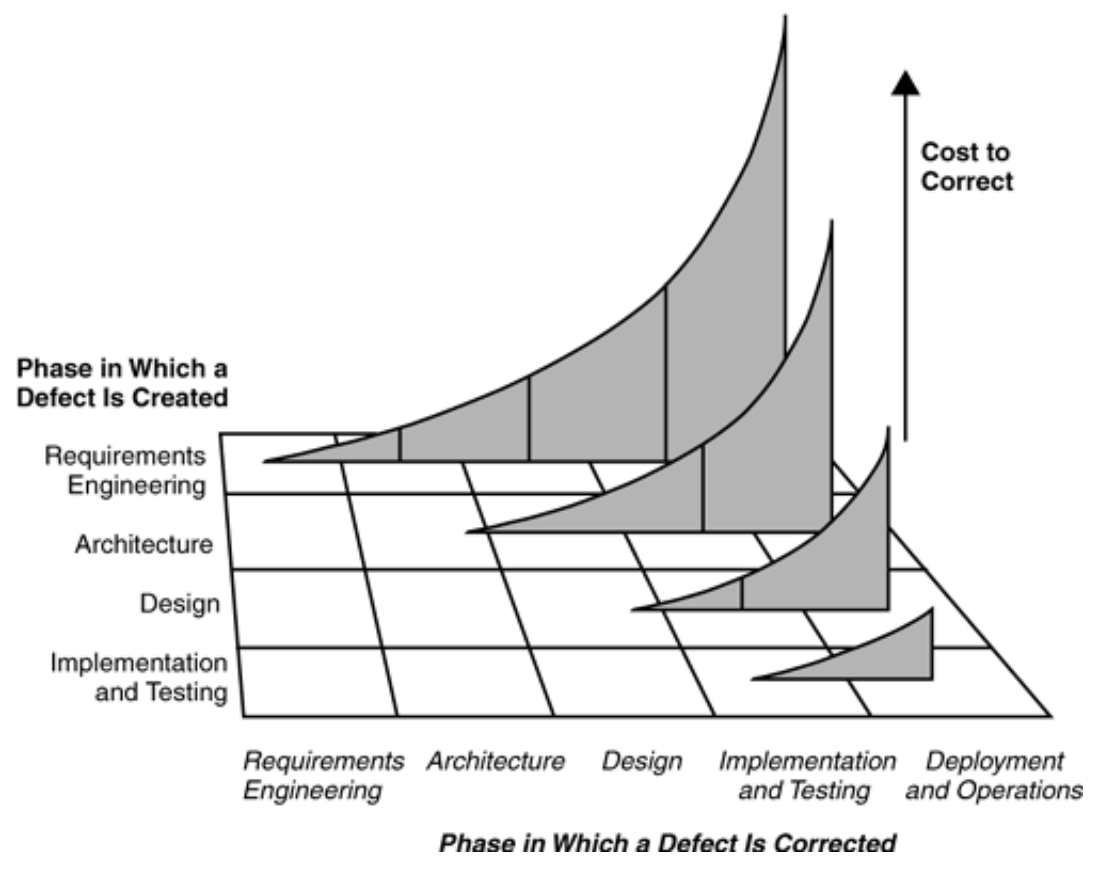

Relative cost to correct error
100
50
20
10
5
2
1
Preliminary design
Detailed design
Code and debug
Integrate
Validate
Operation
Phase in which error is detected


Relative cost to fix fault
200
150
100
50
0
1
2
3
4
10
30
200
Reqs
Specs
Plan
Design
Code
Integ
Maint
Phase in which fault is detected and fixed

Cost to Fix Errors, Cont'd

More specifically,

- **requirement defects are harder to fix than architectural defects,**
- **which are harder to fix than design defects,**
- **which are harder to fix than implementation defects [Allen et al 2008].**


Cost to Correct
Phase in Which a Defect Is Created
Requirements Engineering
Architecture
Design
Implementation and Testing
Requirements Engineering
Architecture
Design
Implementation and Testing
Deployment and Operations
Phase in Which a Defect Is Corrected

Conclusion

Therefore, it pays to find errors during RE.

Also, it pays to spend a *lot* of time getting the requirements specification error-free, to avoid later high-cost error repair, and to speed up implementation—even 70% of the lifecycle!

The 70% is not a prescription, but a prediction of what will happen, as we see later!

Reliability, Safety, Security, & Survivability

We know that we cannot program reliability, safety, security, and survivability into the code of a system only at implementation time. They must be required from the beginning so that consideration of their properties permeate the entire system development [Leveson 1995, Cheheyl *et al* 1981, Linger *et al* 1998].

The wrong requirements can preclude coding them at implementation time.

Prime Example: the Internet

Everybody is complaining about how insecure the Internet is [Neumann 1986]

Many are trying to add security to the Internet, and ultimately fail.

Why?

Internet Requirements

The original requirements for the ARPAnet, which later became the Internet, was that it be completely open.

Anyone sitting anywhere on the net was to be able to use any other site on the net as if he or she were logged in at the other site.

In other words, the ARPAnet was *required* to be open and essentially insecure [Cerf 2003, Leiner *et al* 2000].

Internet Requirements Were Met

And the implementers of the ARPAnet did a damn good job of implementing the requirements!

Adding security to the Internet ultimately fails because there is always a way around the add on security through the inherently open Internet.

Secure Internet from Reqs Up

To get a secure Internet, we have to rebuild the whole thing from requirements up, and there is no guarantee that it will look anything like what we have now and that the same applications would run on it.

Another View of History

The Internet was and *is* an E-type system, …

if there ever was one!

Oy!

Another View, Cont'd

The original Internet sites were only university and non-profit research labs.

No commercial, profit making organizations allowed.

Another View, Cont'd

The code implementing TCP/IP was a research prototype, …

which is fine because this code served as only a proof of concept, …

used by only cooperative, well-behaved users.

Another View, Cont'd

When the concept was proved, it was intended that some *commercial* institutions would build production quality versions of TCP/IP, …

each of which meets the full set of requirements needed to make it a reliable, robust, and secure system.

They would then market it, as with other research prototypes.

Another View, Cont'd

But E-type pressures proved to be too strong.

Some people started seeing the commercial potential of the service of the Internet and not of TCP/IP itself, which was viewed as a utility.

So the research prototype *became* the utility without ever going through the requirements analysis and development necessary to make it reliable, robust, and secure.

Sigh!

RE & Project Costs

The next slides show the benefits of spending a significant percentage of development costs on studying the requirements.

