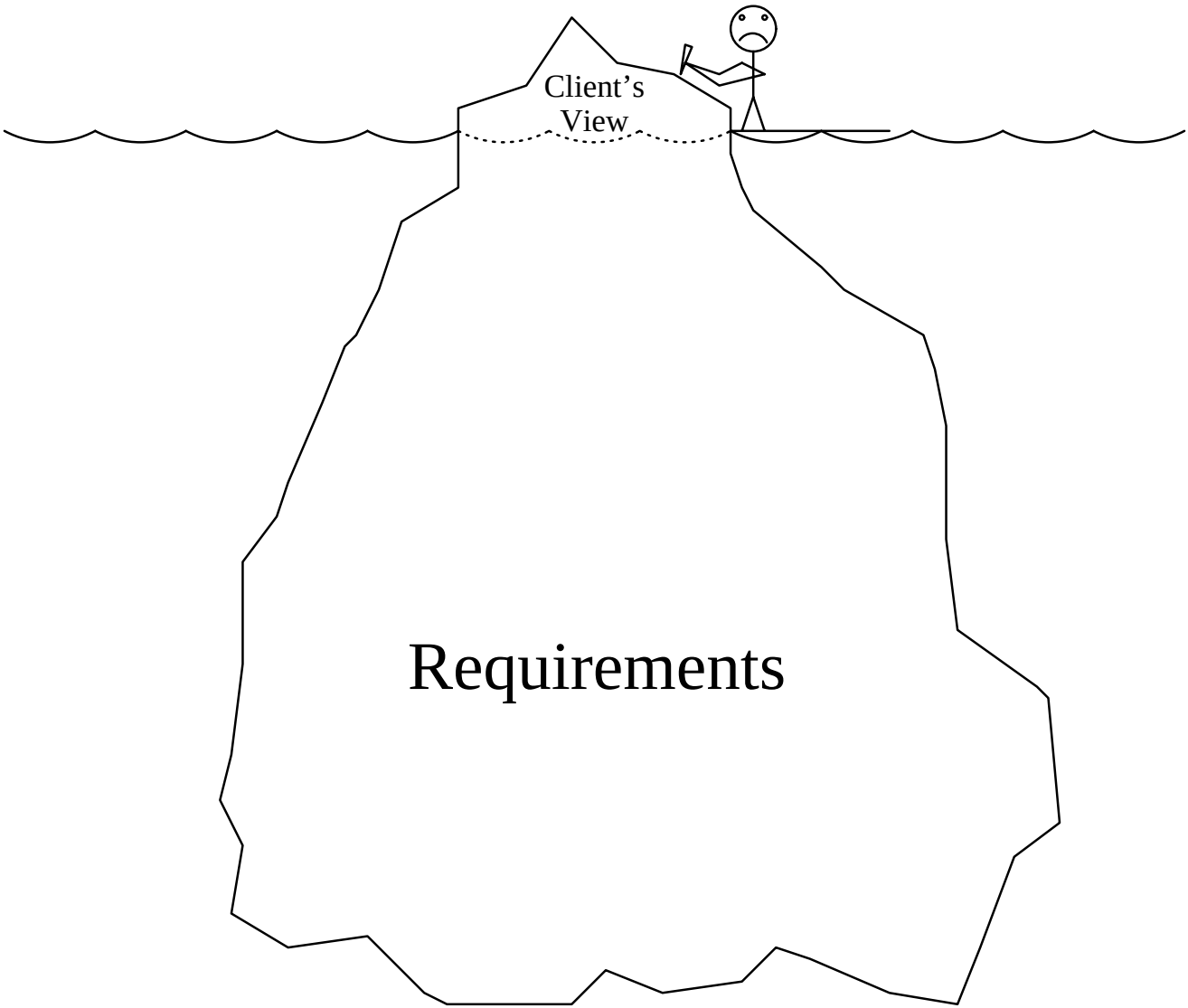
The Requirements Iceberg and Various Icepicks Chipping at It

Daniel M. Berry
dberry@uwaterloo.ca


Client's
View
Requirements

The Boring Title

Requirements Engineering (RE): the Problems and an Overview of Research

Outline

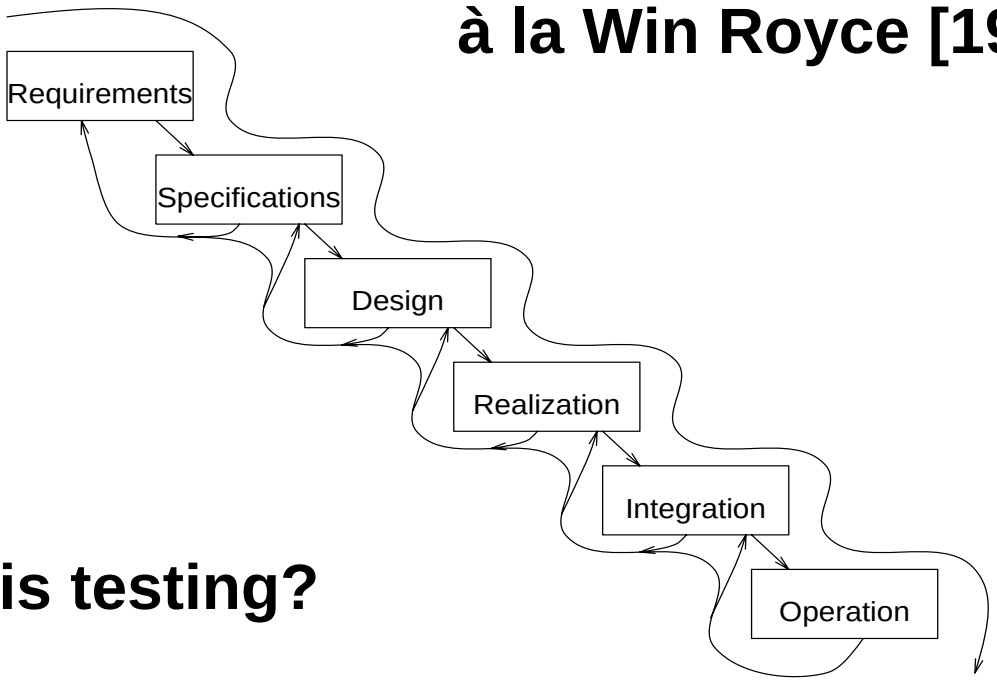
Lifecycle Models
RE is Hard
Why Important to Do RE Early
Myths and Realities
Where Do Requirements Come From?
Formal Methods Needed?
Requirements and Other Engineering
Bottom Line
RE Lifecycle

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!

Problems with Waterfall Model

The main problem, from the requirements point of view, of the waterfall model is the feeling it conveys of the sanctity and unchangeability of the requirements, as suggested by the following drawing by Barry Boehm [1988a].

Problems with Waterfall, Cont'd

This view does not work because requirements *always* change:

- **partially from requirements creep (but good project management helps)**
- **partially from mistakes (but prototyping and systematic methods help)**
- **partially because it is inherent in software that is used (the concept of E-type systems is discussed later!)**

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.

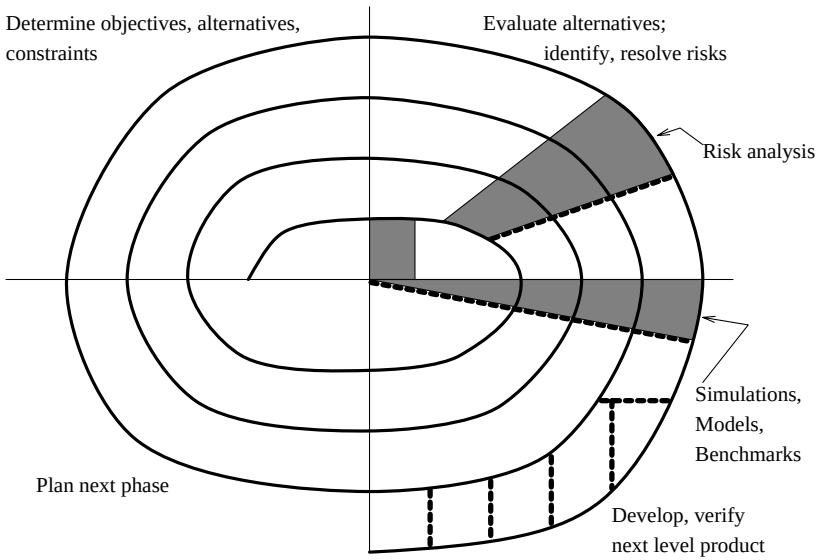
There is a continual stream of updates to the assignment.

SE Lifecycle and Reqs Changes

Thus, the SE lifecycle must be prepared to deal with ever-changing requirements.

More Realistic Lifecycle Model

Spiral Model à la Barry Boehm [1988b]



One may even follow the waterfall in each 360° sweep of the spiral.

Spiral Model

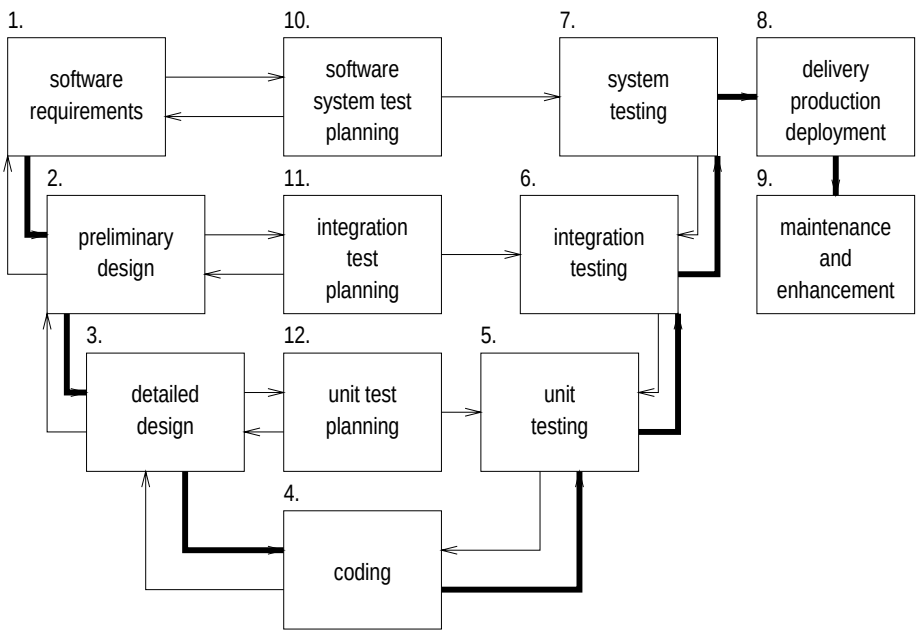
That is, the requirements and the implementation are developed incrementally.

That requirements are changing is planned.

But still, where is testing?

V Model

Some describe one such sweep not as a waterfall, but as a V



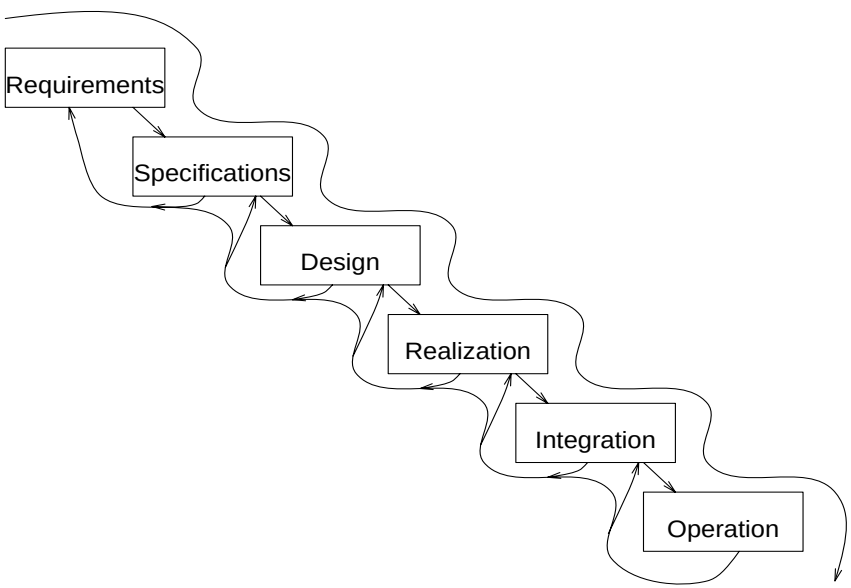
Planned Testing

The V model

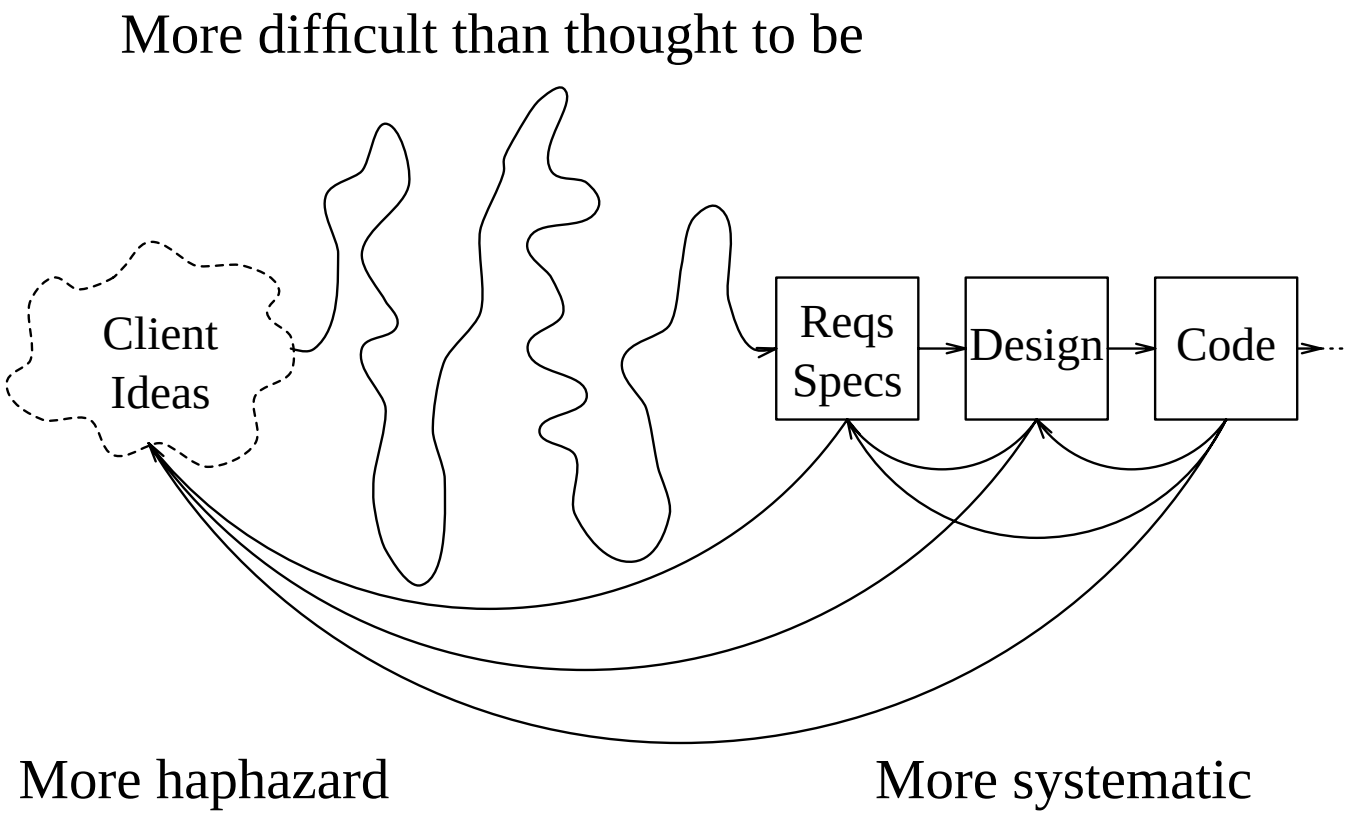
- **tries to indicate different volumes of information flow**
- **puts test planning and testing into its proper place, i.e., *everywhere*!**

More Unrealism in Waterfall

The Waterfall model imples also that all steps are equally systematic:



REAL Lifecycle for One Sweep



Requirements Engineering

That wavy line between Client Ideas and Requirements Specifications is RE.

