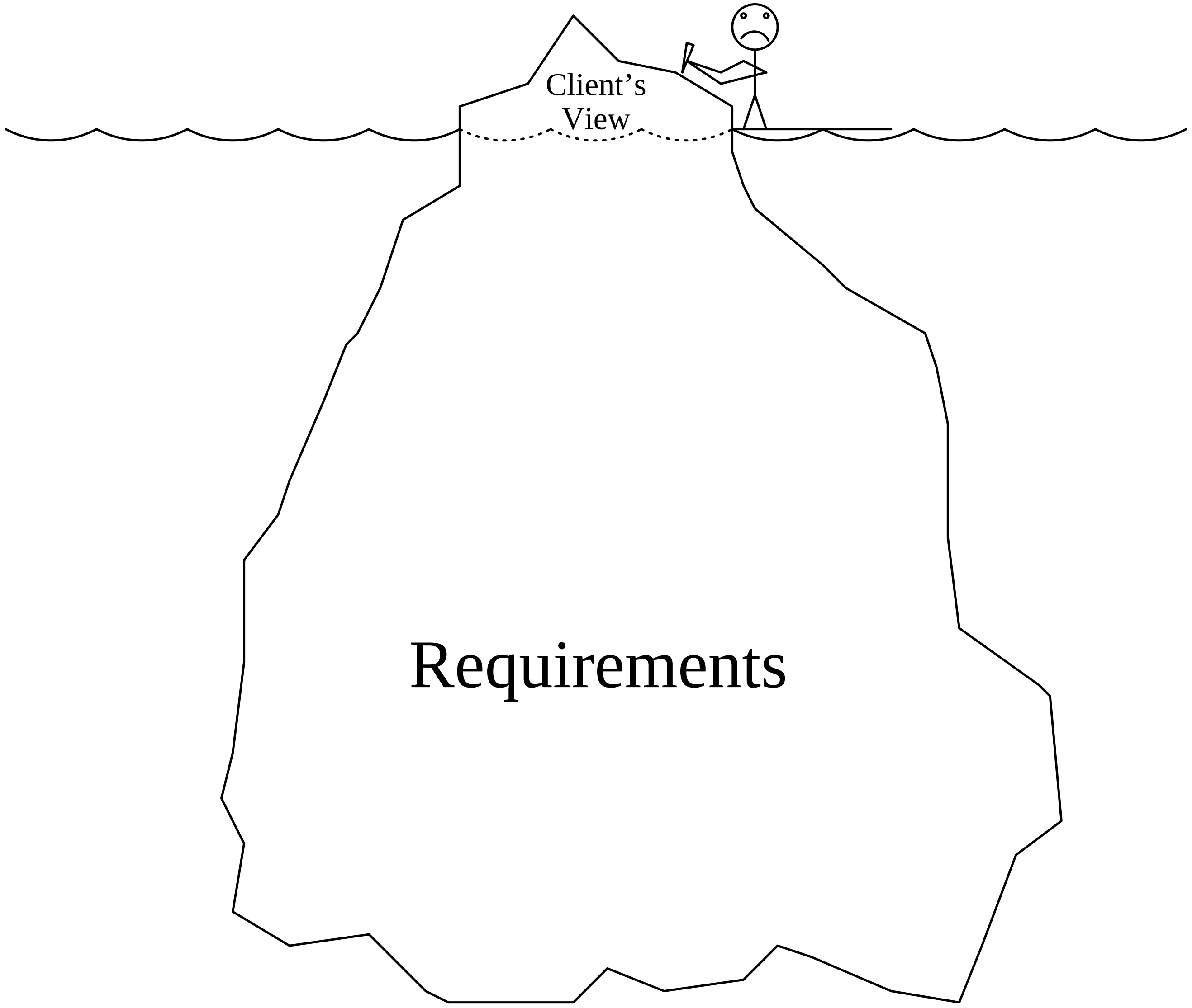
The Requirements Iceberg and Various Icepicks Chipping at It

