Myths and Realities in Software Development

Daniel M. Berry

with material from F.P. Brooks's *Mythical Man Month*

I was going to say:

"Myths and Truths in Software Development"

But that would be putting too much strength to what I say.

For today's realities may turn out to be tomorrow's myths!

Truths?

"There are no universal truths except, of course, this one"

— David Thewlis

Predictions about Computers

"Computers in the future may weigh no more than 1.5 tons."
— Popular Mechanics, 1949

"I think there is a world market for maybe five computers."
— Thomas Watson, Chair IBM, 1943.

“I have travelled the length and breadth of this country and talked with the best people, and I can assure you that data processings is a fad that won’t last out the year.”
— Editor of business books
for Prentice-Hall, 1957.

“But what ... is it good for?”
— Engineer at the Advanced
Computing Systems Division,
IBM, 1968, commenting on the
microchip.

“There is no reason anyone would want a computer in their [sic] home.”
— Ken Olson, president,
Chair and founder, DEC.

Outline

Myths concerning a variety of topics:
- **Management Issues**
- **Lifecycle Models**
- **Lifecycle Steps**
  - **Conception**
  - **Requirements**
  - **Design**
  - **Programming**

  - **Testing**
  - **Maintenance**
  - **Documentation**
- **Theory**

And then, some truths!

Management Issues

- Person Month
- Divisibility of Tasks
- Human Capital
- Team Sizes
- Individual Differences
- Scheduling
- Tools and Methods
  - CASE Tools
  - No Silver Bullet
  - Rationality of Methods
  - Faking It

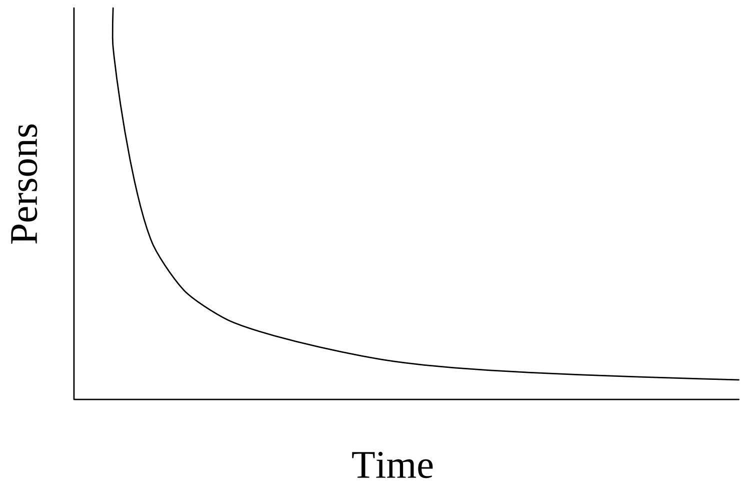
- Project Success/Failure
  - Technical Factors
  - People Factors
  - Sociological Factors
- Project Killers
- Limitations
- CMM
- Processes
- Risk Management

Myth:

Person Month:

a famous unit of measure of work (used to be called "man month")

The name of the unit is itself the myth, because it implies the following graph:



No!

Prime counter-example:

If it takes one couple 9 months to make a baby, then in how many months can 3 couples make a baby?

Nu?!?!

Three kinds of tasks

- **completely dividable**
- **partially dividable**
- **not dividable**

Completely Dividable Tasks

Laying a brick wall is completely dividable after bottom row has been laid.

You can throw as many brick layers as you want to the job.

Each works on an independent part.

The shape of the bricks and the laying pattern guarantee that the independent sections will interface properly when they meet.

There is no need for interface discussions.

In essence, the bottom row is a complete interface specification for all independent parts, no matter how many there are.

For a job requiring *P* PMs, *N* people reduces the elapsed time of the job to close to *P/N* months.

Not Dividable Tasks

Some tasks take a certain minimum amount of time, no matter how many people you throw at the task.