IEEE Definition of Requirement

1. **a condition or capability needed by a user to solve a problem or achieve an objective**
2. **a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document**
3. **a documented representation of a condition or capability as in (1) or (2)**

[IEEE 1998]

Kinds of Requirements

- **functional—what the system should do**
- **non-functional—constraints on system or process, quality requirements**

Some non-functional requirements (NFR) are not specifiable, e.g., user friendliness.

It is often difficult to distinguish the two, e.g., a response time.

Loucopoulos's Definition of RE

Loucopoulos and Karakostas [1995]:

RE is a systematic process of

- **developing requirements through an iterative cooperative process of analyzing the problem,**
- **documenting the resulting observations in a variety of representation formats, and**
- **checking the accuracy of the understanding gained.**

Hsia's Definition of RE

RE is all activities which are related to

- **identifying and documenting customer and user needs,**
- **creating a document that describes the external behavior and associated constraints that will satisfy those needs,**
- **analyzing and validating the requirements document to insure consistency, completeness and feasibility, and**
- **evolution of these needs.**

Zave's Definition of RE

Pamela Zave [1997]:

RE is the branch of SE concerned with real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families.

Zave's Definition of RE, Cont'd

Important aspects of definition [Nuseibeh & Easterbrook 2000]:

- **real-world goals**
  - **what**
  - **why**
- **precise specifications, basis for**
  - **analyzing requirements**
  - **validating with stakeholders**
  - **defining what is to be built**
  - **verifying implementation**

Zave's Definition of RE, Cont'd

- **evolution**
  - **over time for a changing world**
  - **across families with partial reuse**

Systems, Not Just Software

Bashar Nuseibeh & Steve Easterbrook [2000]:

RE has been called a branch of Software Engineering.

In reality, software cannot function in isolation from the system in which it is embedded.

Prefer to characterize RE as a branch of Systems Engineering.

Ryan's Definition of RE

RE is the development and use of cost-effective technology for the elicitation, specification and analysis of the stakeholder requirements which are to be met by software intensive systems.

Stakeholders

A stakeholder is a person who is directly or indirectly affected by the system under construction, .e.g.,

- **customers and users**
- **marketing and sales personnel**
- **developers and testers**
- **maintainers**
- **managers**

But ‘X Engineering’ is Illegal

“Engineering” is a controlled term, not legal to use with “Requirements”.

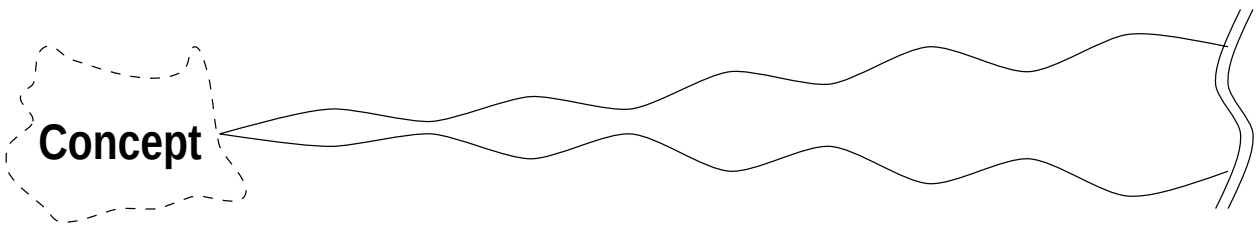
The term “RE” is used as a reminder that the gathering of requirements occurs as part of an *engineering* process.

RE is Hard

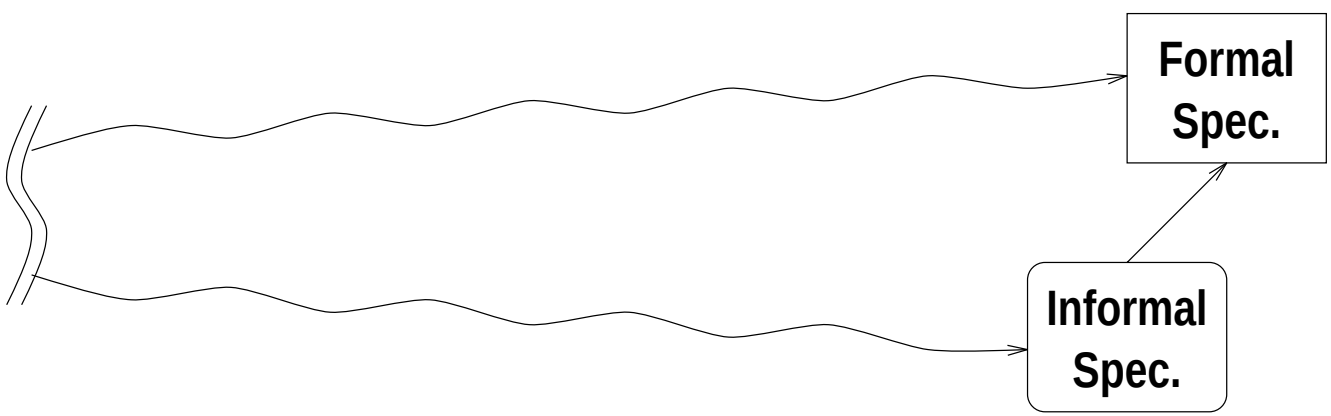
How hard?

Like fighting a platoon of angry, slightly inebriated Klingons.

Distance: Concept → Specs

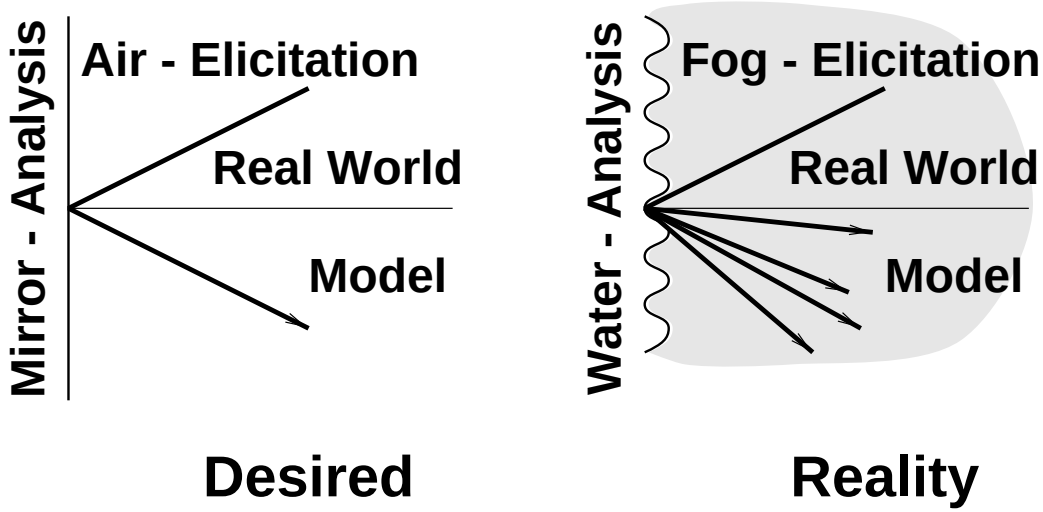


Folded in middle to give feeling of true conceptual distances involved



Modeling Difficulties

Brian Mathews has another view in terms of modeling



What vs. How

A requirements specification is supposed to describe what the system should do and not how.

It is not easy to obey this rule in practice.

What vs. How, Cont'd

One person's how is another person's what; to an author, a WORD data structure is a how issue, but to a system programmer, a data structure in UNIX is a what issue.

In a model of multiple levels of abstraction, each level is both the how of the level above and the what of the level below. [Ryan 1998]

Three Kinds of Requirements

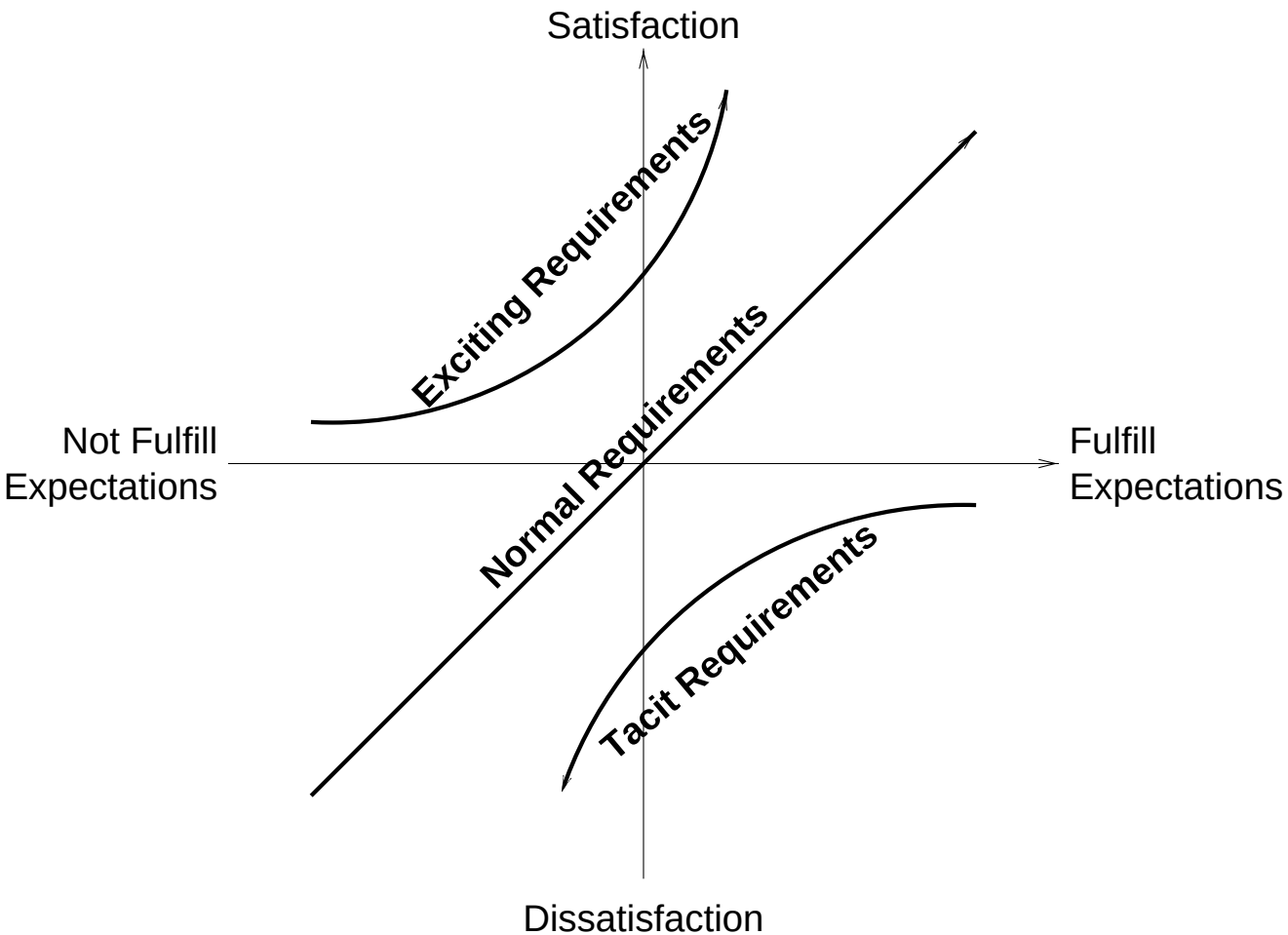
Noriaki Kano [1984] identified 3 kinds of system requirements.

1. ***Normal*—Users mention these with no problems.**
   They contribute proportionally to user satisfaction of the system.
2. ***Exciting*—Creative developers invent these.**
   They contribute dramatically to user satisfaction of the system.

Three Kinds, Cont'd

3. ***Tacit*—Users understand these, but do not mention them; developers do not see these.**
   Unfound by developers, they contribute dramatically to user dissatisfaction of the system.

Three Kinds, Cont'd



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

Cause of Rework	%-age of Total Project Costs
*Changes in user requirements	43%
*Changes in data formats	17%
*Emergency fixes	12%
*Hardware changes	6%
*Documentation	6%
Efficiency improvements	6%
Other	9%

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.**

Worth Fixing an Error?

Sometimes it's not worth fixing a requirements error in software that works pretty well. That is, it's not worth modifying the software to meet a newly discovered requirement.

E.g., at U of Waterloo, a system called QUEST has automated course scheduling and registration so that a student can register on line via the WWW.

Worth Fixing an Error, Cont'd?

For each course *C* for which a student *S* tries to register, QUEST checks that *S* has successfully taken all the prerequisites for *C*.

However, there are degree programs, e.g., ConGESE, that use regular courses, but in a non-normal sequence. In this program, because everyone is already a practicing software engineer, no course has any prerequisites.

Worth Fixing an Error? Cont'd

Thus when a ConGESE student tries to register for the RE course I teach at UW, he or she is usually not allowed to register because he or she has not taken any of the prerequisite courses.

This situation was never considered in developing QUEST.

Worth Fixing an Error? Cont'd

Fortunately, QUEST has provided for superusers that can force QUEST to accept registrations that would otherwise not be allowed. So the students were able to be registered.

Worth Fixing an Error? Cont'd

Is it worth changing QUEST?

Decidely, "No!".

The frequency of the ConGESE situation is once every year, and the number of students involved is between 10 and 20 each time.

It's easy enough for a superuser to handle the situation manually.

Worth Fixing an Error? Cont'd

Modifying the software risks introducing bugs.

This change would be rather complex because it would have to introduce a notion of independent streams with different prerequisite structures and force prerequisites to be associated not with just a course, but a course and a program together. Implementing this means changing the whole course abstraction. Yecch!

Worth Fixing an Error? Cont'd

So this bug is declared as a feature, and the superuser intervention becomes the official way to deal with ConGESE course registrations.

The same solution will be applied to any other degree program with similar requirements....

Until such time as there are so many other programs that superuser intervention becomes burdensome.

Even NASA Doesn't Always Fix

Robyn Lutz and Inés Carmen Mikulski [2003] discuss how NASA sometimes discovers requirements errors and new requirements during system operations.

If a mission is already in progress, sometimes the response to this discovery is to change the procedures for the human operators at mission control rather than to try to modify the embedded system on board a space craft.

Paradox

Fred Brooks [1995b] observed that a general purpose system is harder to design than a special purpose product.

Study of Requirement Errors

by Martin & Tsai [1988]

Experiment to identify lifecycle stages in which requirement errors are found

Polished 10-page requirements for centralized railroad traffic controller, ...

written by the user, a professional railroad traffic controller who had become a programmer—BOBW!

Experiment

Ten 4-person teams of software engineers, pretending to prepare for implementation

User believed that teams would find only 1 or 2 errors

Study, Cont'd

92 errors, some very serious, were found!

Average team found only 35.5 errors, i.e., it missed 56.5 to be found downstream!

Many errors found by *only* one team!

Errors of greatest severity found by fewest teams!

Michael Jackson Says

In a Requirements Engineering ’94 Keynote, Jackson [1994] says:

Two things are known about requirements:

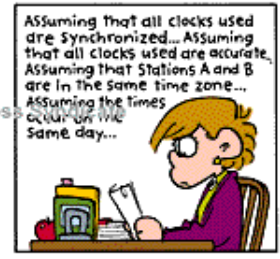
1. **They will change!**
2. ***They will be misunderstood!***

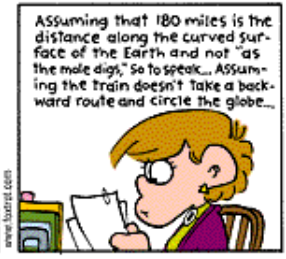
Tacit Assumption Tarpit

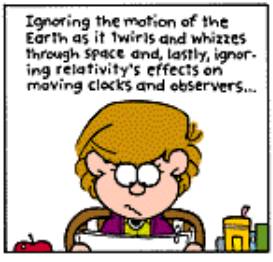
A “Foxtrot” cartoon in which Jason’s homework solution lists *all* the tacit assumptions, and I mean *all*, that allow him to deduce that the average speed of a 180 mile train trip from 10am until 2pm is 45mph.