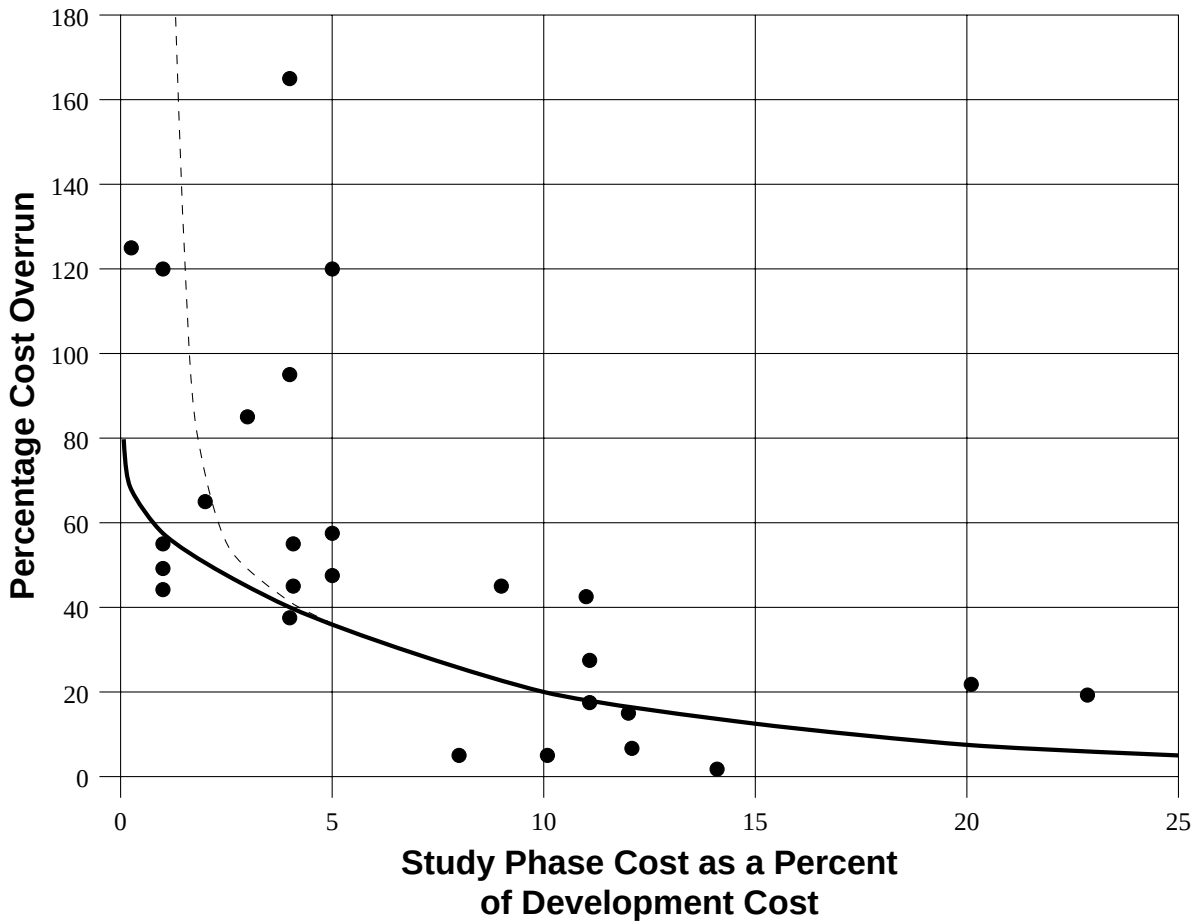
They contain a graph by Kevin Forsberg and Harold Mooz [1997] relating percentage cost overrun to study phase cost as a percentage of development cost in 25 NASA projects.

Project Costs, Cont'd

The study, performed by W. Gruhl at NASA HQ includes such projects as

- **Hubble Space Telescope**
- **TDRSS**
- **Gamma Ray Obs 1978**
- **Gamma Ray Obs 1982**
- **SeaSat**
- **Pioneer Venus**
- **Voyager**

Project Costs, Cont'd



Project Costs, Cont'd

There are three interpretations of the data:

The more you study the problem, ...

1. **the lower the costs,**
2. **the fewer the surprises that cause debugging and rework, and,**
3. **the more accurate the cost estimates are.**

It's probably a mixture of these.

A Case Study of Serious RE

A Master's student of mine, Lihua Ou, did a case study of writing requirements specification in the form of a user's manual [Berry *et al* (Ou) 2004].

It was *very* successful in that I got a piece of software that I wanted, it was implemented well, it does what I want it to do, and there is a well-written manual that describes the software's behavior completely.

A Case Study, Cont'd

Along the way, it ended up being also a case study in just having a serious requirements process, in which implementation did not begin, and was in fact *delayed*, until the requirements were completely worked out and specified satisfactorily.

The Software

The software was a WYSIWYG, direct manipulation picture drawing program, WD-PIC, based on the batch picture drawing language PIC, a TROFF preprocessor.

Lihua Ou's assignment was to produce a first production-quality version of WD-PIC as her master's thesis project.

Ou's Professional Background

Prior to coming to graduate school, Ou had built other systems in industrial jobs, mainly in commerce.

She had followed the traditional waterfall model, with its traditional heavy weight SRS.

She had made effective use of libraries to simplify development of applications.

Ou's Input

Ou was to look at all previous prototypes and UMs as specifications.

She was to filter these and scope them to first release of a production quality version of WD-PIC running on Sun UNIX systems.

Ou's Assignment

Ou was to write a specification of WD-PIC in the form of a UM.

This UM was

1. **to describe all features as desired by the customer, and**
2. **to be accepted as complete by the customer,**

before beginning design or implementation.