Daniel M. Berry
dberry@uwaterloo.ca

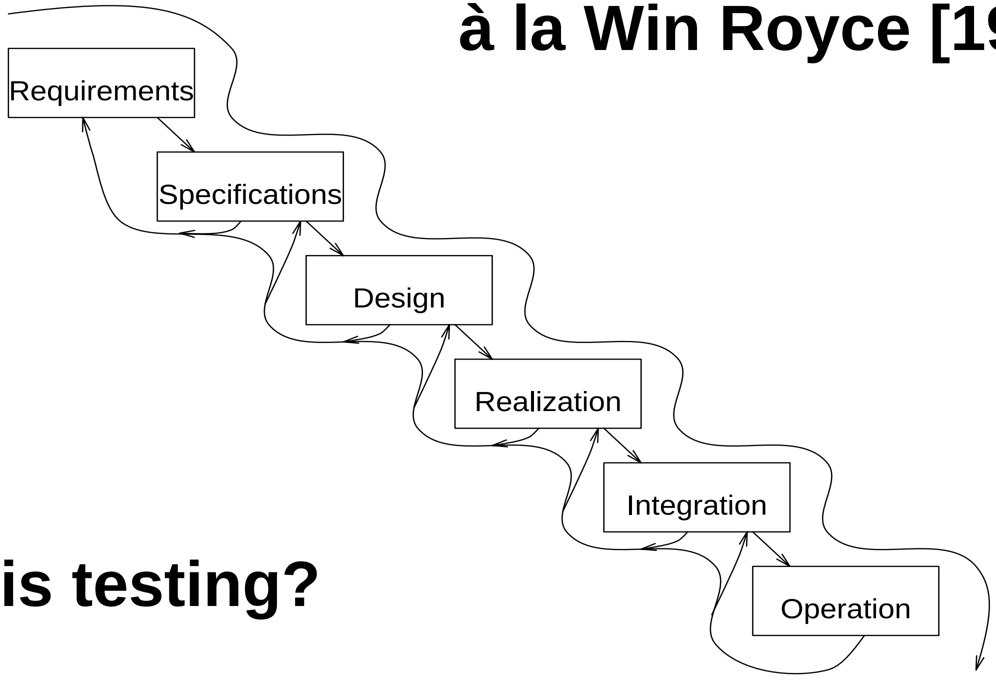

Client's View
Requirements

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

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.**

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.

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.

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.

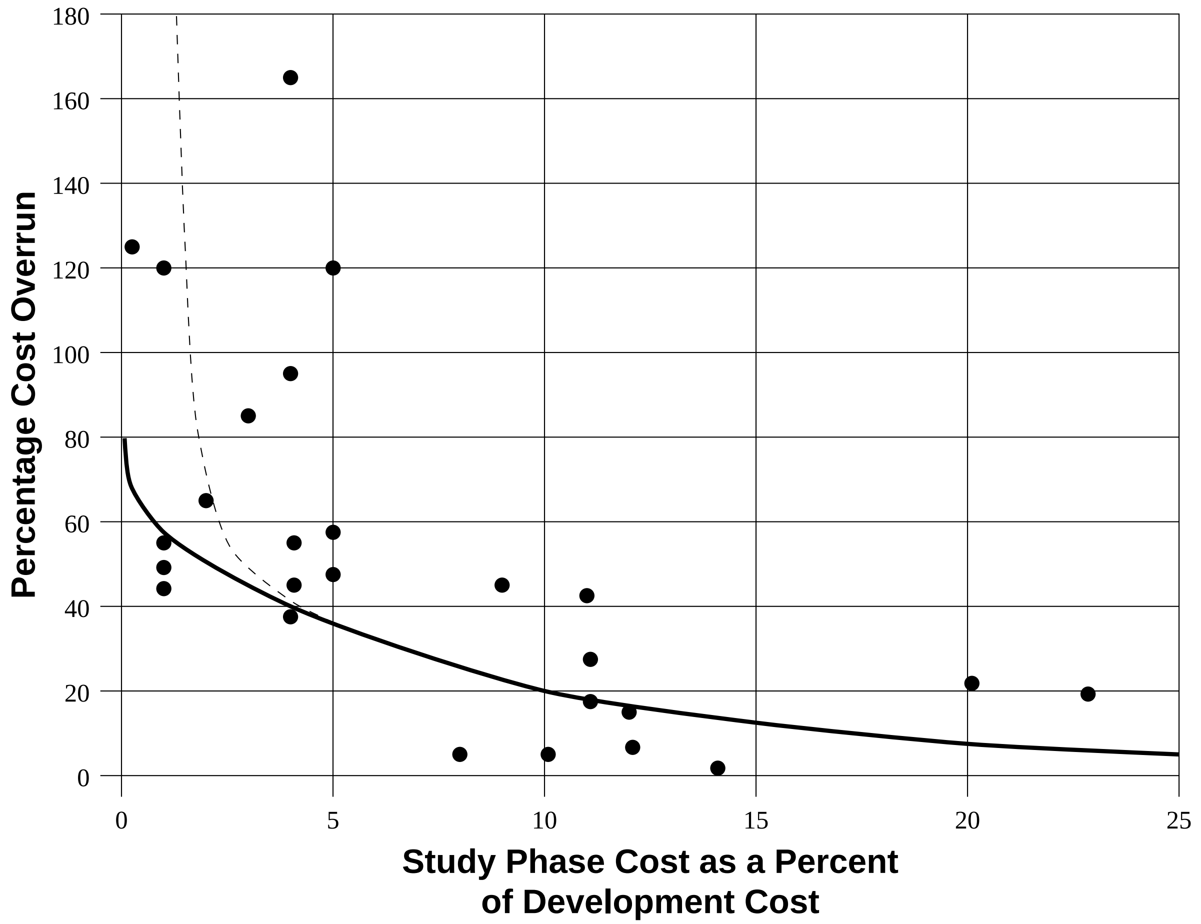
They contain a graph by Kevin Forsberg and Harold Mooz [1997] relating percentage cost overrun to study phase cost as a percentage of development cost in 25 NASA projects.