Examples,

Making baby
Baking a cake
Growing a tree
Learning a domain

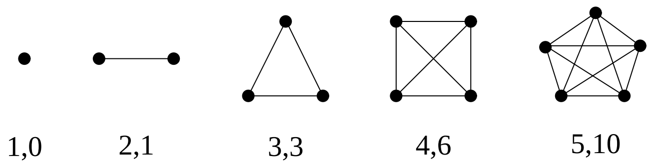
A job requiring *T* months still requires (at least) *T* months even when you have *N* people trying to help.

Partially Dividable Tasks

Most software engineering tasks are only partially dividable, because they require communication among the people over whom the tasks are distributed,

especially when interfaces must be worked out between different people’s work or when everybody’s viewpoint must be understood before proceeding with individual work.

In any such situation, the problem is the amount of communication needed.



number of persons, lines of communication

The number of lines of communication grows as the square of the number of people.

"Too many cooks spoil the pie."

Sobering Reality:

"Adding more people to a late project makes it even later!"

says Fred Brooks, leader of the IBM project to build OS/360.

The time spent catching new people up and in new lines of communication is much greater than the time the new people can add to the work.

So, for a job requiring P PMs, N people require more than P/N months, how much more depends on the communication needs and on N^2.

Human Capital, Unmasked

Tom DeMarco [in *NYT* 14 Apr. ’96] tells the following story:

Imagine, you’re the manager of crack, highly effective, highly motivated, well-knit 5-person team that is humming smoothly on schedule on this important, make-or-break project that simply *must* be done on time.

You learn, to your dismay, that one of the 5, Louise will be leaving at the end of the month.

Wotta disaster... you know you’re in deep s__t.

So you ask personnel to send you another Louise.

Personnel tells you that Louises are, sadly, out of stock, and offers you a Ralph, who is no slouch, with as much experience and skill as Louise.

So the deal is done; Louise is out on the 31st and Ralph is in on the 1st;

From accounting’s point of view, nothing has changed; Ralph’s salary, workload, expertise, etc is the same as Louise’s; so you’re trading like for like, with no net change! What’s the problem?

- **Louise spends a majority of her remaining time writing up what Ralph is supposed to know, contributing less to the project in her last days**

or

- **Ralph starts earlier over accounting’s loud complaints and Louise shows Ralph the ropes, contributing even less to the project while Ralph adds more costs to the project.**

In either case, on 1st, Ralph comes on the job alone, spends first day figuring out who is who, finding the toilets, the coffee room, and supplies, and reconfiguring the workstation left by Louise.

His contribution to the project? Zilch, Nada!

The second and subsequent days he starts poring over Louise's notes.

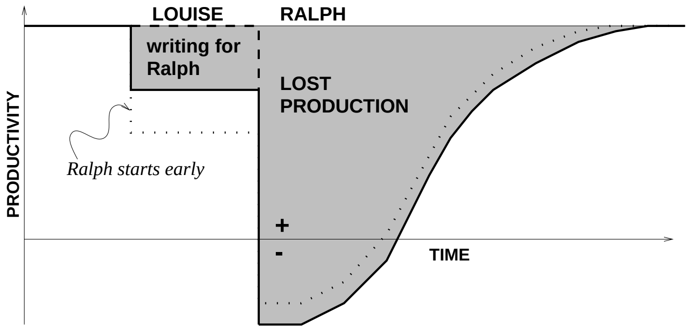
He still adds nothing to the project.

Every now and then, he has a question and goes to ask other team members, who find themselves interrupted from doing their 25% greater workload (since they have to take up slack left by Louise and not yet picked up by Ralph). Ralph's contribution? not even 0, it is negative; he keeps others from working to their capacity.

Ralph continues to be a negative for quite a while and slowly begins to get up to speed.

The graph below shows the situation.