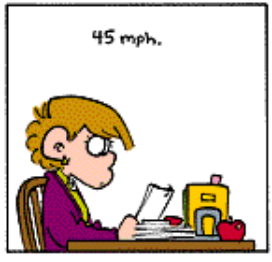
FoxTrot

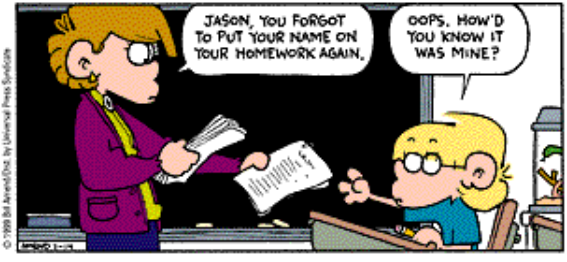
by Bill Amend











Tacit Assumptions, Cont'd

Those tacit assumptions of the problem are reasonable, right?

Unfortunately, the source of most disasters, such as at nuclear power plants, is perfectly reasonable, possibly explicit but usually tacit, assumptions that did not hold in some special circumstances that nobody thought about.

Timely Tacit Assumptions

According to the UW schedule, a TA had a laboratory scheduled for “8:00” (with no “am” or “pm”). He and some students assumed that it was at 8:00am. Most students assumed that it was at 8:00pm. The university meant “8:00pm”.

Perhaps, those who assumed it was “8:00am” could have figured that since classes start at 8:30am, “8:00” had to be “8:00pm”, but that’s stretching it.

Timely Tacit Assumptions, Cont'd

The university schedule should have made it clear that it was at 8:00pm.

The reason it did not say "8:00pm" is that there is no "pm" designation because in most cases, from from 12:00 until 5:00, it is obvious that "pm" is meant. Who goes to class at 4:00am?

More Timely Tacit Assumptions

Once in Tel Aviv, I read a Hebrew no-parking sign saying "8:00–17:00" as saying that there is no parking from 5:00pm until 8:00am the next morning to reserve parking places for the people who live on the street, who would have special exemption certificates.

While numerals are read from left to right, the flow of the sentence is from right to left and the "–" is *not* part of each numeral.

Timely Tacit Assumptions, Cont'd

According to the city of Tel Aviv, the sign means that there is no parking from 8:00am until 5:00pm to keep the street traversable during the business day.

According to Hebrew reading rules, I am correct and the city is wrong.

I could not get the city to see their error and cancel the ticket!

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!**

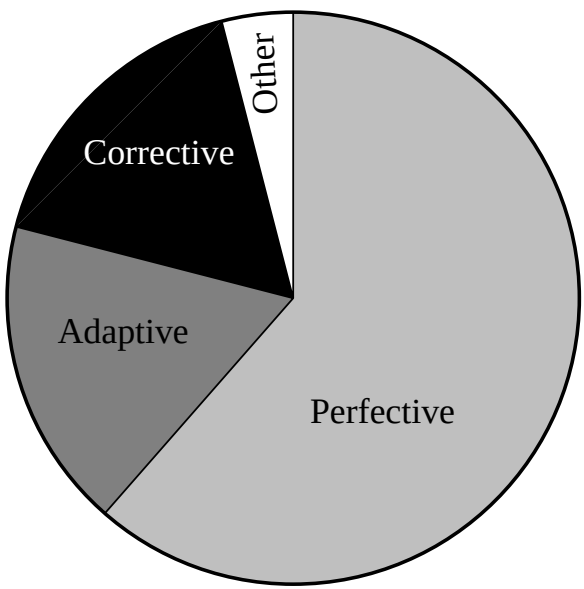
E-Type Software, Cont'd

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!



RE is Hard

Despite the clear benefits of getting requirements complete, right, and error-free early, they are the hardest part of the system development lifecycle to do right because:

- **we don’t always understand everything about the real world that we need to know,**
- **we may understand a lot, but we cannot express everything that we know,**

RE is Hard, Cont'd

- **we may think we understand a lot, but our understanding may be wrong,**
- **requirements change as clients' needs change,**

RE is Hard, Cont'd

- **requirements change as clients and users think of new things they want, and**
- **requirements of a system change as a direct result of deploying the system, as pointed out by Meir Lehman.**

Sources of RE Difficulties

- **RE is where informal meets formal (says Michael Jackson [1995]).**
- **Many requirements are created, not found.**
- **Users, buyers, even developers may be unknown.**
- **Stakeholders have conflicting objectives.**
- **Multiple views exist.**
- **Inconsistency must be tolerated, for a while.**
- **Requirements evolve during and after development.**

The TRUTH About Methodology Literature

Did you ever notice how error- and backtracking-free are example developments from clean requirements in the methodology literature?

Have you ever noticed how your own developments never go quite as smoothly and how your requirements are never totally right?

Truth, Cont'd

As an author of some of this literature, I will let you in on a little secret!

Shhh!

What you see in the literature is cleaned up from the rather messy real-life development.

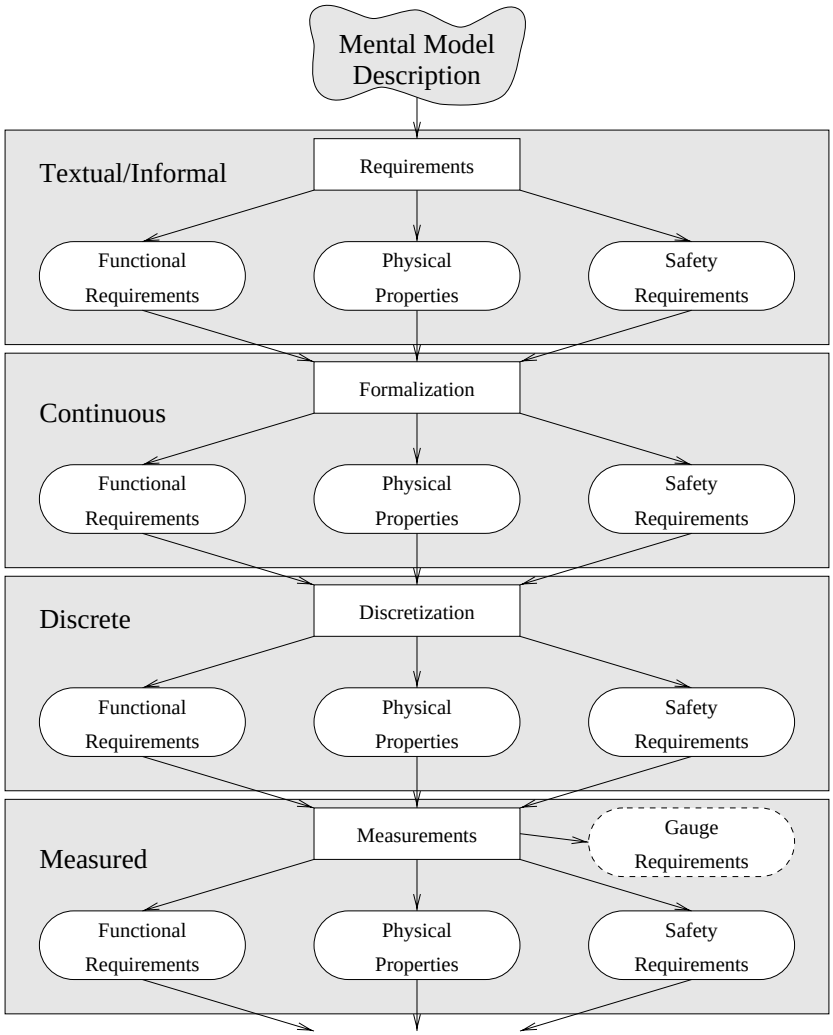
The requirements specification was modified more than the design and code.

Nu?

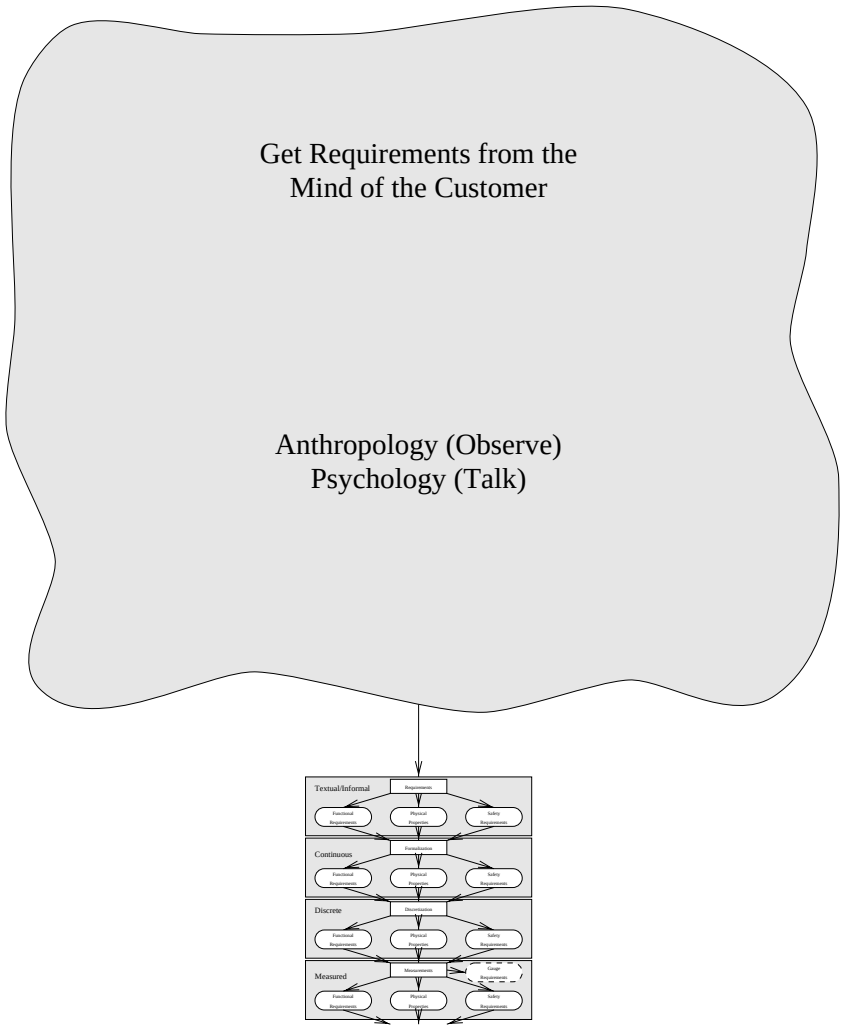
Montenegro's View

Sergio Montenegro described the reality by showing an older view of the requirements engineering process:

and then a newer view, formed after hearing an earlier version of this talk:



Subprocesses of the Requirements Phase



Relation between Getting and Formalizing Requirements

Why Important to Do RE Early

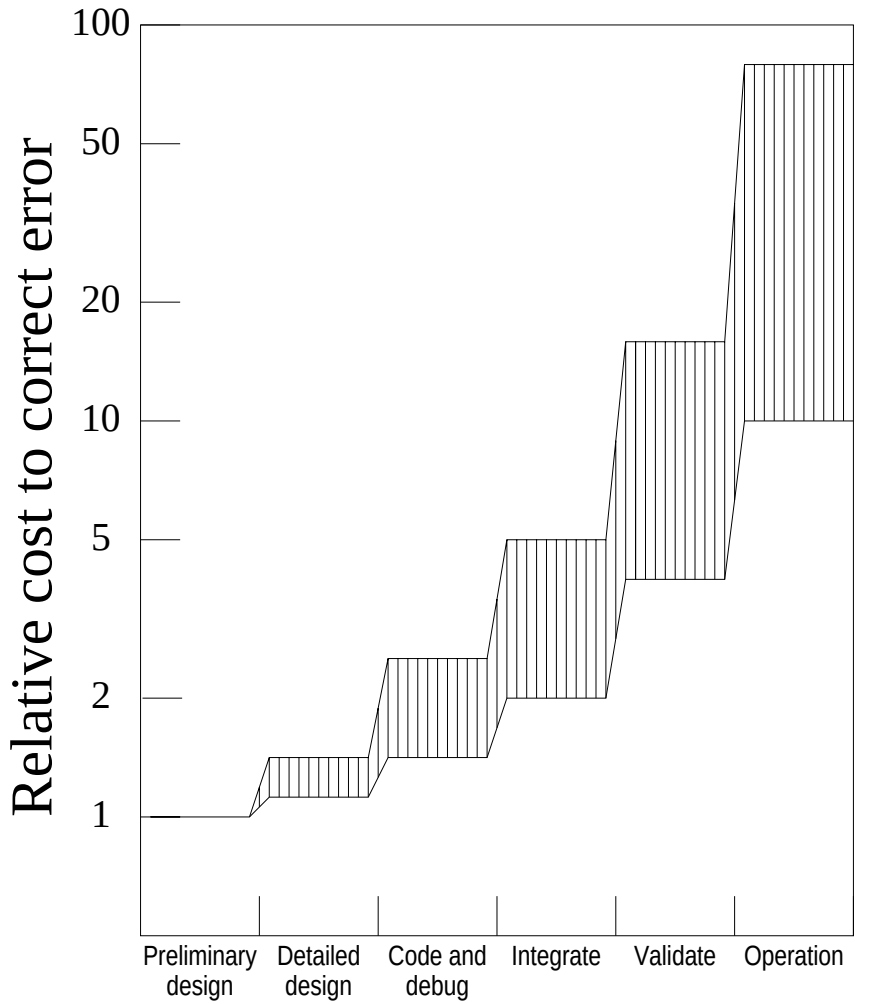
The BIG Question:

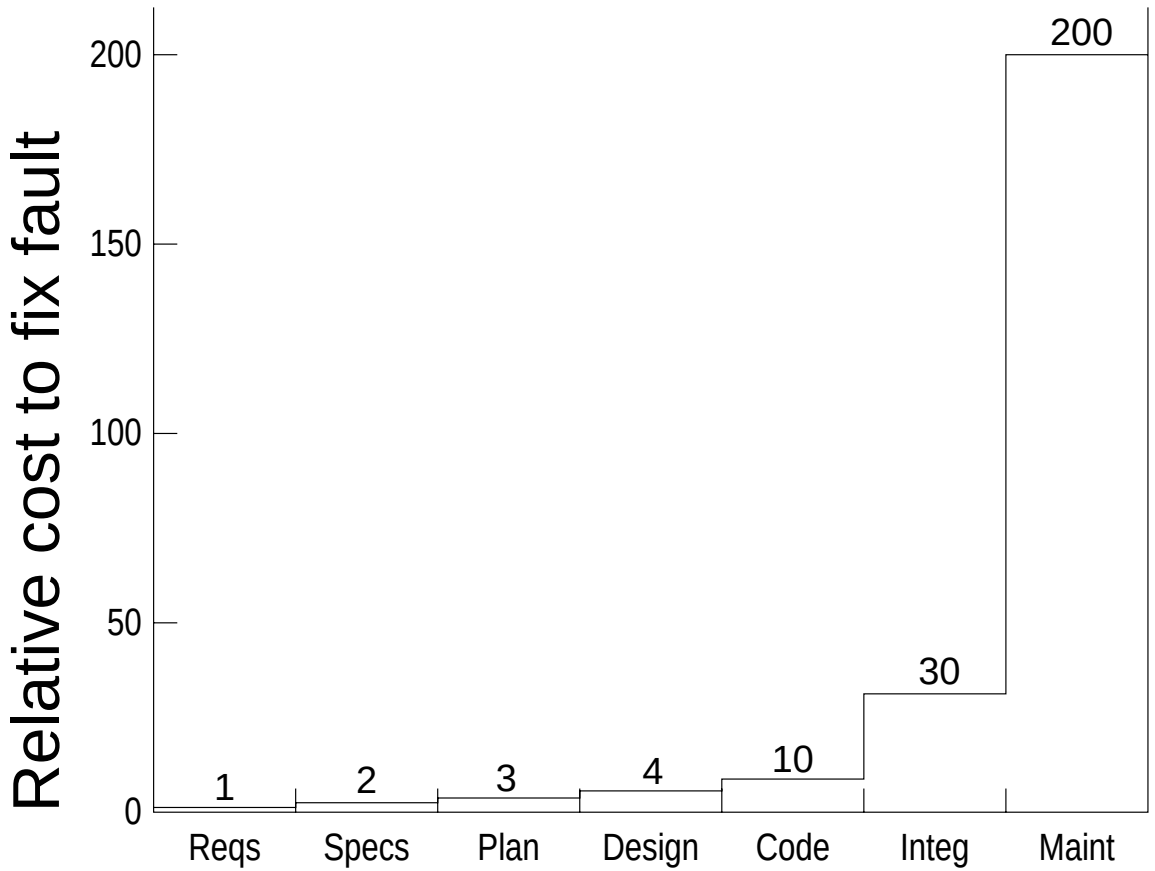
Why is it so important to get the requirements right early in the lifecycle? [Boehm 1981, Schach 1992]

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.