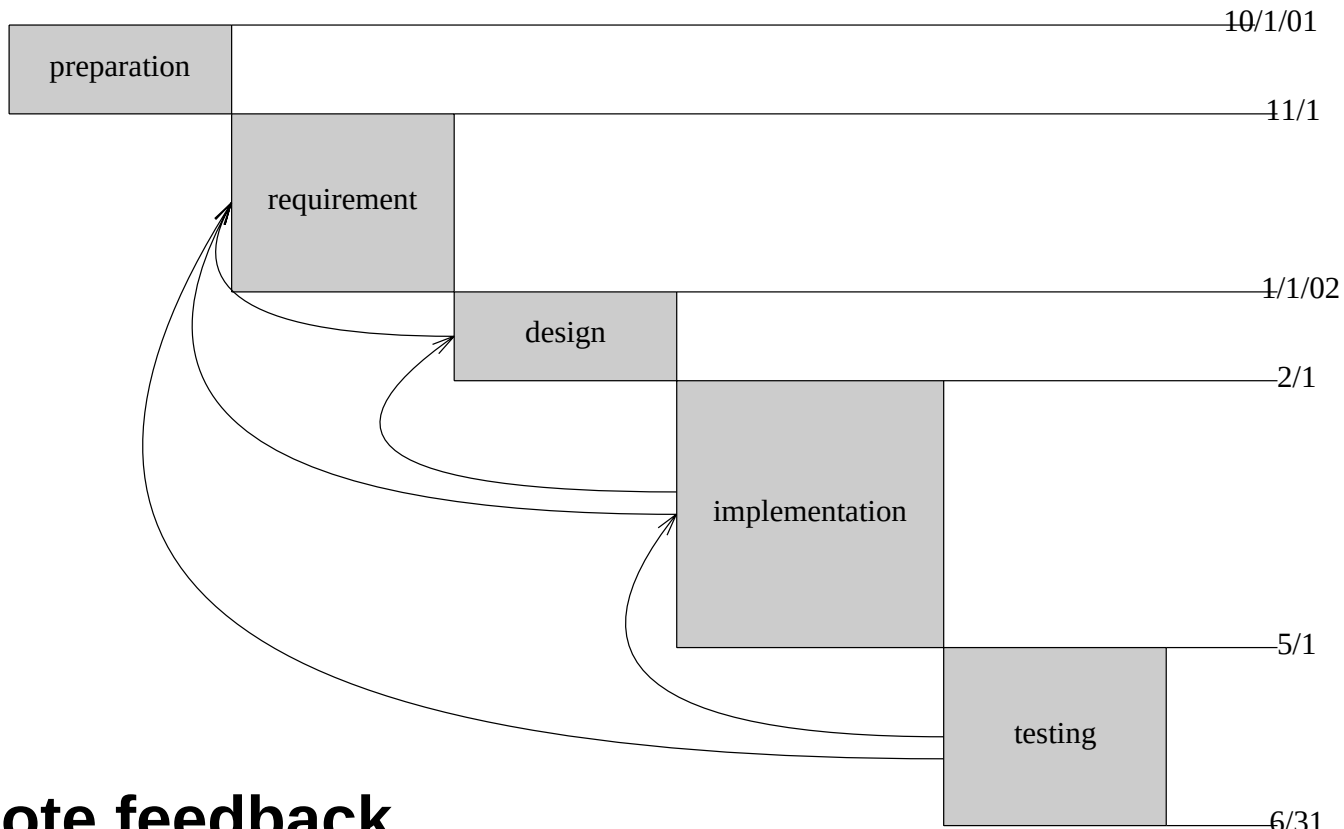
Ou's Assignment, Cont'd

Once implementation started, whenever new requirements were discovered, the UM had to be modified to capture new requirements.

In the end, the UM was to describe the program as delivered.

Project Plan

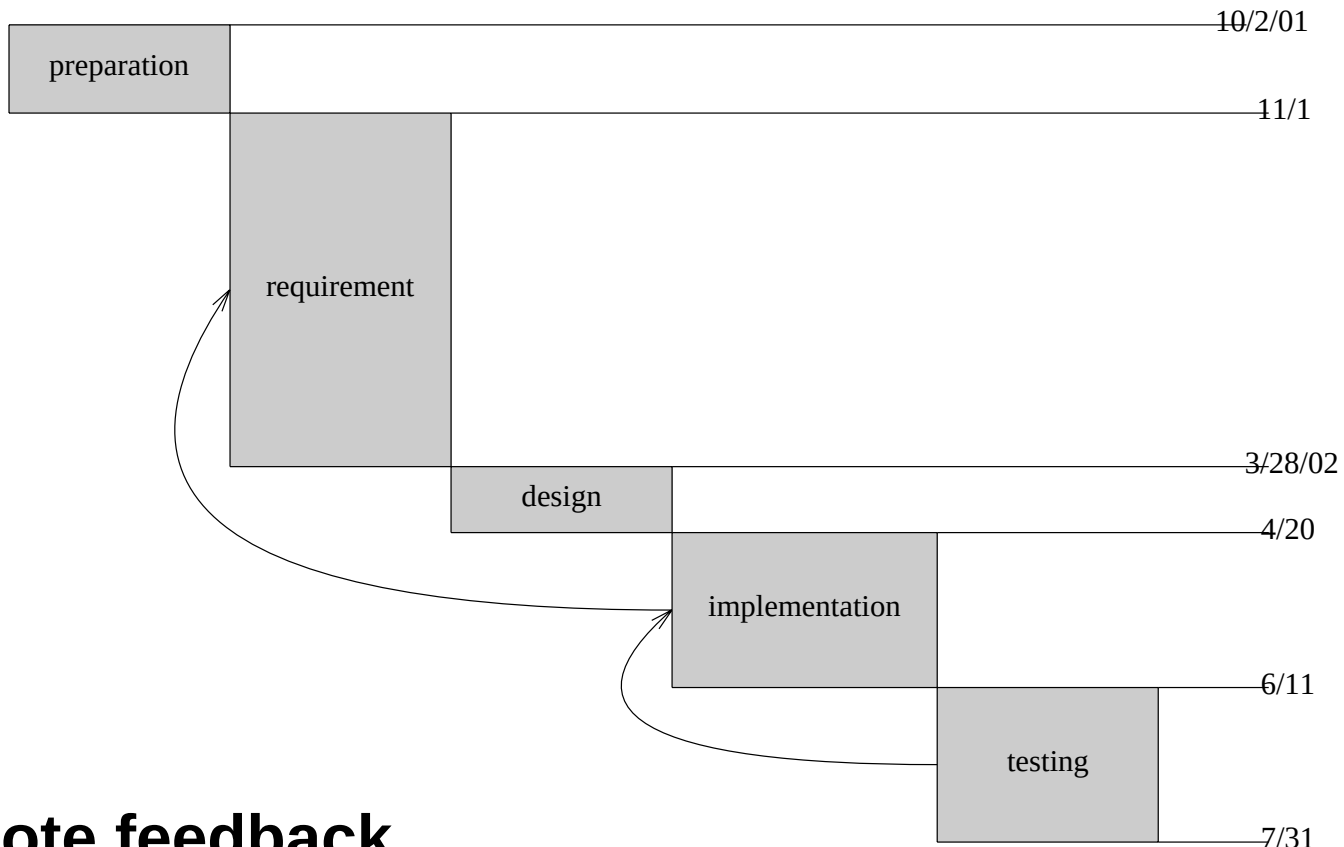
Duration in months	Step
1	Preparation
2	Requirements specification
4	Implementation
2	Testing
1	Buffer (probably more implementation and testing)
10	Total planned



Note feedback

Actual Schedule

Duration in months	Step
1	Preparation
4.9	Writing of user's manual = reqs spec, 11 versions
.7	Design including planning for maximum reuse of PIC code and JAVA library
1.7	Implementation including module testing and 3 manual revisions
1.7	Integration testing including 1 manual revision and implementation changes
10	Total actual



Note feedback

What Happened?