Cost of Getting Up to Speed



In a typical large project, it can take up to two years to get up to speed. In this case, the lost production (integral above the curve) is about a person year of work.

That is, each time a new engineer is hired, a full year's salary has to be invested in that engineer before the engineer begins to pay off.

Perhaps people should be recognized as an investment and not an expense.

Myth:

Big teams are better!

We have a staff of thousands working on your program!!

Reality:

Big teams are great for 43-person Squamish, symphonies, epic movies, etc., but not for making pies and software!

“Too many cooks spoil the pie.”

Recall that the number of lines of communication grows as the square of the team size.

Speaking of Fred Brooks and OS/360, folklore has it that OS/360 was built in the IBM Army of Ants approach by a team of consisting of Brooks and 1000 nameless people, one of whom is buried in the tomb of the unknown programmer at the IBM corporate cemetery in Poughkeepsie, New York.

The first UNIX system was built by a team of three people, Dennis Ritchie, Ken Thompson, and Brian Kernighan, and none was the leader *per se*.

- **Which operating system is in voluntary use all over the world?**
- **Which operating system is used as a basis for many others?**
- **Which operating system is dissected as an object of study in text books and courses about operating systems?**
- **Which operating system was started first? Which arrived first at a relatively stable state in which new releases were for enhancements rather than bug fixes?**

Need I say any more?

Various studies indicate that the optimal team size is between 2 and 5, with 3 being the mode.

Well, certainly a team with fewer than 2 members is not a team!

With more than 5 team members, team management begins to dominate the work; i.e., each additional person costs more in team management time than he or she adds to potential work time.

Myth:

All programmers are the same.

All experienced programmers are equally skilled.

This job requires 5 person years; I've got one year to do it; so give me 5 programmers.

Reality:

Sackman, Erickson, and Grant's 1965 experiment to show that interactive programming was more effective than batch programming failed to produce significant results because the effect of the independent variable (use of interactive vs. batch submission of the job) was drowned out by individual differences in programmers of equal experience.

One experienced programmer was found to be 28 times more effective than another equally experienced programmer!

- **If you have 5 of the first kind, you'll finish.**
- **If you have 5 of the second kind, you won't!**
- **In fact, you might even be able to finish in time with only *one* of the first kind; if you can arrange the teams so that he or she will not be slowed down by having to communicate with other team members.**

The idea of Chief Programmer Teams is to build a small team around one of these super programmers

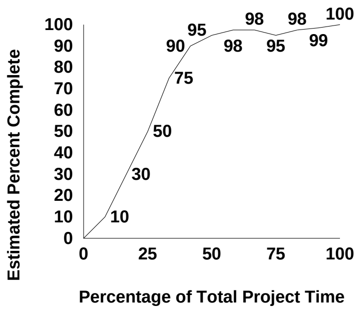
- **to allow the super programmer to do his or her stuff with a minimum of distraction by drudgery, which is done by the other team members, and**
- **to limit communication to a star configuration with the chief in the center, for which the growth in number of lines of communication is linear in the growth of team size.**

Myth:

The program is 95% done!

Reality:

Actual data from Jim Tomayko:



Do you believe “95% done” any more?

Programmers are among the most optimistic people in the world.

They continue to believe in their ability to solve problems instantly even in the face of continued, repeated evidence to the contrary.

Each is even more optimistic about him/herself than anyone else.

Myth:

CASE tools will solve all your problems

Just look at all the advertisements in the trade magazines promising 1000% improvement in software productivity if you buy the advertised CASE tools!

Reality:

Fred Brooks says:

“There’s no silver bullet!”

- **Essence**
- **Accidents**

“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**

- **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**

- However, the essence has resisted attack!

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

  and we produce the same brain-damaged software that makes the same stupid mistakes

  as 30 years ago!

More Reality:

Even the best tool will not make a good programmer out of a bad one.

In fact, a bad programmer will use good tools to turn out worse programs more quickly than ever before.