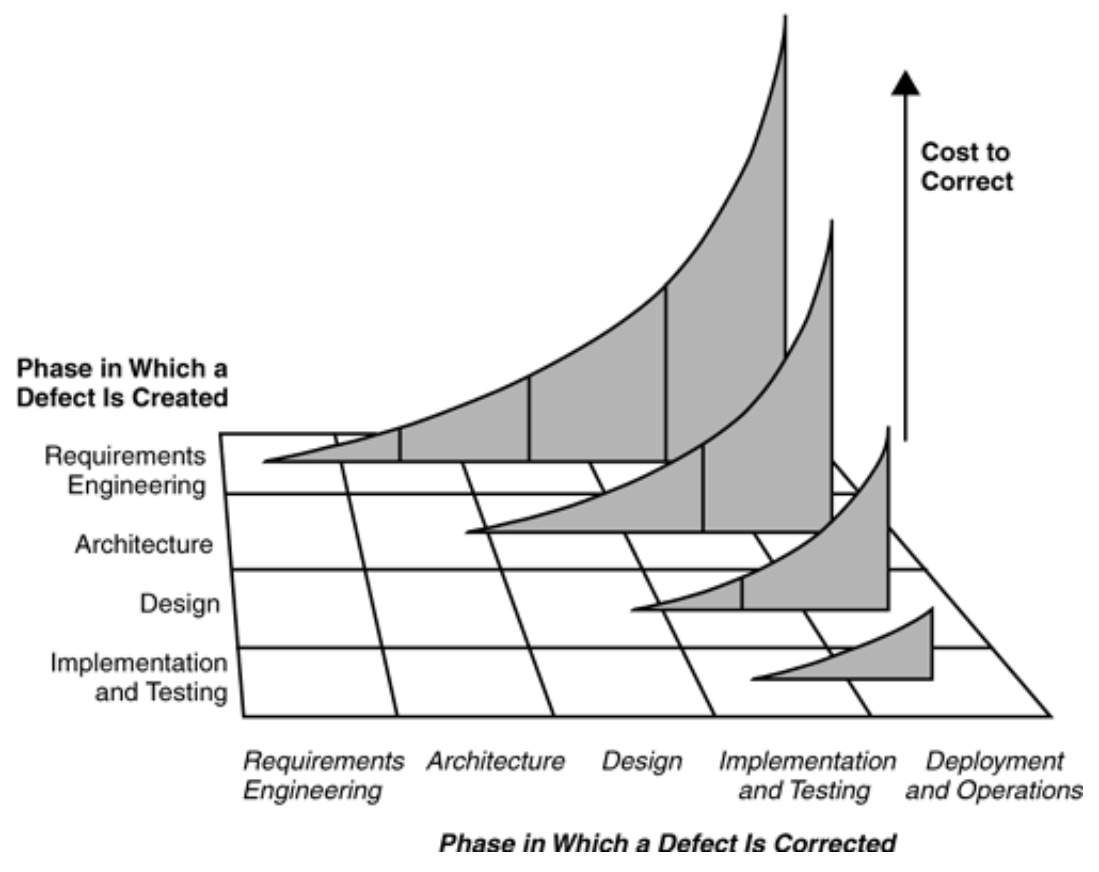

Relative cost to correct error
100
50
20
10
5
2
1
Preliminary design
Detailed design
Code and debug
Integrate
Validate
Operation
Phase in which error is detected


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!

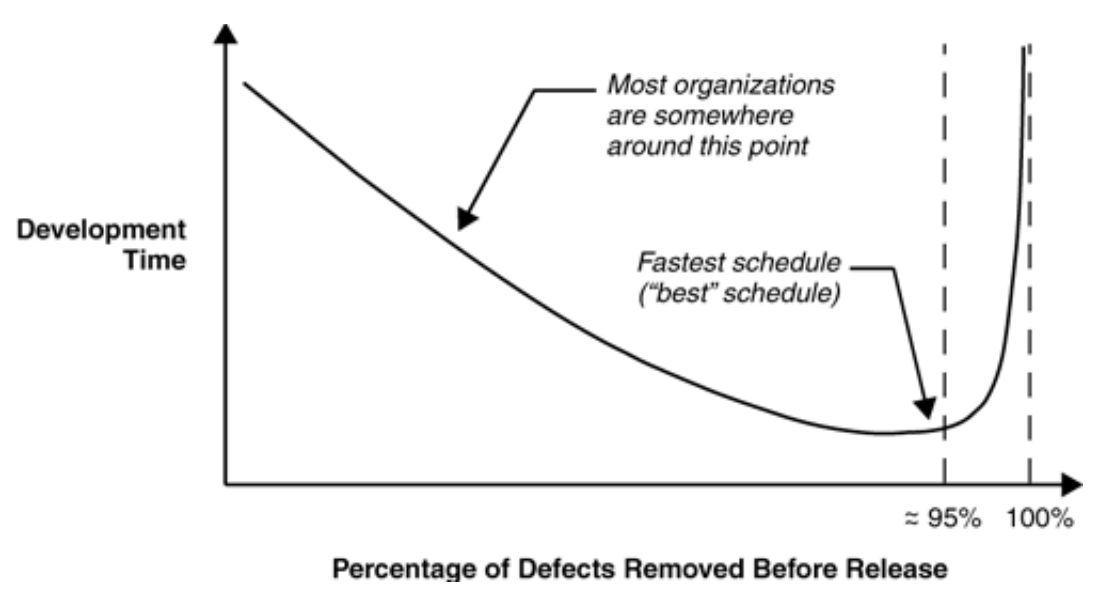
Conclusion, Cont'd

Allen et al [2008] show a graph of how

total development time of software

relates to

the percentage of defects removed before release of the software.


Most organizations are somewhere around this point
Development Time
Fastest schedule ("best" schedule)
≈ 95%
100%
Percentage of Defects Removed Before Release

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!

User Interfaces (& Errors)

The same is true about good user interfaces.

They cannot be programmed in later.

They must be required in from the beginning [Shneiderman 1984, Kösters, Six & Voss 1996].

Same is true also about system responses to erroneous input.

RE & Project Costs

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

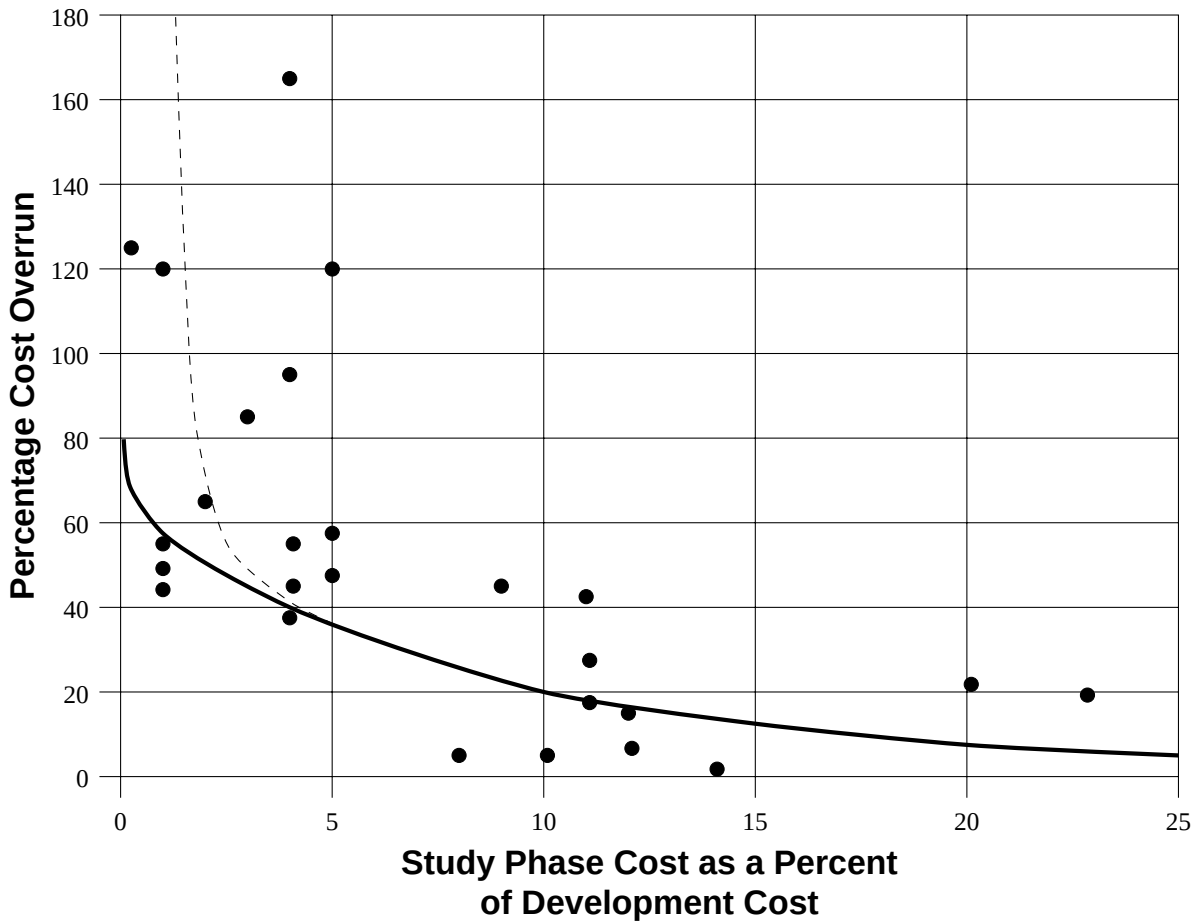
They contain a graph by Kevin Forsberg and Harold Mooz [1997] relating percentage cost overrun to study phase cost as a percentage of development cost in 25 NASA projects.

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.

Some Advantages of Project Delay

Arnis Daugulis [1998] reports a significant increase in the quality of the requirements for a power plant control system as a result of delays in start of production caused by shortage of funds.

They had the luxury of time to do two revisions of the requirements.

He shudders to think of the failure that would have resulted had they started to implement the first requirements on time.

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.

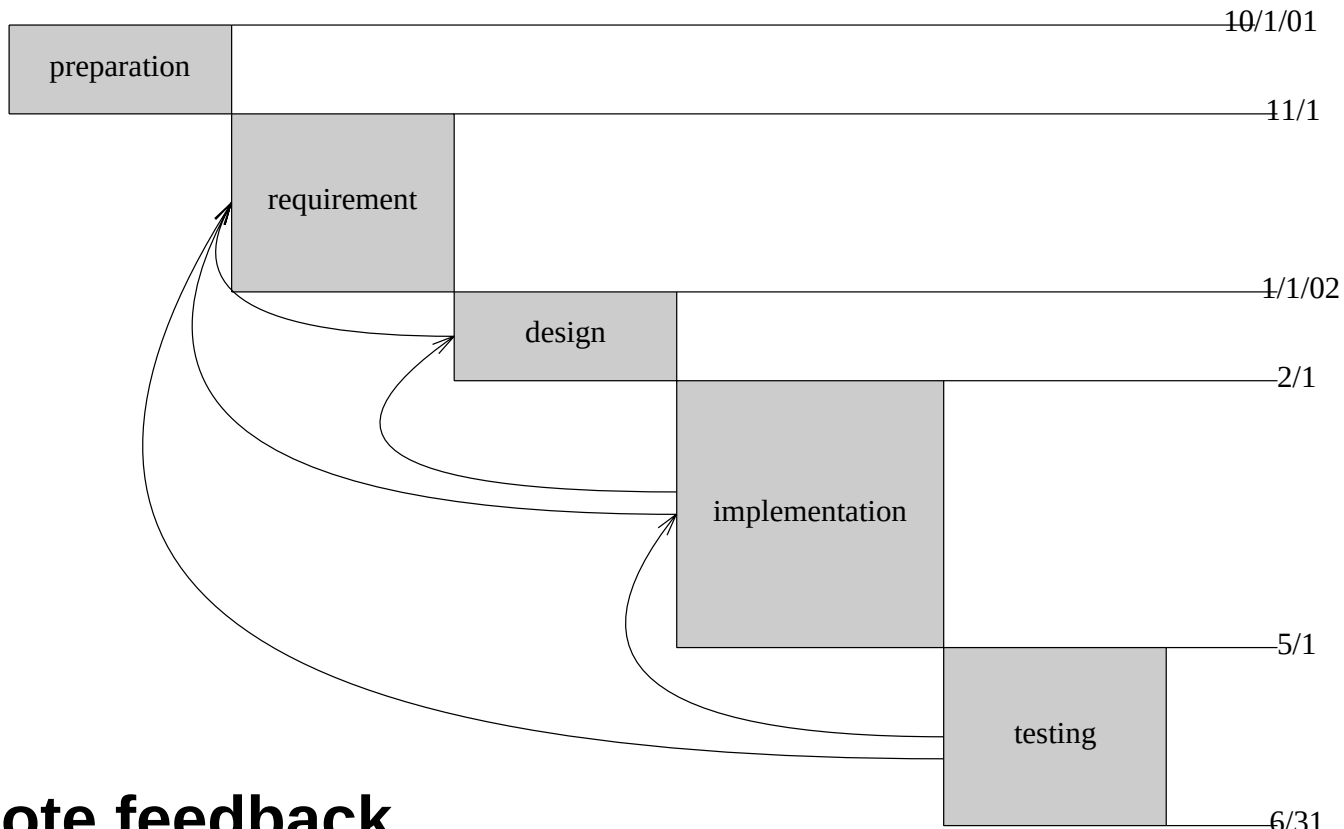
Ou's Assignment, Cont'd

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

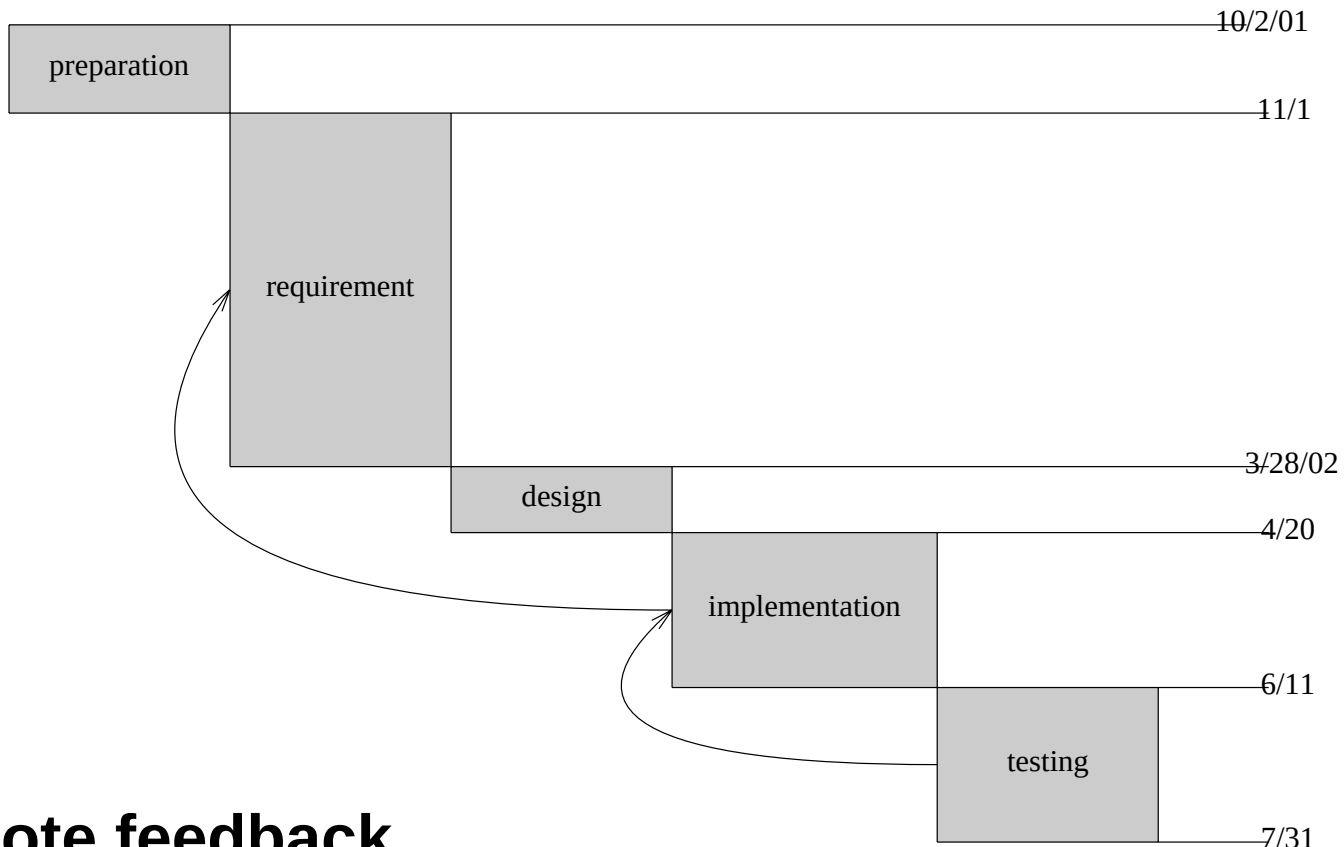
Duration in months	Step
1	Preparation
2	Requirements specification
4	Implementation
2	Testing
1	Buffer (probably more implementation and testing)
10	Total planned



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.