While detailed plan was not followed, total project time was as planned.

Also, Ou produced two implementations for the price of one, for:

- **(planned) Sun with UNIX and**
- **(unplanned) PC with Windows 2000**

Surprise

Ou was more surprised than Berry that she finished on time.

Berry had a lot of faith in the power of good RE to reduce implementation effort.

Adding to Ou's surprise was that the requirements phase took nearly 5 months instead of 2 months; the schedule had slipped 3 months out of 10, what appeared to be *way* beyond recovery.

Then and ...

Ou's long projected implementation and testing times and the 1 month buffer indicate that she expected implementation to be slowed by discovery of new requirements that necessitate major rewriting and restructuring.

Then and Now

This time, only minor rewriting and no restructuring.

Thus instead of 2 months specifying and 7 months implementing and testing,

she spent 5 months specifying and only 4 months implementing and testing.

Why?

By spending 3 additional months writing a specification that satisfied a particularly hard-nosed customer who insisted that the manual convince him that the product already existed,

Ou produced a specification that

- **had very few errors and**
- **that was very straightforwardly implemented.**

The Errors

Almost all errors found by testing were relatively minor, easy-to-fix implementation errors.

The two requirement errors were relatively low level and detailed.

They involved subfeatures in a way that required only very local changes to both the UM and the code.

What Helped?

All exceptional and variant cases had been worked out and described in the UM.

Thus, very little of the traditional

- **implementation-time fleshing out of exceptional and variant cases and**
- **implementation-time subconscious RE.**

Test Cases

The manual's scenarios, including exceptions and variants turned out to be a complete set of black box test cases.

Tests were so effective that, to our surprise, ...

scenarios not described in the UM, but which were logical extensions and combinations of those of the UM worked the first time!

The features composed orthogonally without a hitch!

Satisfied Customer

Berry found Ou's implementation to be production quality and is happily using it in his own work.

Another Case Study of Serious RE

This one involved what is called a lightweight formal method [Breen 2005].

At Philips Electronics, Michael Breen consulted for the project to develop CDR870 to become the first audio separate CD recorder aimed at the consumer market.

The CDR870 was to be the first of a product line.

Success or Failure

The key factor determining the conduct of the project was that it had to be finished in 6 months, in time for next Christmas.

Meeting this deadline and its implied schedule defines success or failure for the project.

The critical component of the project was the application-level software to be developed *from scratch*.

An Impossible Project?

The consensus among domain experts at Philips, people who had worked on similar systems in the past, was that it was impossible to finish in time. There were just too many unknowns.

Two people, including Breen, felt it would be possible *IF* ...

(There's ↑ the proverbial big "if".)

High Quality RS Essential

They realized that success depended critically on having a high quality requirements specification (RS) with *no* omissions, inconsistencies, and other defects, from which the code could be written straightforwardly.

High Quality RS Or Else

Any omission, inconsistency, or defect in the RS meant that the programmers would have to do RE on the fly, ...

leading inevitably to mistakes, backtracking, and other nasties, ...

leading in turn to delays, an unpredictable schedule, and flaky software, i.e., ...

to failure to deliver by Christmas.

Requirements for RS

The RS would have to be of only the user-visible behavior, to have a pure WHAT RS with complete freedom to choose the HOW based on the available technology ...

and the RS would have to be accompanied by a suite of automated regression tests ready to use at any stage to test the software's and the system's compliance with the RS.

Started RS Provisionally

They started writing the RS on a provisional basis.

They would proceed to implementation *only* if they had completed the RS in time *and* the RS met its quality requirements.

FSM-Based RS

Breen got the project to use multi-variable FSMs specified in tabular form.

The available natural language descriptions were translated into an initial tabular FSM specification.

Benefits of FSM-Based RS

This immediately exposed potential incompleteness in the form of unfilled table positions.

This immediately pinpointed inconsistencies in the form of multiple transitions from the same state.

Some potential incompletenesses proved to be DON’T CARES.

Benefits, Cont'd

Some potential inconsistencies proved to be the need for additional states.

The tabular FSMs gave the engineers the information they needed to rapidly resolve these problems.

The FSM model was built and checked manually.

Fortunately, the FSMs were not beyond the upper bound of what can be managed manually.

Result:

They finished the specification in time and it was judged by all involved to be of good, actually superb, quality.

Some felt it was the best RS that they had ever written. They had confidence that it was complete and consistent. They had confidence that it could be implemented straightforwardly with a minimum of delay and no backtracking.

To Implementation

They proceeded to implementation.

They finished the implementation in time with very few delays to flesh out requirements. The tests ran smoothly and served to expose the few implementation faults. No show stopping faults were found.