The same can be said for software development methods:

Here is an example of a so-called structured, goto-less program.

```
for i from 1 to 4 do
    case i in
        1: s1,
        2: s2,
        3: s3,
        4: s4
    esac
od
```


This is, of course, equivalent to

```
s1; s2; s3; s4
```

The so-called structured program is pretty disgusting if you ask me.

A good tool is one that automates clerical portions of tasks that you, a good programmer, do in the course of good programming.

It helps avoid stupid errors.

A stupid error is an algorithmically avoidable error!

Mainly, *you* are stupid if you let an error that a program can detect go undetected!

Example of good CASE tool: Stu Feldman's

make

relieves programmers of having to keep track of which modules depend on what others and of which ones have been updated since last compiling and linking the full program so that no more modules than need to be compiled again are compiled again.

A bunch of myths:

Programmers would like to think of themselves as rational.

Methodologists would like to believe that all programmers can be taught to be rational.

All would like to believe that rational programmers write good software!

Methodologists write papers and books describing how to use their methods to write code rationally.

All of these papers and books have examples of nice, clear step-by-step rational developments of code from requirements.

Reality

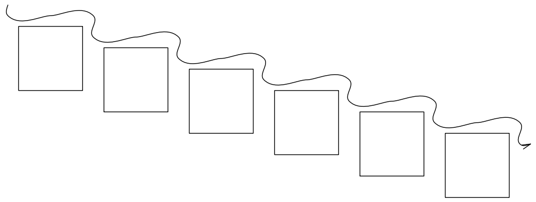
Funny thing is that these authors probably revised their examples as much of the rest of us!

I know; I have written such a monograph.

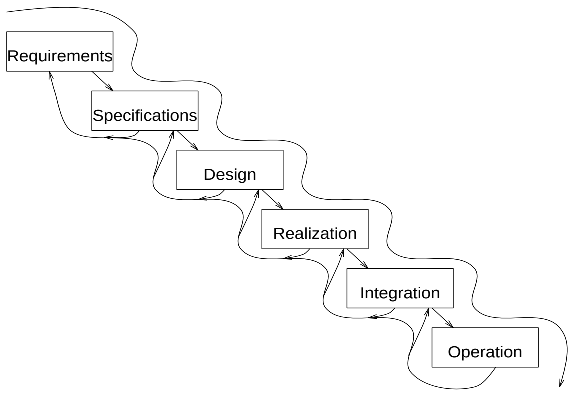
The same applies to lecturers on software engineering methods.

Myth:

Methodologists would have you believe that good programmers actually follow some variation of the waterfall lifecycle or some such.

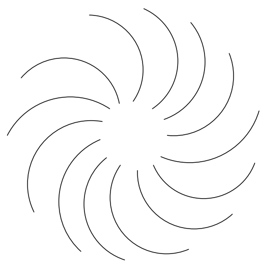


Closer to Reality:



Real Reality:

The hurricane model



In both you get wet, but a hurricane is much wetter and messier.

But in the eye of the hurricane, there is a false sense of calm.

Not Rational

OK, OK, so process is *not* rational. Nu?

But, there is value to describing the development of software as if it were rational, i.e., of faking a rational process.

"I know that I've been fakin' it!"
— Paul Simon

Faking It

David Parnas and Paul Clements suggest writing the documentation as if the development *were* rational.

Be prepared to modify it, when the development changes direction as the developers get too wet in the storm.

Myths:

The most important factors determining the success of a software development project are its

1. ***programming language and***
2. ***tools.***

Reality

All wrong, despite what language and tool designers would have you believe.

For one thing, the real influence of the listed items is in reverse order.

1. **tools and**
2. **programming language.**

A good tool can make even assembly language appear object oriented!

So then what *are* the important influences?

Certainly, more important than these is the competence of the team members.

Recall the discussion on individual differences earlier and how they completely washed out the technological difference in the programming environment.