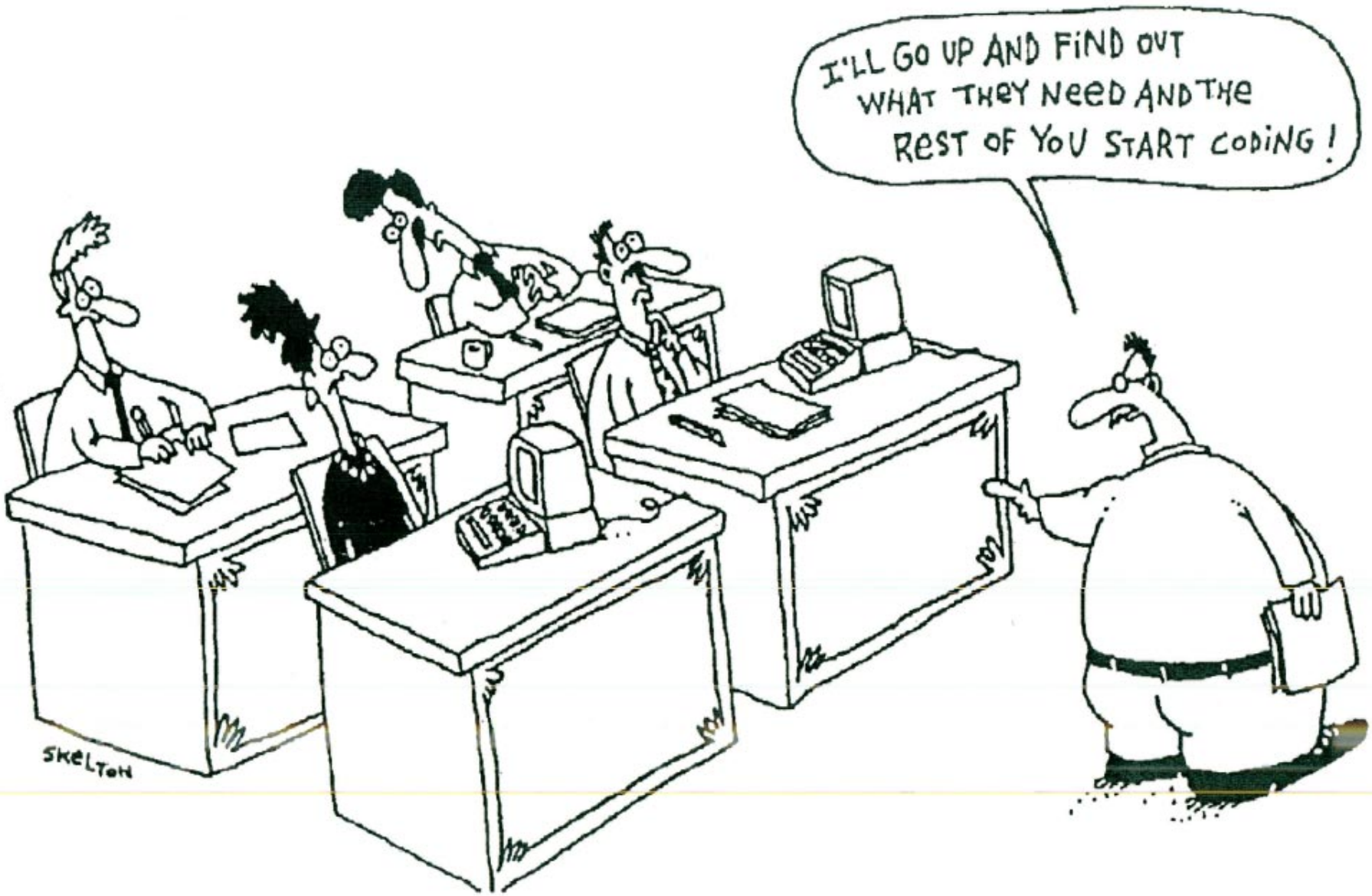
Success!

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.**

Coding before RE, Cont'd

According to Ruth Dameron, ...

> **The programmer who says the first line is suffering from the myth that the customer would be able to know what he or she wants and to say it just because the programmer asked.**

Need Prototype

Most people (especially non-technically oriented) learn while doing; they've got to see some kind of prototype (even if it's only yellow stickies on a board) to *discover* what they want.

First also expresses the nonsensical notion that somehow, coding can begin before it's known *what* to code.

Prototyping

Actually, there *is* a circumstance in which it's good to start coding before requirements are completely known.

To develop a throw-away prototype from which to learn requirements.

But, it should be a throw away!

IKIWISI

I’ll know it when I see it!

This is how most clients really identify requirements.

They cannot tell you what they want, but if they see what they want, they spot it immediately!

Sometimes, anything you show them is it (AYSTII).

IKIWINT & IKIWIDSI

Flip sides of IKIWISI (identified by Janusz Dobrowolski)

- **IKIWINT—I'll know it when it's not there!**
- **IKIWIDSI—I'll know it when I don't see it!**

We Don't Have Time for RE

"***I know that it is important to get requirements, but we don't have time for it; we have to get to coding to meet our deadline!"***

We Don't Have Time, Cont'd

At least one Ph.B. has been heard to say, "Wally, we don't have time to gather the product requirements ahead of time. I want you to start designing the product anyway. Otherwise it will look like we aren't accomplishing anything."

Wally stops working because he knows that the project is doomed anyway.

We Don't Have Time, Cont'd

I will prove that we, in fact, always do write requirements during the normal commercial software lifecycle.

Therefore, since we always do it, we must have had enough time!

Never Needed RE Before

"We've* never *written a requirements document and we're still successful!"

So says the manager of a project that delivered tested software with a user's manual or an on-line help.

So says the manager justifying a decision to plunge into development without first determining and specifying requirements.

Sorry to Disappoint You!

Kamsties, Hörmann, and Schlich [1998] observe:

Any project that does testing has to determine the requirements in order to determine covering test cases and their expected outputs. The test plan ends up being a requirements specification.

Sorry, Cont'd

Any project that produces user documentation has to determine the requirements in order that the documentation describe all of the features well. The documentation ends up being a requirements specification.

Even Microsoft does both.

So there is no escaping determination and specification of requirements (unless you don't do testing and user documentation!).

Sorry, Cont'd

There is no avoiding the time required to determine and specify the requirements.

However, if you produce requirements only as a side effect of testing and documentation, you lose the key benefit of *early* requirements determination and specification, namely finding errors at the least cost.

Sorry, Cont'd

If you stop official requirements determination, to move on to coding, and the requirements are not completely determined for the current scope, then …

programmers make microdecisions to fill in on missing requirements all along the Michael Jackson continuum.

Sorry, Cont'd

What are the chances that these programmer-determined requirements, often made in the best interest of the programmer, will match the client's true requirements?

Discovery of requirements during writing of test cases generally leads to massive rewriting of code whose architecture has no place for the newly discovered requirements.

They'll change anyway!

"Why bother writing down requirements now? We'll discover that they're wrong and have to change 'em anyway!", …

as a reason not to write a requirements spec up front and to just start coding.

They'll change anyway! Cont'd

You are right!

The requirements you write now *will* be wrong, and

you *will* have to change 'em.

They'll change anyway! Cont'd

But the question is:

"How will you discover that the requirements are wrong?

What will tell you that you have made a mistake in the requirements?"

They'll change anyway! Cont'd

In my experience, we discover that we have made a mistake *only* as a *result* of writing them down.

After all, until we write the requirements down, while they are still ideas in the air, they are perfect!

But that's only an illusion, arising from the fact that we have not fully explored the implications.

They'll change anyway! Cont'd

When we start writing things down, and *not* before, the implications begin to show.

So how do we usually discover that there is something wrong with the requirements?

They'll change anyway! Cont'd

In my experience, we make these discoveries only when we write down something that contradicts something we have written before.

So the mere writing of the specs is what exposes the wrong requirements that have to be changed.

They'll change anyway! Cont'd

Thus, if you don't write the requirements down *before you start the coding*, …

and discover the wrong requirements before you start the coding, …

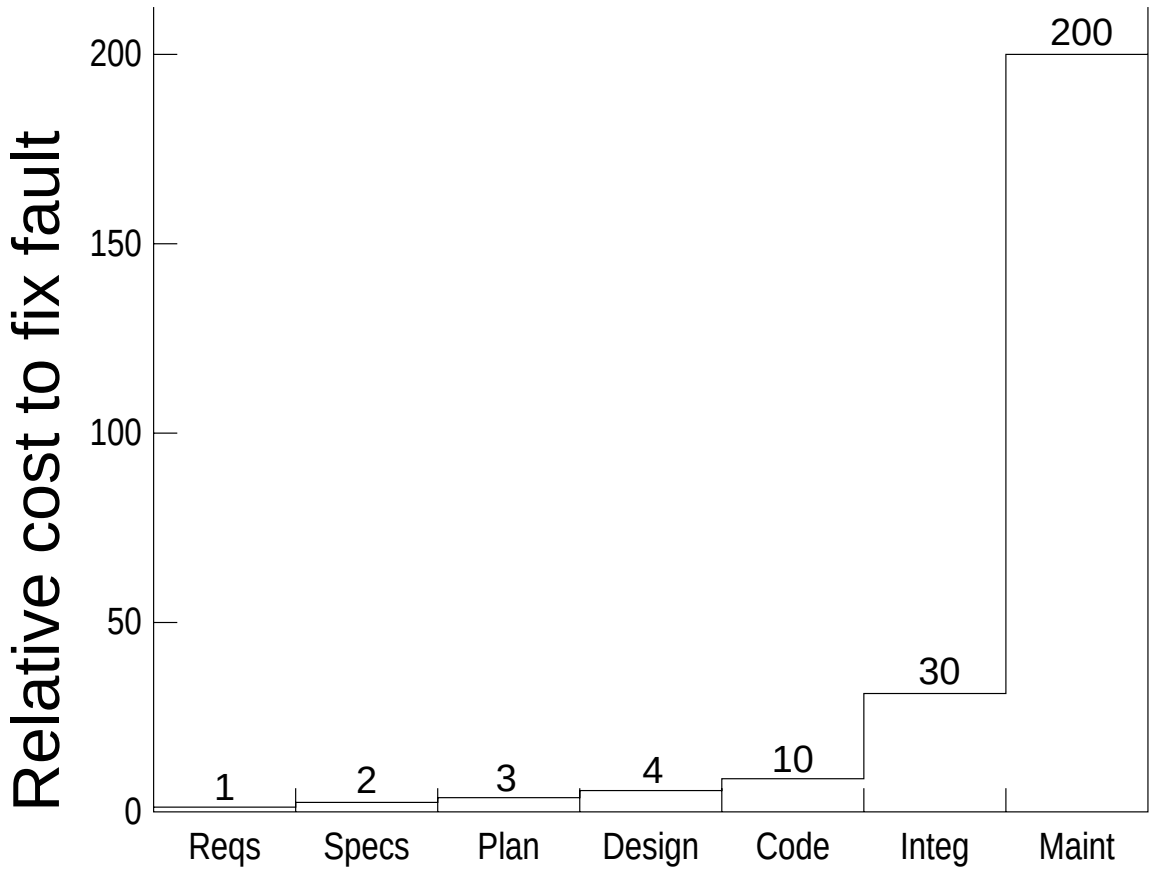
you *will* discover the wrong requirements when you actually start writing something down, *during coding*.

They'll change anyway! Cont'd

But then it will cost 10 times more to fix that if you had made the discovery before you started coding.

Oy!!!!

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.

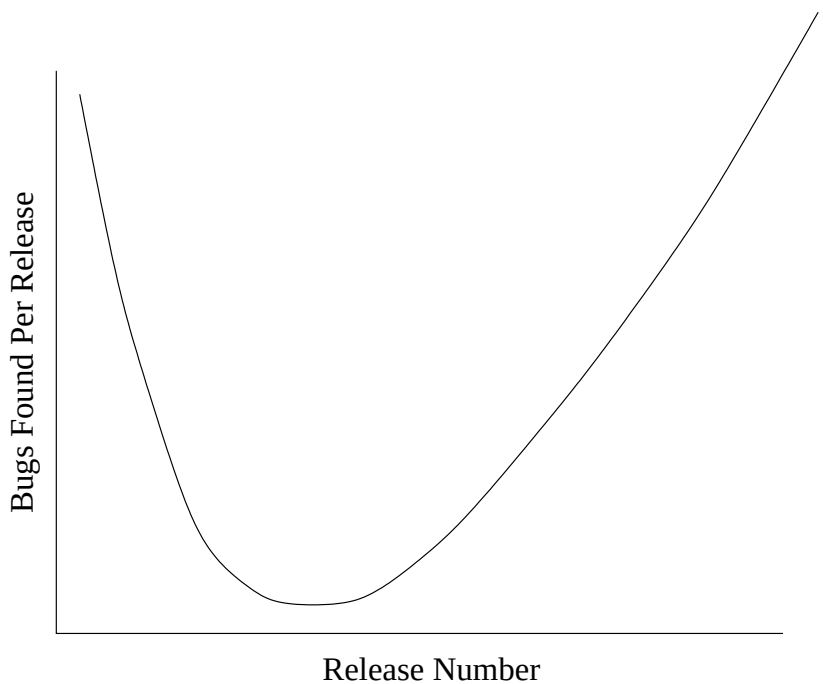
Decay of SW Cont'd

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.**

Purpose of Methods

One view of software development methods:

Each method has as its underlying purpose to *tame the B-L graph* for the CBS developments to which it is applied.

That is, the method tries

- **to delay the beginning of the B-L upswing or**
- **to lower the slope of that B-L upswing or**
- **both.**

Information Hiding

For example, David Parnas's (1972) Information Hiding (IH)

hides implementation details to make it possible to change the implementation of an abstraction by modifying the code of only the abstraction and not of its users.

IH, Cont'd

Thus, for any implementation change, only one module is changed and the architecture is *not* changed.

B-L upswing is delayed and its slope is reduced.

∴, B-L upswing is tamed!

Now back to the main thread!

May Not Even Have a Problem!

The client often says that he or she requires a specific solution of an unknown or nonexistent problem rather than any solution to a specific problem [Gause and Weinberg 1990].

"*We gotta automate!*"

The problem with such a client is that there may not even be a problem that requires any solution, or if there is, other solutions, including non-computer, may be better!