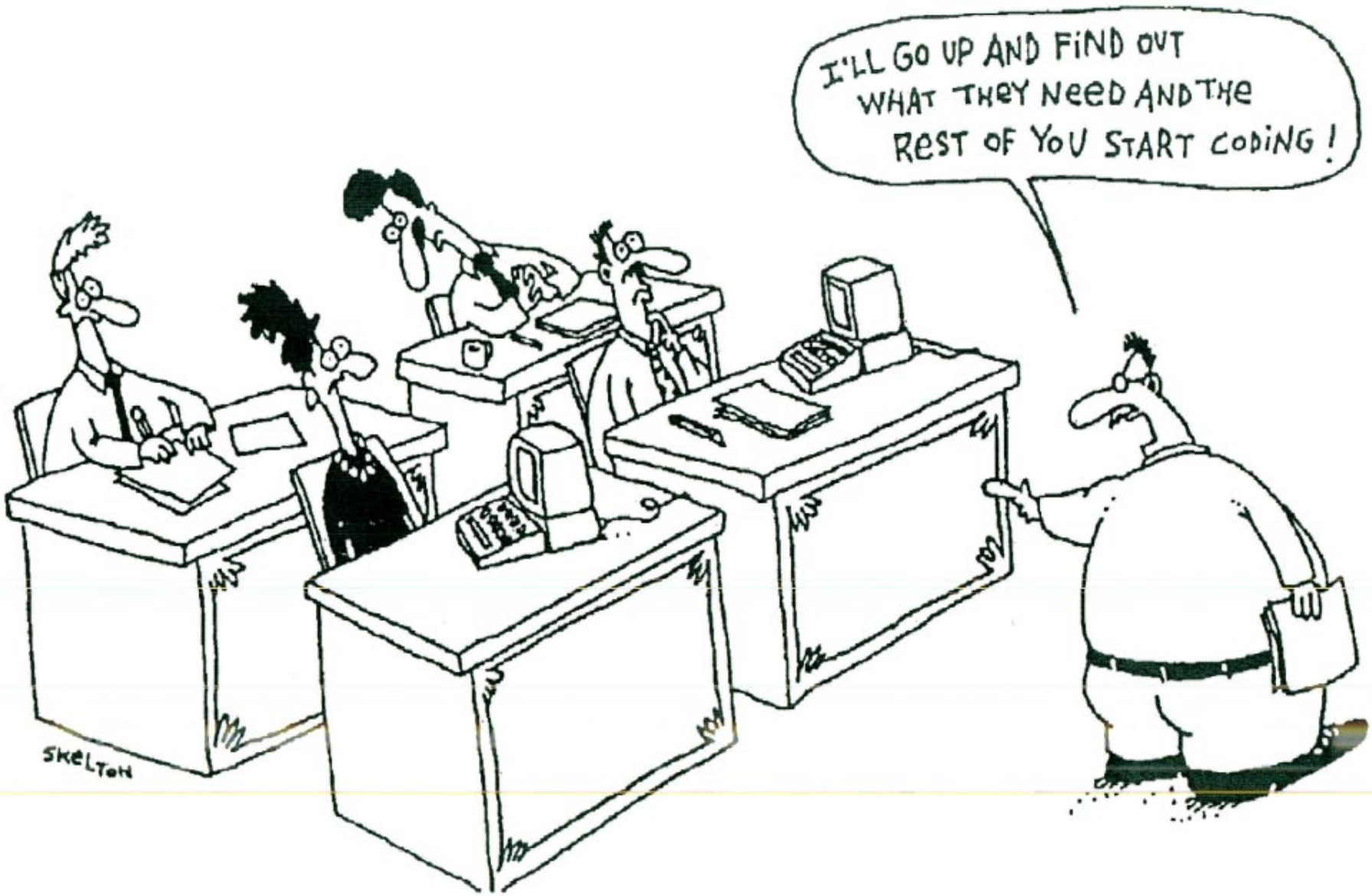
Success!

They delivered a high quality product in time for Christmas!

The tabular FSM specification approach continues to be used for subsequent members of the product line.

Myths and Realities

A bunch of myths about requirements and the answering realities


I'LL GO UP AND FIND OUT WHAT THEY NEED AND THE REST OF YOU START CODING!
SKELTON

Coding Before RE

Several related myths:

"You people start the coding while I go find out what the customer wants."

Requirements are easy to obtain.

The client/user knows what he/she wants.

Coding Before RE, Cont'd

"You people start the coding while I go find out what the customer wants."

Obeying this order amounts to a *very* bad bet!

It's practically guaranteed to end up at least doubling the cost of writing the code and developing the system.

Coding Before RE, Cont'd

If as little as 10% of the code written in advance of knowing the full requirements has to be changed after the full requirements are known, …

the cost of writing the code has doubled:

Bad Bet

If *C* is the cost of writing the advance version, the cost of fixing the advanced version when as little as 10% of it has to be changed,

then the total cost of writing the code is

$$C + (10 \times 0.1 \times C) = 2 \times C$$

Oy!

and it gets worse if more than 10% has to be changed.

Better Bet

So what's a better use of the programmers who would become idle if they are not put to work starting the coding while I go find out what the customer wants?