A good programmer can write good code in *any* language, including assembly or Pascal and can fake the effect of any tool with a few good tricks and a flexible programming environment!

However, the most important factor determining a project's success or failure is something else entirely.

Tom DeMarco says,

"The very best technology never has as much impact as girlfriend or boyfriend trouble."

"... the project's sociology will be more important to eventual success and failure than the project's technology."

Bill Curtis considers

“techies [themselves to be the key] non-technological factors in software engineering,” even more important than technological factors.

In other words, the sociology and politics of the team will make or break the team.

Curtis, Krasner, and Isco found that:

“Software development tools and practices had disappointingly small effects in earlier studies, probably because they did not improve the most troublesome processes in software development.”

“Processes” refers to the management of the steps and procedures of the development.

Humphrey, Kitson, and Kasse report that:

“For low maturity organizations, technical issues almost never appear at the top of key priority issue lists, ... not because technical issues are not important but simply because so many management problems must be handled first.”

Fred Brooks says it too

“People are Everything”

- The issues are managerial, not technical.
- Every study shows the crucial importance of people.
- Projects don’t move; only goals move!

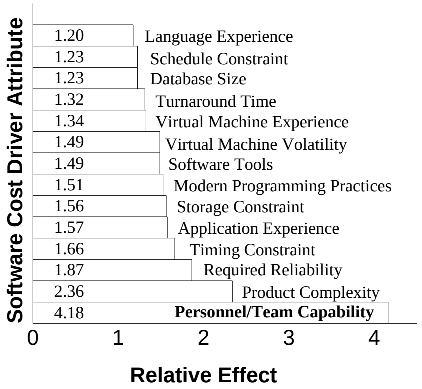
Also, Brooks, in saying that there is no silver bullet, concluded:

“The central question in how to improve the software art centers, as it always has, on people.”

Barry Boehm, who has written the book on software cost estimation says:

“Personnel attributes and human relations activities provides by far the largest source of opportunity for improving software productivity.”

Boehm’s determination of relative contribution of various factors:



Watts Humphrey, who wrote the book on software processes says:

While technology offers considerable potential for improvement, in many organizations the software process is sufficiently confused and incoherent that non-technological factors impede the effective application of technology.

In other words, ...

A team of highly competent programmers who are also highly territorial, egotistical politicians will fail while a team of equally competent programmers, who are also egoless, cooperative, team players will succeed.

Myth:

Technology is the most important factor in project success (or failure).

Reality:

Joseph Blackburn, Gary Scudder, and Luk Van Wassenhove's survey study of improving speed and productivity of software development shows that having talented people is far more important than having good technology.

Given a choice between investing in talented, expensive people and good, expensive tools, go for the talented people even though they are more expensive than the expensive technology.

They say that there *is* a silver bullet, the creative, talented, super programmer.

"Faster than a speeding silver bullet! Look up in the sky! Is it a bird? Is it a plane? No, it's super programmer!"

Project Killers

Tom DeMarco, in his ICRE ’96 Keynote, lists but a few:

- **The user hates me.**
- **One of the stakeholders is willing to come to the table only if a key precondition is met.**
- **The 3 principal stakeholders are outraged that a 4th stakeholder has been identified and invited to participate**

- **One of the stakeholders has publically declared his distrust of one of the others**
- **One of the key participants stands to be substantially disenfranchised by installation of the new system.**
- **People don’t feel safe here.**

Killer Neutralizing Skills

Tom DeMarco lists only a few:

- **interdependent decision making**
- **conflict resolution**
- **negotiation**
- **ability to apply Win-Win**

Win-Win (Theory W)

Boehm: The Win-Win approach involves the following basic steps performed by stakeholders and an architect-facilitator:

1. **Identify stakeholders’ win conditions.**
2. **Identify issues involving win condition conflicts.**
3. **Formulate and evaluate options addressing the issues.**
4. **Formulate, vote on, and adopt agreements on mutually satisfactory options.**

Among these skills, there is nothing very technical (except knowledge about options)!