Another Myth

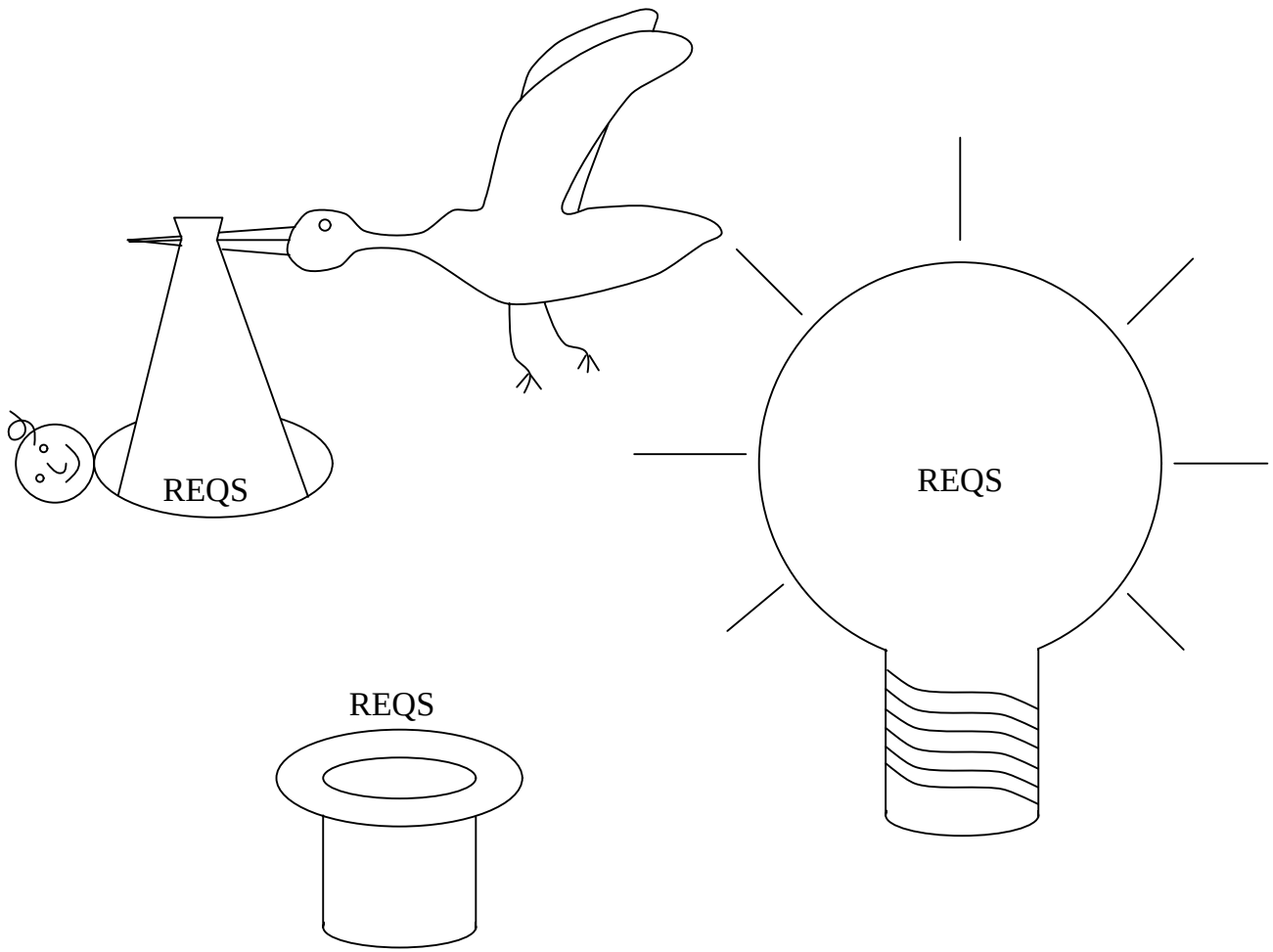
After the requirements are frozen, ...

Ha!
Ha ha!
Ha ha ha!
Ha ha ha ha!

The only clients that are satisfied and have stopped asking for changes are themselves frozen!

We have already seen why requirements will *never* be frozen!

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

Wally says to his Ph.B., "I can't start the project because the user won't give me his requirements.". The Ph.B. replies, "Start making something anyway. Otherwise we'll look unhelpful." Wally replies, "So, our plan is to cleverly hide our competence." The Ph.B. observes, "You think too much."

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!

Creativity

Roberto Tom Price reminds us not to forget the requirement engineer's

- **imagination**
- **ideas**
- **suggestions**

i.e. *creativity* [Maiden, Gizikis, & Robertson 2004]

like an architect for a new building, using input from the client

More About Creativity

Neil Maiden and Alexis Gizkis [2001] in asking from where do requirements come, identify 3 kinds of creativity needed in RE:

Three Kinds of Creativity

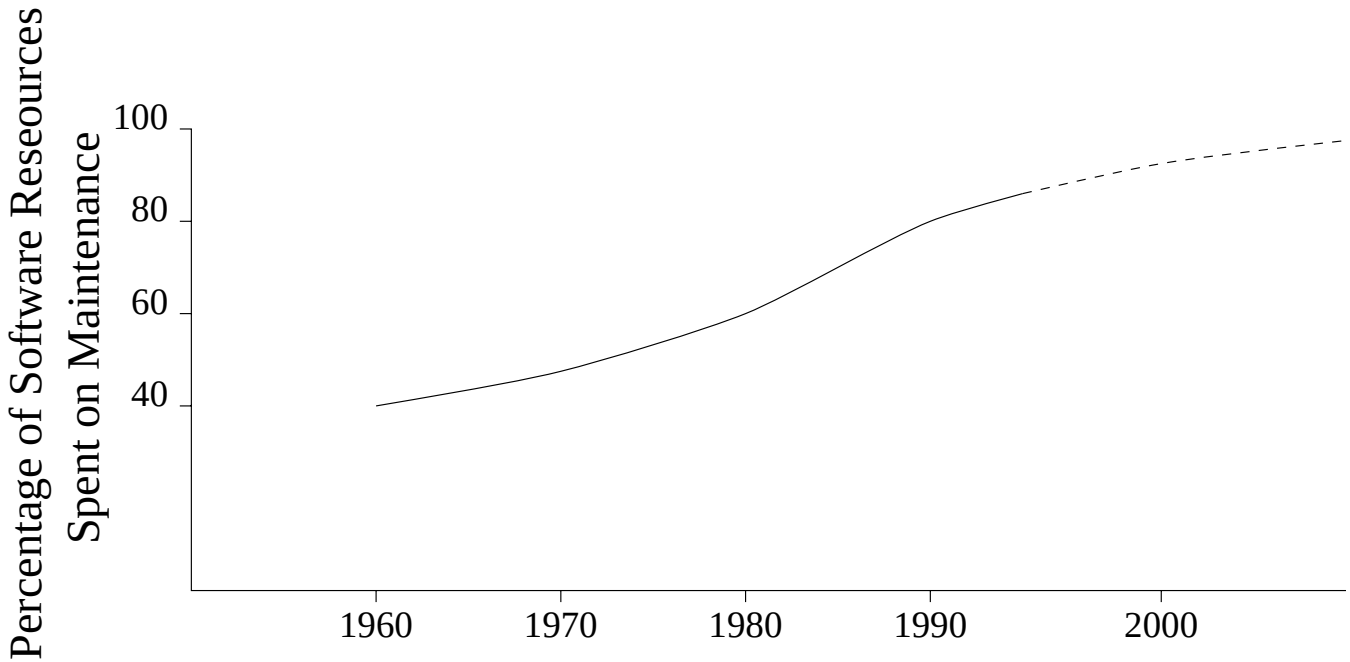
1. **exploratory creativity: explores a possible solution space and discovers new ideas**
2. **combinatorial creativity: combines two or more ideas that already exist to create new ideas, and**
3. **transformational creativity: changes the solution space to make impossible things possible.**

Whence, Cont'd

Goguen further observes that most of the effort for a typical large system goes into maintenance.

In fact, Parnas [1994] has the data:

Whence, Cont'd

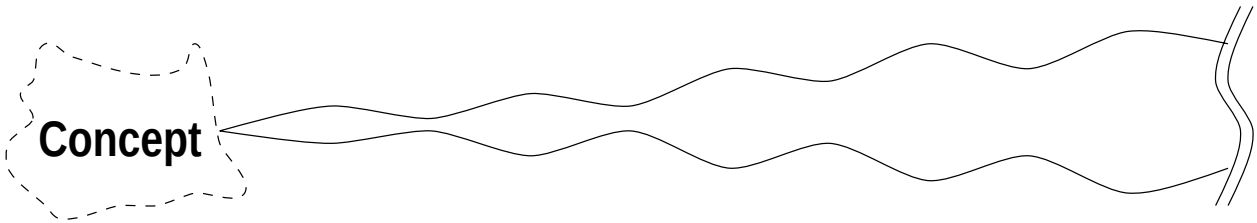


Growing Percentage of Maintenance Costs

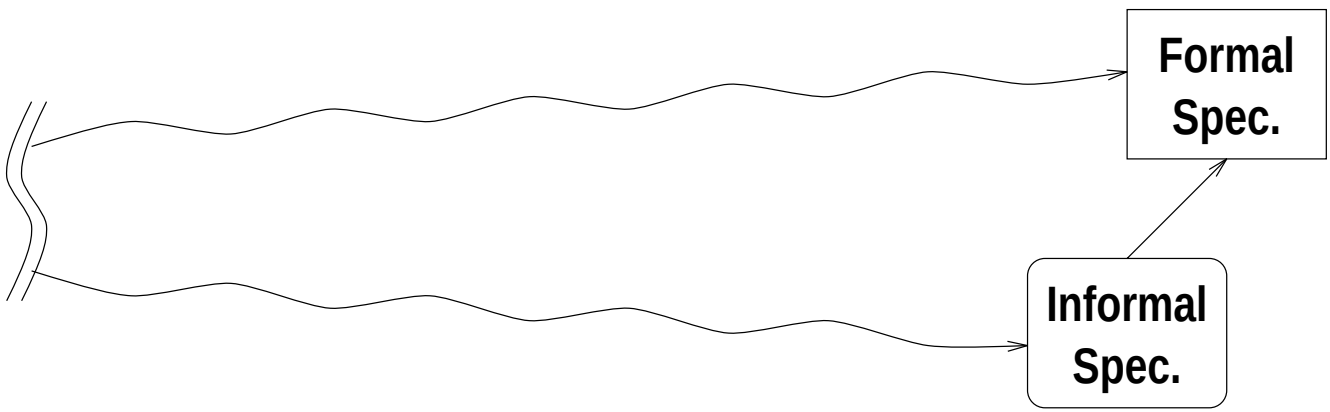
Formal Methods Needed?

Some formal methodologists say that this is the fault of insufficient effort put into being precise in the early, specification stages of software development.

However, recall the conceptual distances involved:



Folded in middle to give feeling of true conceptual distances involved



FMs Needed, Cont'd

Goguen believes that "a deeper reason is that much more is going on during so-called maintenance than is generally realized. In particular, reassessment and re-doing of requirements, specification, and code, as well as documentation and validation, are very much part of maintenance...."

FMs Needed, Cont'd

Later, he adds, "it only becomes clear what the requirements really are when the system is successfully operating in its social and organisational context.... it is impossible to completely formalise requirements ... because they cannot be fully separated from their social context."

This is precisely the phenomenon of E-type systems.

Formal Methods Myths

Goguen has identified other myths about requirements, again based on the mistaken idea that the hard part about requirements are their specification.

FM Myths, Cont'd

If* only *you had written a formal specification of the system, you wouldn't be having these problems.

Mathematical precision in the derivation of software eliminates imprecision.

FM Myths, Cont'd

What is the reality?

Yes, formal specification are extremely useful in identifying inconsistencies in requirements specifications, especially if one carries out some minimal proofs of consistency and constraint or invariant preservation, ...

just as writing a program for the specification!

FM Myths, Cont'd

Don't get me wrong.

This formality is good, because it finds errors early, thus reducing the costs to fix them.

However, formal methods do *not* find all gaps in understanding!

FM Myths, Cont'd

As Eugene Strand and Warren Jones [1982] observe, "Omissions of function are often difficult for the user to recognize in formal specifications"....

just as they are in programs!

FM Myths, Cont'd

von Neumann and Morgenstern (*Theory of Games* [1944]) say,

"There's no point to using exact methods where there's no clarity in the concepts and issues to which they are to be applied."

Preservation of Difficulty

Indeed, Oded Sudarsky has pointed out the phenomenon of *preservation of difficulty*. Specifically, difficulties caused by lack of understanding of the real world situation are not eliminated by use of formal methods; instead the misunderstanding gets formalized into the specifications, and may even be harder to recognize simply because formal definitions are harder to read by the clients.

Bubbles in Wall Paper

Sudarsky adds that formal specification methods just shift the difficulty from the implementation phase to the specification phase. The “air-bubble-under-wallpaper” metaphor applies here; you press on the bubble in one place, and it pops up somewhere else.

One Saving Grace

Lest, you think I am totally against formal methods, they *do* have one positive effect, and it's a BIG one:

Use of them increases the correctness of the specifications.

Therefore you find more bugs at specification time than without them, saving considerable money for each bug found earlier rather than later.

Analogy with Math Theory

Building new software is like building a new mathematical theory from the ground up:

- **Requirements gathering: deciding what is to be assumed, defined, and proved**
- **Development as a whole: assuming assumptions, defining the terms, and proving the theorems**

Math Theory, Cont'd

- **Design: determining the sequence of assumptions, definitions, and theorems to build the theory**
- **Implementation: carrying out the designed theory and proving the theorems**

Math Theory, Cont'd

Mathematical papers and books show only the results of the implementation,

This implementation is what is considered the mathematics, and not the requirements gathering and design that went into it.

But I know, from my own secret past life as a mathematician, that the hard, time-consuming parts are the requirements gathering and design.

Math vs. SW Development

- **Software is usually developed under strict time constraints, and mathematics is usually not.**
- **Mathematics development is subjected to error-eliminating social processes, and software development is subjected to a lot less.**

Math vs. SW, Cont'd

- **Mathematics is written for a human audience that is very forgiving of minor errors so long as it can see the main point; software is written for the computer that is very literal and unforgiving of minor errors**

  - **For people, UKWIM works, and they accept imprecision.**
  - **For computers, UKWIM and DWIM do not work, and computers do not accept imprecision.**

Another Implication of Growing Maintenance Costs

For many programs, which are more and more often enhancements of legacy software, any original requirements specifications that may have existed are long gone.

The original programmers are long gone.

The old requirements have to be inferred from the software.

Another Implication, Cont'd

What is inferred may not capture all features.

Also the obvious requirement of not impacting existing functions in the enhancement is very easy to state, but, oh, so hard to satisfy.

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]).

RE and Other Engineering

Speaking of architects and other engineers that get requirements from clients, ...

While software requirements gathering has much in common with requirements gathering for buildings, bridges, cars, etc., there are significant differences in:

- **the flexibility and malleability of the medium, and**
- **the degree to which basic assumptions are on the table, up for grabs.**

Other Engineering, Cont'd

Michael Jackson [1998] considers the more traditional engineering disciplines.

Civil Engineering
Mechanical Engineering
Aeronautical Engineering
Electrical Engineering

Their engineers make machines by describing them and *then* building them.

Other Engineering, Cont'd

Software engineers make machines merely by describing them.

Software is an intangible, infinitely malleable medium.

Other Engineering, Cont'd

To build a new car from requirements the way software is built from requirements would be called *totally rethinking the automobile*. In fact, each new kind of car is really a minor perturbation of existing kinds of cars.

Other Engineering, Cont'd

Perhaps, we should do much more minor perturbation of existing software, i.e., practice reuse, but in many cases we cannot, simply because we are developing software for an entirely new application.

Bottom Line

The notions that

- **one can derive requirements**
  or
- **even interview a few people to get requirements**

are patent nonsense.

Ample Evidence That

- **The later an error is detected, the more it costs to correct it.**
- **Many errors are made in requirements elicitation and definition.**
- **Many requirements errors can be detected early in the lifecycle.**
- **Typical errors include incorrect facts, omissions, inconsistencies, and ambiguities.**
- **Requirements problems are industry's biggest concern.**

OK, OK, You're Convinced!!!