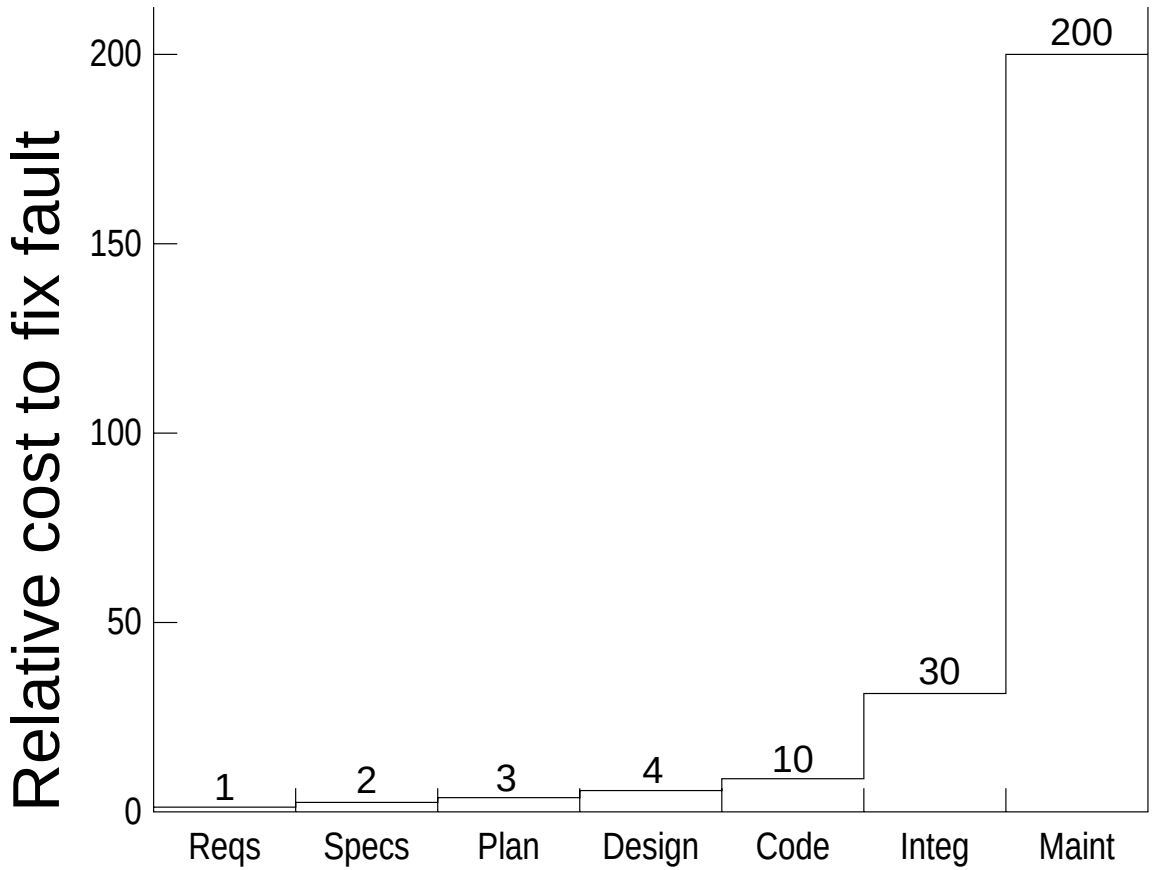
Better Bet

So what's a better use of the programmers?

Have them join the RE team

- **to provide more brain power to the RE effort and**
- **to help the RE team know when the requirements specification is complete enough that it can be programmed without the programmers' having to ask questions.**

Recall Cost to Fix Errors



Why Fixing Code So Costly

The BIG question:

“Why does it cost so much to fix code?”

It’s because updating code correctly is like lying perfectly consistently, which is very hard to do.

Why is lying so hard?

What is the lie?

Why Is Lying So Hard?

You've murdered someone, but

you don't want to take the rap for the crime.

Fortunately, no one actually saw you commit the murder.

Cover Up!

So, there's a chance that you can explain away those little facts

that would place you at the scene of the crime at the time of the crime, and

that would give you the motive to do the crime.

Concoct a Story

All you have to do is to to come up with a consistent story that fits

all the potentially incriminating facts

including that the victim is dead,

that *anyone* can see,

but that does not incriminate you, …

Concoct, Cont'd

when you have no way to be sure that you have identified

all such potentially incriminating facts

and

all such anyones.

Oy!!!!!

The Proof of the Lie

There will always be

that inconvenient witness that innocently reports

that inconvenient fact out of nowhere, that you did not know of

that proves that your story is a lie.

What is the Crime?

What is the crime with the software?

It has a big fat BUG!!!! Oy!!!!!!! Woe!!!!!!!

You gotta totally eradicate the bug, …

without throwing out the software and starting all over.

Actually, Why Not?

Because, too many people,

including those that can fire you if you are perceived as making a mistake,

think that *that* would be a crime, …

to waste all the good work that was done so far!

So you modify the existing code.

What Is The Lie?

The lie is making *all* parts of the modified code appear as if …

they were produced during …

an application of the current development method …

to produce the modified code from scratch, …

under the constraint that you cannot change the architecture of the code …

that has, and maybe even led to the BUG!

The Exposure of the Lie

The lie gets exposed because

there is always some overlooked piece of code

that is affected by the changes you made elsewhere in the code.

sigh! ☹

Where the Expense is

The expensive part of fixing defects in code is the attempt to find every last cockamamie piece of code that is

affected by the parts that you need to change and

affecting the parts that you need to change

recursively applied until

no new parts and

no new changes

are identified.

It's SO expensive …

It's *so* expensive, that fixing code costs at least 10 times what fixing the relevant requirements specification would cost.

Thus, if as little as 10% of the code has to be modified, it's cheaper to throw out the incorrect code and start all over than to fix the code.

But, no manager can bring him or herself to do that!

Empirical Evidence?

There is some empirical evidence to support this, …

but because so few people are willing to stick their necks out to try this, …

the reports are few and far between, not enough for generalization.

Evidence, Cont'd

Note that there are lots of project failure reports, …

many of which point to the difficulty of consistently updating code correctly.

What About Refactoring?

Refactoring, i.e.,

changing the architecture of the code,

without changing its behavior, and

reusing as much of the code as possible:

Isn’t that the BIGGEST lie ever?

Has to be done SO carefully

You have to do it piecemeal, one refactoring at a time,

moving as few code chunks as possible,

modifying as little code as possible,

and then testing,

before doing any other refactorings.

Why?

Indeed, why?"

Limiting the Scope of Bugs

to *try* to limit the scope of where the bug can be

when a refactoring does not preserve the behavior of the code.

This limiting does not always work! ☹ sigh!

This piecemeal work adds to the cost of the changes.

Compounding the Lie

Let’s see what happens when these sorts of in situ code fixes happen repeatedly so that the lies get compounded,

which often happens in a crime when the concocted story begins to unravel.

Recall Type E Systems

Meir Lehman classifies a CBS that solves a problem or implements an application in some real world domain as an E-type system.

Once installed, an E-type system becomes inextricably part of its application domain so that it ends up altering its own requirements.

So there is *no* hope of getting ahead of requirements.

Most Changes are Requirements Changes

Not all changes to a CBS are due to requirement changes.

But, as early as 1978, Bennett Lientz and Burton Swanson found that 80% of maintenance changes deal with requirements changes.

Decay of Software

As early as 1976, Laszlo Belady and Meir Lehman observed the phenomenon of eventual unbounded growth of errors in legacy programs that were continually modified in an attempt to fix errors and enhance functionality.

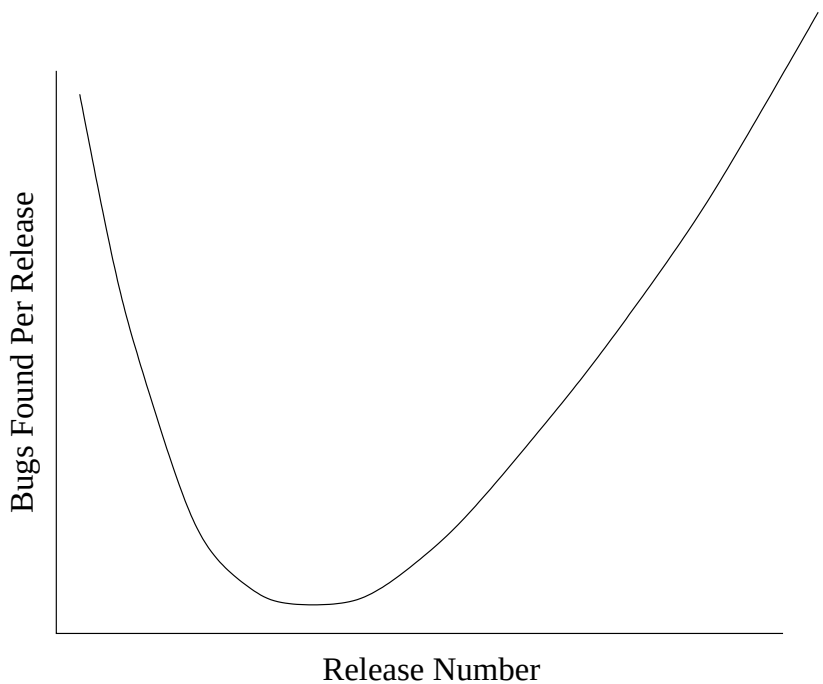
Decay of SW Cont'd

When a change is made, it's hard to find all places that are affected by the change.