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.

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 emself 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, …

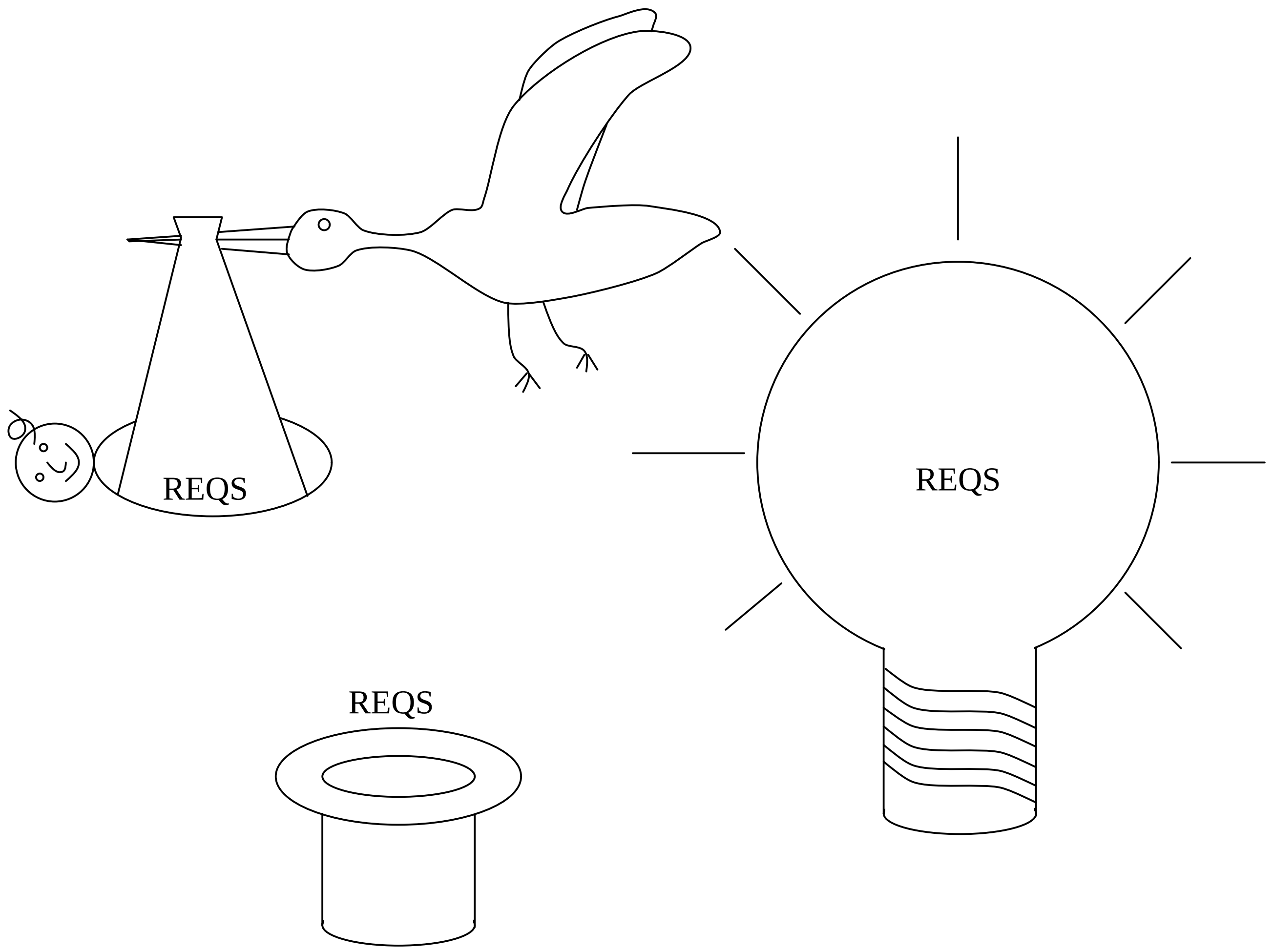
the reports are few and far between, not enough for generalization.

Evidence, Cont'd

Note that there are lots of project failure reports, …

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

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!