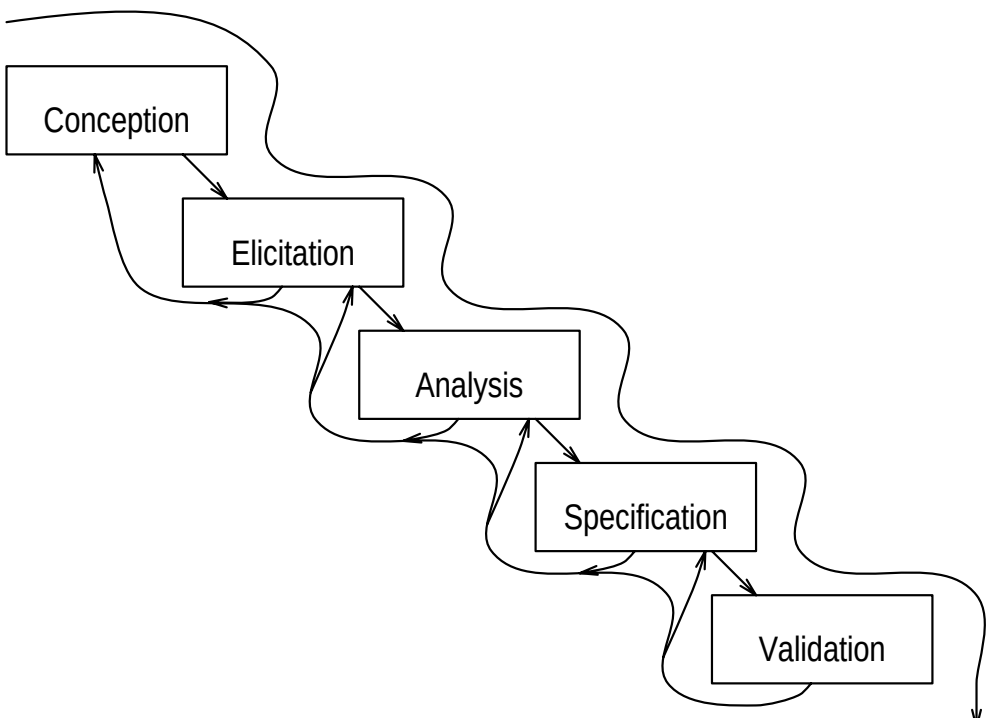
So what do we DO about it?

What research is being done to solve the problems?

First, recognize that the problem is HARD!

Second, recognize that requirements engineering has its own lifecycle

Requirements Engineering Lifecycle



RE Lifecycle, Cont'd

This, of course, is an idealization, just as much as the original waterfall.

Reality is that there is a spiral, with each sweep going through this entire subwaterfall, and all the steps in the sweep happening concurrently.

Another RE Lifecycle

Kevin Ryan offers the following RE process:

- **Identify Requirements**
- **Document Requirements**
- **Validate Requirements—Have we elicited and documented the right requirements?**
- **Verify—Have we got the model right— and Validate—Have got the right model—Models**
- **Rank Requirements by Priority**
- **Select and Plan Requirements**

Scoping

Some facts:

- **Many projects fail because what was required was too big for the available resources, and there is no way to subset the requirements, i.e., it's all or nothing!**
- **80% of execution is in 20% of the code, a rule of thumb used by compiler writers.**

Scoping, Cont'd

- **1998 Standish Group data [Neumann 1999] show that among features in sampled mass-marketed applications,**
  - **45% are never used,**
  - **19% are rarely used,**
  - **16% are sometimes used,**
  - **13% are often used, and**
  - **only 7% are always used.**

Must Scope Requirements

The solution to these problems is to scope, but how?

Must rank requirements [Davis 2003]

1. **absolutely essential**
2. **essential**
3. **important**
4. **nice**
5. **fluff**

Subsetting Requirements

Select an affordable, coherent subset of the requirements consisting of the most important requirements.

OR

Build to cost, perhaps in a spiral.

Use Cases and Scenarios

Consider an interactive computer-based system (CBS), *S*. It is called interactive because its users interact with it.

A user tends to think of *S* in the ways that he or she will use it.

Using a CBS -1

For example, if *S* is an electronic voting system, the average voter user tends to think of two particular ways to use *S*:

- **registering to vote, and**
- **voting, i.e., marking a ballot in an election.**

Using a CBS -2

There are other users of this *S* that think of other particular ways to use the *S*.

For example, the voting manager user has other particular ways to use *S*:

- **preparing a ballot for a particular election,**
- **opening the polls for a particular election for voting by voters,**
- **closing the polls for a particular election,**
- **counting the votes for a particular election.**

Using a CBS -3

Of course, the voting manager is also a voter and can also register to vote and vote.

Use Case

Each such particular way to use *S* is called a *use case*.

It is one case of the many ways to use *S*.

Scenario

A use case should not be confused with a closely related concept called a *scenario*. A scenario of *S* is a particular sequence of interaction steps between a user of *S* and *S*.

Use Cases and Scenarios

The relation between use cases and scenarios can make both terms clearer.

A single use case contains many, many scenarios.

A use case *UC* of *S* has a so-called *typical* scenario. This scenario is that identified by the stakeholders of *S* as being the normal case of *UC* that proceeds with all decisions being made in the so-called normal or typical way.

Variations

***UC* also consists of variations called *alternatives* and *exceptions*.**

Alternatives -1

An alternative of *UC* is a sub-use-case, which may include many scenarios, that achieves the main goal of *UC* through different sequences of steps or fails to achieve the goals of *UC* although it follows most of the steps of *UC*.

Alternatives -2

For example, for the *voting* use case, the typical case is voting in one session.

One alternative is voting in multiple sessions.

Another alternative is starting to vote, but not finishing.

Exceptions

An exception of *UC* is a sub-use-case, which may include many scenarios, that deals with the conditions along the typical scenario and other sub-use-cases that differ from the norm and those already covered.

For example, for the *voting*, what to do with a non-registered voter trying to vote is an exception.

Misuse Cases

Sindre and Opdahl [2000] and Alexander [2002] have proposed *misuse* cases to capture use cases that a system must be protected against, that you do not want to happen at all, e.g., a security breach.

Sharing of Subsequences

Observe that all the scenarios of a use case share many subsequences of steps.

Some sets of these subsequences of steps may constitute sub-use-cases that are worth considering as use cases that can be used by other use cases.

For example, the *voting* and *registering* use cases may share the steps for validating that a user is a registered voter.

Power of Scenarios

Aren't scenarios nice? They can be used

- **to help elicit requirements**
- **as a basis for user validation of requirements understanding**
- **as a basis of an active review of specifications**
- **as a basis of a quick and dirty prototype**
- **as a basis of covering black-box test cases for implementation**
- **as a basis for writing a user's manual**

Wow!!

Agility vs. Full RE

In situations in which high-level requirements are not fully understood, ...

in which prototyping should be done to help identify requirements and to validate them with the customers and users, ...

and iterative, incremental, or Spiral models are appropriate, ...

Agility for Prototyping

then agile methods (AMs), such as eXtreme programming (XP) [Agile Alliance, Highsmith & Cockburn 2001, Eberlein & Leite 2002], help focus quickly to produce a series of well-inspected [Tomayko 2002] and well-tested increments, and can be used to carry out the prototyping effort ...

But Must Refactor

so long as it is understood that the code produced is a prototype and that it must be tossed in order that development of a high-quality production version be started. In XP, terminology, this is known as massive refactoring.

Power of XP

After all, in XP, each iteration starts with the writing of stories (i.e., scenarios) describing functions to be implemented in this iteration *and* test cases for testing that the functions have been implemented correctly.

This is far better than in most prototyping regimes, and the result should be higher quality software than under most prototyping regimes.

However...

However, *all* the data I have seen say that full RE is the most cost-effective way to produce quality software, and that it will beat out any AM any time in the cost and quality attributes.

When all the details are hammered out during an RE process that lasts until it is done (and not until a predetermined date), the development proceeds so much quicker and with so few bugs!

Tradeoff

What is the tradeoff?

Full RE Pros & Cons

Full RE leads to better and better understood code

But risks failure to complete as nervous managers pull the plug or as funding runs out

AMs Pros & Cons

Incremental and AMs make sure that you always have something that runs

But risks always having an incomplete, incorrect, and flaky program.

Agility vs Full RE

Therefore, if the problem is just getting the high-level requirements right, then go the incremental or agile development route, ...

but if the problem is fleshing out the requirement details given a clear vision, then go full RE route.

RE is Still an Art

And most importantly, ...

There are no real solutions yet!

It is very much an art form.

Research Topics

Earlier Work, prior to mid-80s

Later Work, mostly after mid-80s

Some temporal overlap, because as we will see, the work is classified by nature, and that nature has changed slowly.

I apologize in advance if I have left out things about which I am not aware.

Earlier Work-1

Languages and Tools:

PSL/PSA [Teichroew 1977]
SADT [Ross and Ross & Schoman 1977]
RSL [Alford 1977]
RDL [Winchester 1982]
PAISley [Zave 1982]
RML [Borgida, Greenspan & Mylopoulos 1985]
IORL [Salton & McGill 1983]

Earlier Work-2

Alan Davis wrote the book! *Requirements Analysis and Specification* [1990]

Focus was on analysis and specification, *not* on elicitation

Later Work-1

More consideration of elicitation

Recognition of importance of sociology and psychology

Later Work-2

- **Elicitation**
- **Analysis**
- **Natural Language Processing**
- **Tools and Environments**
- **Changes**
- **Empirical Studies**
- **RE as a Human Activity**

Global View-1

Requirements Engineering **is:**

How to squeeze requirements out of the client's mind and environment without damaging the client or the environment!

Elicitation **is:**

How to squeeze information out of the client's mind without damaging the client!

Global View-2

***Analysis* is:**

How to squeeze as much additional information as possible out of what has been obtained by squeezing the client and the environment!

***Natural Language Processing* (NLP) is concerned with:**

How to automate as much of the analysis squeezing as possible

Global View-3

Tools and Environments **deal with:**

How to automate the storage of information before and after analytic squeezing as well as all kinds of squeezing!

Changes **cause that:**

No matter how hard you squeeze, analyze, etc. there are new requirements, and the old ones change.

Global View-4

***Empirical Studies* are concerned with:**

Understanding squeezing by observing it or parts of it in real-life circumstances!

***RE as a Human Activity* is concerned with:**

Understanding how humans do the squeezing as a social activity.

Use of Exemplars in RE Research

Many areas of SE use published exemplars, e.g., the KWIC Index System, for research case studies.

Since the problem is *how* to get the information for requirements, published exemplars are too polished and too late.

What is normally done to prepare exemplars for publication *is* the subject

Post Mortem Reports

Post mortem reports from failed projects, e.g., the automation of the London Ambulance Service [Finkelstein & Dowell 96] dispatching system, provide useful insight in how *not* to do requirements engineering.

Interestingly, the LAS learned its own lessons and won the 1997 British Computer Society Excellence Award!

Another is the CAPSA Report [Finkelstein & Shattock 2001]

Elicitation Overview-1

The gist of the specific work is:

- **Identify the stakeholders; these are the people you have to ask!**
- **The customer is always right.**
- **People matter.**
- **Interviewing does not get all the information that is needed.**

Elicitation Overview-2

- **The requirements engineer should get into the client's work place, blend into the woodwork or among the employees, and observe, learn, and question, in order to overcome ignorance of the problem domain.**
- **The requirements engineer should become an employee of the client to learn the ropes well enough to understand the underlying rationale behind the way things are done.**

Elicitation Overview-3

- **The requirements engineer should parlay his or her ignorance into on-the-spot questions that expose tacit assumptions and special cases.**
- **Don’t tell them what you mean; show them! This works in both directions!**
- **Prototype to capture emergent requirements and contextual factors.**

Elicitation Overview-4

- **Getting all stakeholders together for week-long facilitated meetings at which the requirements are shlogged out together buys commitment from all stakeholders.**
- **Scenarios and use-cases, which are descriptions of the ways that the intended system will be used to achieve specific or classes of tasks, are a useful way to focus users' attention on what they actually do and to document what they really do in the course of doing their work.**

Elicitation Specifics-1

- **Ignorance hiding in elicitation and analysis [Berry 1980, 1983]**
- **Scenarios and Use Cases [Hooper & Hsia 1982, Jacobson 1992]**
- **Concept of abstract user and prototype elicitation management tool [Burstin 1984]**
- **Brainstorming [Gause & Weinberg 1989]**
- **Contextual Inquiry, an anthropological approach to understanding client [Holtzblatt & Jones 1990, Beyer & Holtzblatt 1998]**

Elicitation Specifics-2

- **Study of discourses in elicitation techniques, e.g., interviews [Goguen & Linde 1993]**
- **Storyboarding & paper mockups [Zahniser 1993]**
- **Joint Application Development [Carmel, Whitaker & George 1993] [Wood & Silver 1995]**
- **Importance of ignorance in elicitation [Berry 1995]**

Elicitation Specifics-3

- **Stakeholder Identification [Sharp, Finkelstein & Galal 1999]**
- **Scenarios and Prototyping to capture emergent contextual factors [Carroll, Rosson, Chin, & Koenemann 1998, Dzida & Freitag 1998]**

Analysis Overview-1

- **Basic idea of analysis is to**
  - **derive implications of,**
  - **resolve inconsistencies in, and**
  - **determine what is missing from**

  the information that has been gathered so far so that follow up questions may be asked.
- **A key job of the analyst is to model the system under design and its environment. This is hard work, but from a good model, requirements are almost there for picking.**

Analysis Overview-2

- **Modeling the enterprise in which the requirements are situated is essential.**
- **The requirements engineer must constantly attempt to validate his or her understanding with the client or user.**
- **Prototypes are a nice way to make models and scenarios concrete to clients and users.**
- **Clients' and Users' validation responses to prototypes are far more credible than to long written specifications.**

Analysis Overview-3

- **One has to be careful about the information content of a prototype, what is there and not, what is intended and not, in short, its meaning.**
- **It is essential to be able to trace the history of a requirement from its conception through its specification and on to its implementation.**
- **Tracing allows determining the meaning of a prototype.**

Analysis Overview-4

- **Distilling scenarios into use cases is a useful way to put some order into the myriad scenarios described by diverse users.**
- **Error checking, handling, and prevention will not happen in a program *unless* they are explicitly required of the program; this is particularly so in safety-critical software for which the dangers are not readily apparent.**

Analysis Overview-5

- **While being specified, requirements *are* logically inconsistent.**
- **Goals and intents are important to understand and document.**
- **Ranking requirements by priority allows scoping, which in turn allows building to resources.**
- **Requirements and design affect each other.**

Analysis Overview-6

- **Dealing with non-functional requirements (NFR) is tough since often they are not quantifiable or expressible.**
- **Analysis = Modeling, Modeling, Modeling!**
  - **Enterprise Modeling**
  - **Data Modeling**
  - **Behavioral Modeling**
  - **Domain Modeling**
  - **NFR Modeling**

Analysis Specifics-1

- **Prototyping for requirements discovery and validation [Wasserman et al 1984 & 1986a & b, Bowers & Pycock 1993, Luqi 1992]**
- **Software safety fault isolation techniques [Leveson 1986 & 1995]**
- **Viewpoint Resolution [Leite 1987, Easterbrook 1993]**
- **Brainstorming [Gause & Weinberg 1989]**
- **Issue-based information system (IBIS) [Burgess-Yakemovic & Conklin 1990]**

Analysis Specifics-2

- **Domain modeling [many, surveys: Kang *et al* 1990, Lubars *et al* 1993]**
- **Object-orientation as a natural view [many, including: Coad & Yourdon 1991, Zucconi 1993]**
- **Non-Functional Requirements [Mylopoulos, Chung & Nixon 1992].**
- **Enterprise Modeling with Scenarios [Mylopoulos, Chung & Nixon 1992].**
- **Goal-directed RE [Dardenne, van Lamsweerde & Fickas 1993]**

Analysis Specifics-3

- **System Scoping & Bounding [Drake & Tsai 1994]**
- **Requirements Traceability [Gotel & Finkelstein 1994]**
- **Living with Logical Inconsistency of Specs [Easterbrook & Nuseibeh 1995, 1996, Hunter & Nuseibeh 1997]**
- **Ranking Requirements by Priority [Karlsson & Ryan 1997, Karlsson, Olsson & Ryan 1997]**
- **Intent Specifications [Leveson 1997]**

NLP Overview-1

The total amount of information to deal with for any real problem is HUGE and repetititititive.

We desire assistance in extracting useful information from this mass of information.

NLP Overview-2

We would like the extracted information to be

- **summarizing,**
- **meaningful, and**
- **covering.**

From 500 pages, we want 5 pages containing *all* and *only* the meaningful information in the 500 pages.

NLP Overview-3

We prefer less summarization and occasional meaningless stuff than to lose some meaningful stuff, because in any case, a human will have to read the output and at that time can filter out the meaningless stuff.