So any change, including for correcting an error, has a non-zero chance to introduce a (new) error!

Belady-Lehman Graph

They modeled the phenomenon mathematically and derived a graph:



B-L Graph, Cont'd

In practice, the curve is not as smooth as in the figure; it's bumpy with local minima and maxima.

It is sometimes necessary to get very far into what we will call *Belady-Lehman (B-L) upswing* before being sure where the min point is.

Min Point

The min point represents the software at its most bug-free release.

After this point, the software's structure has so decayed that it is very difficult to change anything without adding more errors than have been fixed by the change.

Freezing SW

If we are in the B-L upswing for a CBS, we could roll back to the best version, at the min point.

Declare all bugs in this version to be features.

Usually not changing a CBS means that the CBS is dead; no one is demanding changes because no one is using the software any more.

Exceptions to Death

However, many old faithful, mature, and reliable programs have gone this way, e.g.:

- **cat, and other basic UNIX applications,**
- **vi, and**
- **ditroff**

Their user communities have grown to accept, and even, require that they never change.

Non-Freezable Programs

IF

- **the remaining bugs of best version are not acceptable features, or**
- **the lack of certain new features begins to kill usage of the CBS**

THEN a new CBS has to be developed from scratch

- **to meet all old and new requirements,**
- **to eliminate bugs, and**
- **to restore a good structure for future modifications.**

Another alternative

Use best version of legacy program as a feature server.

Build a nearly hollow client that

- **provides a new user interface,**
- **has the server do old features, and**
- **does only new features itself.**

Tendencies for B-L Upswing

The more complex the CBS is, the steeper the curve tends to be.

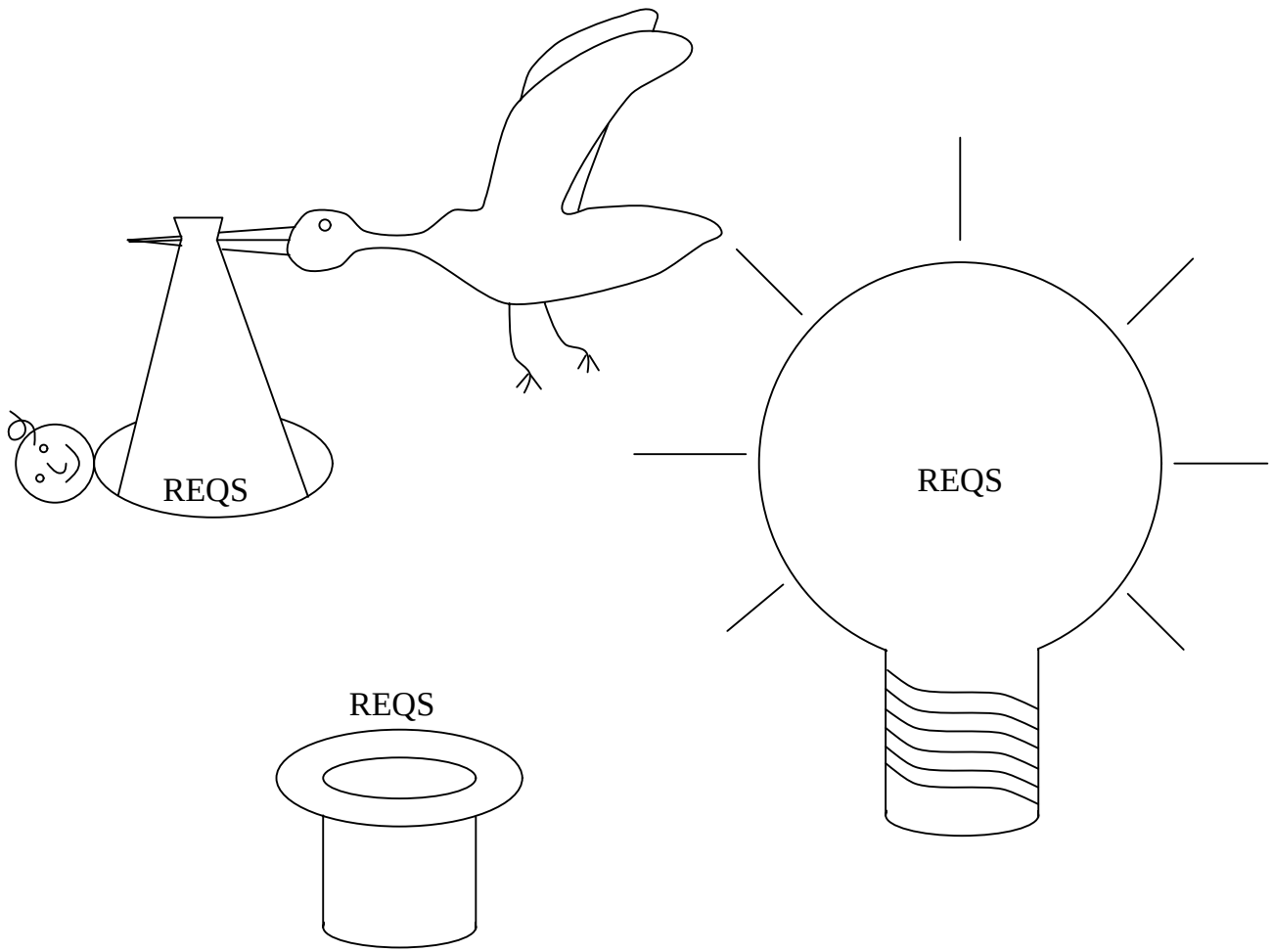
The more careful the development of the CBS is, the later the min point tends to be.

Occasionally

Occasionally, the min point is passed during the development of the first release, as a result of

- **extensive requirements creep that destroyed the initial architecture, or**
- **the code being slapped together into an extremely brittle CBS built with with no sense of structure at all.**

Whence Do Requirements Come?



Whence, Cont'd

Joe Goguen [1994a] says, "It is not quite accurate to say that requirements are in the minds of clients; it would be more accurate to say that they are in the social system of the client organization. They have to be invented, not captured or elicited, and that invention has to be a cooperative venture involving the client, the users, and the developers. The difficulties are mainly social, political, and cultural, and not technical."

Whence, Cont'd

Interviewing does not really help because when asked what they do, most people will quote the official policy, and not what they actually do. Most of what they really do, which is not specified by the policy, is what they do in situations not covered by the policy.

We're not even talking about conscious, politically safe mouthing of the policy.

Whence, Cont'd

Many people simply do not remember the exceptions unless and until they actually come up. Their conscious model of what happens *is* the policy.

Therefore, the requirements engineer has to *be* there when the exceptional situations come up in order to *see* what really happens.

Whence, Cont'd

Moreover, many people just do not know *why* they do something, saying only that it's done this way because the policy says so.

They very often do not even know why the policy is the way it is.

Whence, Cont'd

Moreover, many people just do not know *how* they do something, drawing a complete blank or saying only, "Watch me!".

For example, how do *you* ride a bicycle? Nu?

Whence, Cont'd

Don Gause and Jerry Weinberg [1989] tell the story of the woman who always cuts off ⅓ of a raw roast before cooking both pieces together.

She was asked "Why?" ... "???"

Her mother was asked "Why?" ... "???"

Her grandmother was asked "Why?" ...

Whence, Cont'd

Because the pot of this woman's grandmother was too small to accommodate the full length piece. Nu?

Whence, Cont'd

In other words, the policy once made sense, but the person who formulated the policy, the reasons for it, and the understanding of the reasons are long since gone.

Whence, Cont'd

For example, many companies that have committed all data to a highly reliable data base continue to print out the summary in quintuplicate.

Why? At the time of automation, the five most senior members of the company, who long ago retired, refused to learn to use the computer to access the data directly!

More on Whence

Recall that Joe Goguen [1994a] says, "It is not quite accurate to say that requirements are in the minds of clients; it would be more accurate to say that they are in the social system of the client organization. They have to be invented, not captured or elicited, and that invention has to be a cooperative venture involving the client, the users, and the developers. *The difficulties are mainly social, political, and cultural, and not technical*." [italics are mine]

Social, Political, & Cultural?

Several others have observed that emotions, values, beliefs, politics, and culture play a significant role in whether or not users accept and use deployed information-technology systems (ITSs).

Management tries to introduce ITSs to automate and transform business processes.

Employee View

However, many times, employees see these ITSs not as work savers or work facilitators, but fear them as job eliminators, job trivializers, and job complicators.

Such employees have difficulty using the ITSs, refuse to use them, or even sabotage them!

Sabotage

Isabel Ramos (Santos) *et al* [1998, 2002, & 2004] report several failed ITS deployments because of these fears and, in one case, user sabotage:

- **a mistake-logging system**
- **a new centralized system for a university library**
- **an OTS ERP system replacing a home-brewed system in a commercial company**
- **a CSCW system in a university classroom**

Political Reasons for Failed Projects

Johann Rost [2004] writes about political reasons for failed software projects.

He describes how subversive behavior can sabotage software projects.

LAS and CAPSA

The deployments of the London Ambulance System [Finkelstein 1993, Finkelstein & Dowell 1996] and the deployment of CAPSA, the Cambridge University's new on-line accounting system [Finkelstein & Shattock 2001] failed miserably.

Ramos believes that a prime cause of these failed deployments was the failure to deal with the stakeholders' emotions, values, and beliefs during their RE processes.

Technology vs. Politics

M.B. Bergman, J.L. King and K. Lyytinen [2002] observe, "Indeed, policymakers will tend to see all problems as political, while engineers will tend to see the same problems as technical. Those on the policy side cannot see the technical implications of unresolved political issues, and those on the technical side are unaware that the political ecology is creating serious problems that will show up in the functional ecology."

Technology vs. Politics, Cont'd

Bergman, King, and Lyytinen go on to say, "We believe that one source of opposition to explicit engagement of the political side of RE is the sense that politics is somehow in opposition to rationality. This is a misconception of the nature and role of politics. Political action embodies a vital form of rationality that is required to reach socially important decisions in conditions of incomplete information about the relationship between actions and outcomes."

User Acceptability

Boehm and Huang [2003] observe, project can be tremendously successful with respect to cost-oriented earned value, but an absolute disaster in terms of actual organizational value earned. This frequently happens when the product has flaws with respect to user acceptability, operational cost-effectiveness, or timely market entry."

Note that user acceptability is an emotional issue.

User Acceptability, Cont'd

Boehm and Huang add, "the initiative to implement a new order-entry system to reduce the time required to process must convince the sales people that using the new system features will be good for their careers. For example, if the order-entry system is so efficiency-optimized, it doesn't track sales credits [which prove who sold what], the sales people will fight using it."

Emotional RE

Ramos [Ramos (Santos) *et al* 1998, 2002, & 2004] suggests that the requirements engineer be on the lookout for signs of all sorts of social, emotional, political, and cultural problems among the customers and users during RE.

When such a problem is found, it should be explored with an eye to adjusting the requirements of the ITS so that the problem is ameliorated or even goes away.

Just Managerial Issues?

Some who see these ITS deployment problems regard the problems as managerial problems and not as requirements problems.

In one sense, they are right, in that these problems require action by management, addressing social issues.

What is a Requirements Problem?

However, any problem that can prevent the successful deployment of a system, whether it be

- **incorrect function,**
- **failure to notice tacit assumptions,**
- **or anything else**

should be identified as early as possible so that dealing with it can permeate the entire system design and development process.

Requirements Problems!

Perhaps a so-called managerial problem borne of emotion can be solved by a simple change in functionality or user interface, e.g., by eliminating a hated feature entirely.

Delaying consideration of any problem drives up the cost of solving the problem once it is identified as seen in graphs earlier.

When viewed this way, all such problems become requirement problems.

Managerial Solutions!

In the end, it may very well be that the adopted solution to an problem may be considered managerial, e.g., educating users and their managers, providing incentives for adopting, etc.

However, such solutions, especially that of educating users, may be applied also to what might appear to be a functional or user-interface issue (as did NASA [Lutz & Mikulski 2003]).