Stupidity is preferred to intelligence if the latter can lose information as a result of it not ever being perfect.

NLP Overview-4

Expressing scenarios in natural language keeps them understandable by the clients and users but runs the risk of ambiguity.

The user's manual, written in natural language, turns out to be a very good requirements specification [Berry, Daudjee, *et al* 2004].

NLP Specifics-1

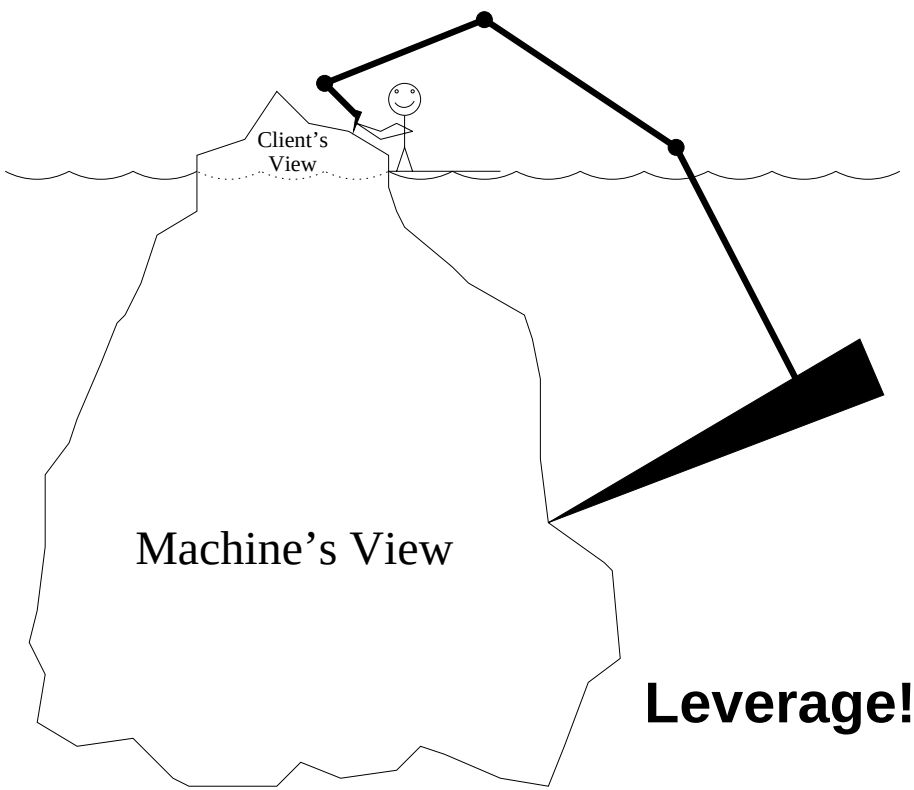
- **Restricted natural language processing of requirements ideas to get specifications [Saeki, Horai, *et al* 1987]**
- **Natural language abstraction identification with lexical affinities [Maarek 1989]**
- **Application domain lexicon building and tools [Leite & Franco 1993]**

NLP Specifics-2

- **AI-based natural language processing [Ryan 1993]**
- **Nonintelligent, fully covering natural language abstraction identification using signal processing techniques [Goldin 1994]**
- **Scenarios in Natural Languages [Somé, Dssouli, & Vaucher 1996]**

Martin Feather's View of RE Tools

The Requirements Iceberg and Various Machine-Assisted Icepicks Chipping at It



Leverage!

Tools & Environments Overview-1

Even with summarizing tools,

the amount of information that the requirements engineer must deal with is HUGE,

the number of relations between the individual items of information is $HUGE^2$, and

the number of relations between the individual relations is $HUGE^4$...

Tools & Environments Overview-2

So we want an environment filled with useful tools that help manage all the information needed to produce a requirements specification from the first conceptions, and then to be usable for the rest of the lifecycle.

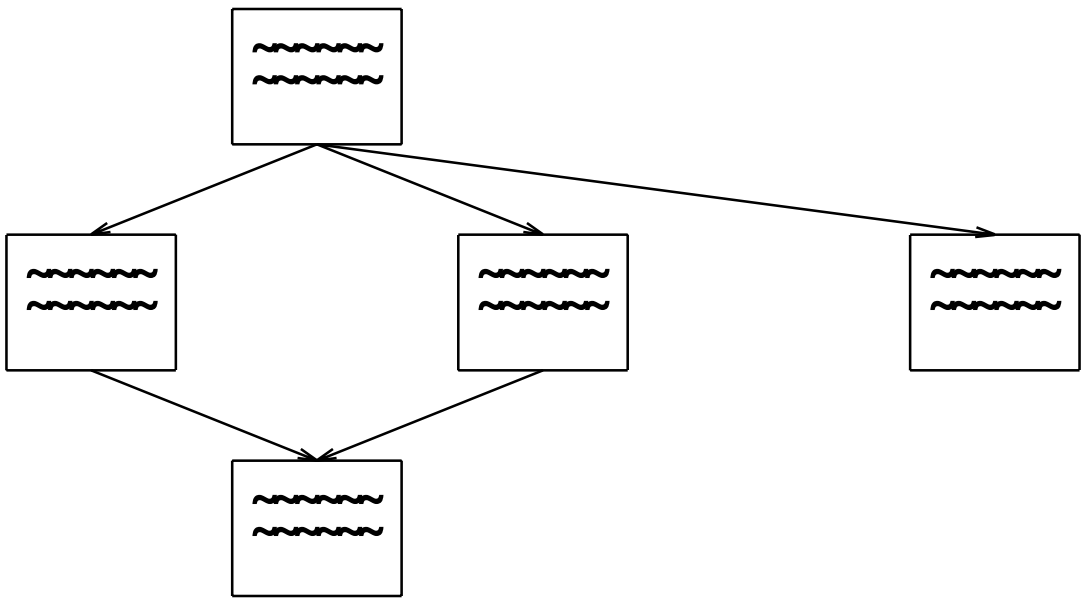
Tools & Environments Specifics-1

- **Graphical Issue-Based Information System (gIBIS) [Burgess-Yakemovic & Conklin 1990]**
- **Hypermedium as requirements engineering environments [Potts & Takahashi 1993, Kaindl 1993]**
- **Full spectrum, including traceability analysis, requirements engineering tool, READS [Smith 1993]**

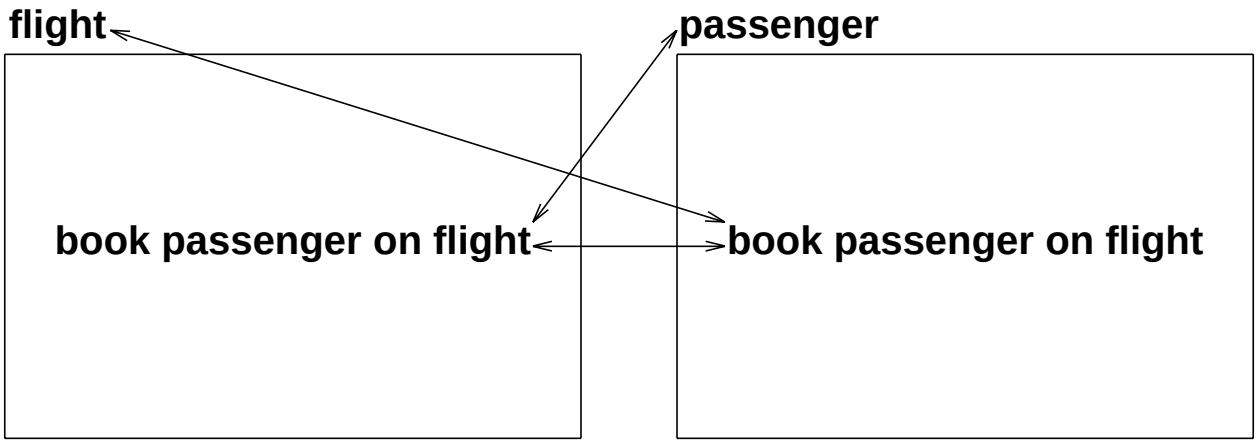
Tools & Environments Specifics-2

- **PRIME, KAOS for formal requirements modeling and analysis [Dardenne et al 1993]**
- **Multimedia hypermedium as requirements engineering environments [Wood, Christel & Stevens 1994]**
- **DOORS (Quality Systems and Software), icCONCEPT-RTM (Integrated Chipware), Requisite Pro (Rational→IBM), & other industrial requirements management platforms**

Some Views of Hypermedia Tools

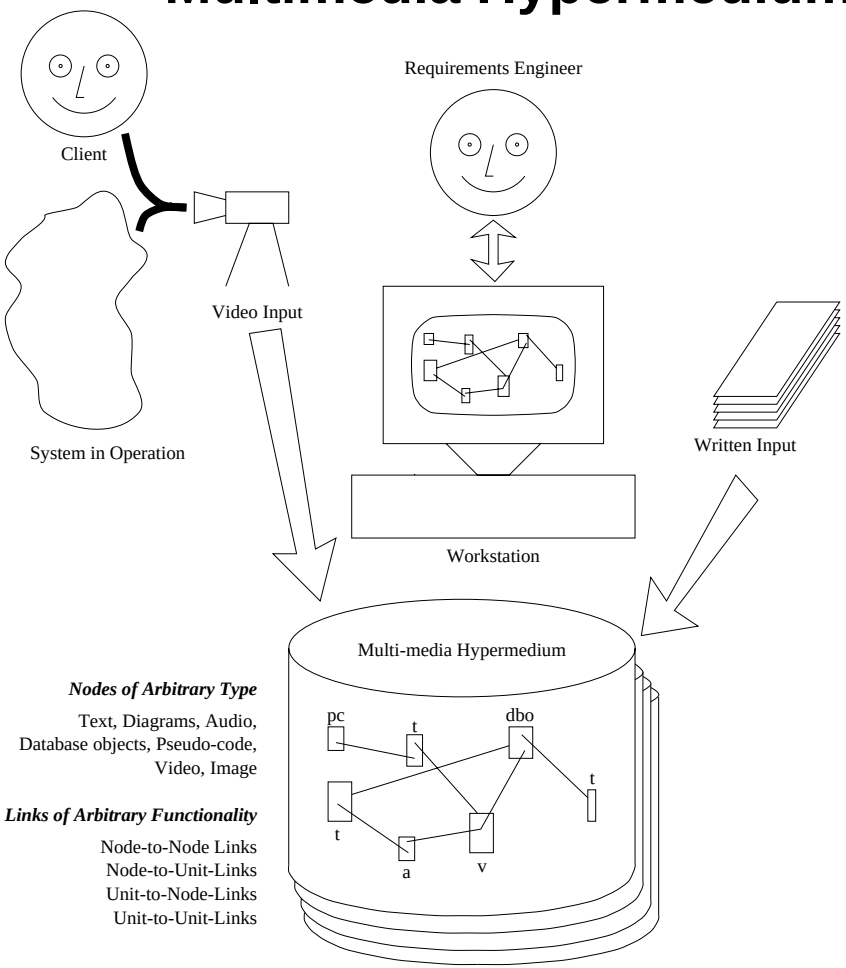


Network of Nodes


flight
passenger
book passenger on flight
book passenger on flight

Links

Multimedia Hypermedium



Nodes of Arbitrary Type

Text, Diagrams, Audio,
Database objects, Pseudo-code,
Video, Image

Links of Arbitrary Functionality

Node-to-Node Links
Node-to-Unit-Links
Unit-to-Node-Links
Unit-to-Unit-Links

Time-stamped Prior Copies of Hypermedium

Changes Overview-1

- **Change is the only thing that is permanent about software and requirements.**
- **Too many methods for RE (as for SE) assume incorrectly that the R (and the S) does not change, and these methods simply wilt under the face of the continual relentless changes that occur.**

Changes Overview-2

- **Nowadays, a growing majority of the software we write is reworked legacy code, code that is too valuable to scrap, too difficult to modify or extend without error, too expensive to rebuild, but inadequate in its current form. RE for legacy software must deal with ripple effects on requirements that are largely unknown.**

Changes Overview-3

- **Configuration management and tracing are methods for dealing with changes in software, but they work only with the total cooperation of the people involved, people who generally disdain the regimentation required to use them. This drawback applies equally if not more so to configuration management and tracing of requirements.**
- **Prototyping is also a tool for controlled evolution of software and requirements.**

Change Specifics

- **General Software Configuration and Change Management [Carter, Martin, Mayblin & Munday 1984, Tichy 1985]**
- **Prototyping [Davis 1992]**
- **Traceability [Gotel & Finkelstein 1994]**
- **Inconsistency Management [Easterbrook & Nuseibeh 1995]**
- **Change Impact Analysis [Bohner & Arnold 1996]**
- **Combating Requirements Creep [Berry 1998]**

Empirical Studies Overview-1

- **We have come to recognize that it is essential to test in actual industrial use those methods and tools that the research proposes, that it is not enough to declare that a method or tool should or must obviously work, that it is not enough to apply the method or tool to toy examples.**

Empirical Studies Overview-2

- **We have also come to recognize that it is essential to observe industrial SE and RE in order to fully understand the problems faced by the practitioners, that it is not enough to theorize what must be their problem, or to decide for them what their problems are.**

Empirical Studies Specifics

- **User Participation in RE [El Aman, Quintin & Madhavji 1996]**
- **Inspections and RE [Kantorowitz, Guttmann & Arzi 1997]**

RE as a Human Activity Overview-1

Nuseibeh and Easterbrook remind us that the context in which RE takes place is usually a human activity system.

Therefore RE draws on other disciplines for help in understanding the process.

- **Cognitive Psychology: how people describe needs**
- **Anthropology: observations of human activities**

RE as a Human Activity Overview-2

- **Sociology: political & cultural changes**
- **Linguistics: RE requires clear communication in human languages**
- **Legal documents and requirements specifications have identical requirements: must anticipate all possible eventualities and contingencies and must be unambiguous**

RE as a Human Activity Overview-3

- **Philosophy**
  - **Epistemology: stakeholder beliefs**
  - **Phenomenology: real-world observations**
  - **Ontology: agreement on objective truths**

RE as a Human Activity Specifics

- **Linguistics [Knuth, Larrabee & Roberts 1989, Dupré 1998]**
- **Cognitive Psychology [Posner 1993]**
- **Anthropology [Goguen & Jirotka 1994b, Goguen & Linde 1993]**
- **Sociology [Holtzblatt & Beyer 1995, Beyer & Holtzblatt 1998]**
- **Ambiguity [Berry, Kamsties & Krieger 2000, Berry & Kamsties 2004, Mich & Garigliano 2002, Mich 2001]**
- **Ontologies [Breitman & Leite 2003]**

Future

Lots more work is needed

Please join in!!

Recent Surveys

- **“Requirements Engineering: A Roadmap”, Nuseibeh & Easterbrook [2000]**
- **“Requirements Engineering in the Year 00: A Research Perspective”, van Lamsweerde [2000]**

Insightful Books

- ***Exploring Requirements: Quality Before Design*, Gause & Weinberg 1989**
- ***Are Your Lights On? How to Figure Out What the Problem REALLY Is*, Gause & Weinberg 1990**

Conferences and Workshops

- **International Workshop on Software Systems Design 1–10**
- **International Symposium on Requirements Engineering ’93, ’95, ’97, ’99, & ’01**
- **International Conference on Requirements Engineering ’94, ’96, ’98 & ’00**
- **International Requirements Engineering Conference ’02, ’03, ’04, & ’05**
- **IFIP WG 2.9 Software Requirements Engineering ’95, ’96, ’97, ’99, ’00, ’01, ’02, ’03, & ’04**

Journals

- ***Software Practice and Experience***
- ***Journal of Systems and Software***
- ***IEEE Software***
- ***Journal of Automated Software Engineering***
- ***Requirements Engineering Journal***
- ***ACM Interactions***
- ***Journal of Information and Software Technology***

Research Networks

- **RENOIR, Requirements Engineering Network Of International cooperating Research groups, a network of excellence sponsored by the European Union**

Web Pages

- **Requirements Engineering Newsletter (these are the files named renl*: ftp://ftp.cs.city.ac.uk/pub/requirements/**
- **Requirements Engineering Bibliography: http://www.inf.puc-rio.br/~bdbib/**
- **RENOIR: http://www.cs.ucl.ac.uk/research/renoir/**