Myth:

***Next year, the machine will be big enough and fast enough that we won’t have these limitations*.**

Reality:

Our ambitions, fueled by a realization of increased computing power (speed, space, and bandwidth) always exceeds the limits, not only of the computing power, but of our mental capability to deal with them as routine.

Barry Boehm said back in 1984, “There is never enough time or money to cover all the good features we would like to put into our software products. And even in these days of cheap hardware and virtual memory, our more significant software products must always operate within a world of limited computer power and main memory.”

He could have said it this year too!

Myth:

***We have a 4 rating in the CMM, so you know that we are a good software development company*.**

CMM = Capability Maturity Model [Paulk *et al*]

A model and a measure of the maturity of a software development organization's software development process.

Scale of 1 through 5 with 5 being best.

5 Optimizing	Technology Innovation Continuous Improvement
4 Managed	Measurement of Process Process & Product Quality
3 Defined	Continual Reviews Engineering Process
2 Repeatable	Planned Lifecycle Project Management
1 Initial	Chaos Heroes

5 Optimizing	Controlled Process
4 Managed	Measured Process
3 Defined	Defined Process
2 Repeatable	Basically Managed Process
1 Initial	Ad hoc Process

Predictions According to CMM

For 200K-line business data processing product:

CMM Level	Durat'n Months	Effort PMs	Faults Detect.	Faults Deliver.	$ Cost Devel.
1	29.8	593.5	1348	61	5.4M
2	18.5	143.0	328	12	1.3M
3	15.2	79.2	182	7	.73M
4	12.5	42.8	97	5	.39M
5	9.0	16.0	37	1	.15M

But ...

That is *only* a prediction.

To date, to my knowledge, there is *no* experimental verification of the prediction.

However, for better or worse, a lot of people believe in the prediction.

Reality

CMM is a good model in that *if* an organization is good *then* it will score high.

The problem is that the converse, "*if* an organization scores high *then* it is good", does not necessarily hold!

Unfortunately, the DoD and some MoDs are using CMM rating as a rating of goodness of potential contractors.

Analogy:

Many parents say:

**"If you have a fever, you may stay home from school.
If you don't have a fever, you may go on the camping trip."**

While sickness does normally lead to a high fever, fever and sickness are not logically equivalent.

Many children learn to manipulate body temperature to achieve nefarious goals.

Paradoxes about Processes

Tom DeMarco in his ICSE 18 ('96) Keynote said that our efforts to standardize and improve process have had some positive effect, ...

but not much!

In particular:

> **The organizations that have invested most heavily in methodology and process have not been the major beneficiaries.**

Paradox 1

Every time you improve process, work becomes harder.

This is a corollary of Brook’s NSB thesis that there is an irreducible core of work, not subject to improvement and not mechanizable.

So process does not help with these and just adds to the work.

We end up focusing more and more on the irreducible core as more and more of the rest gets automated, and what’s left is thus more and more uniformly hard.

Paradox 2

Focus on process tends to make an organization risk averse.

And who are the big winners?

The risk takers or the safe ones?

It’s the old

Armor vs. Mobility

argument for the military.

Armor = process and there is no room for mobility.

I am told that the process at Microsoft stinks, but they have the money to spend on being mobile and to hell with the risks! They can afford to lose a million here, a million there on bad projects.

Paradox 3

The problems of software re-use have been utterly intractable, but also our greatest success.

Just look at the great success of programming libraries!

But who can predict at planning time, which libraries will sell?

Paradox 4

We adapt to fast change but not to slow change.

So, we adapted well to microcomputers, 4th generation languages, WWW, etc., but not well to the progressive dinosaurization of data processing.

Paradox 5

Riskier projects are safer, in that you end up keeping your job!

Because the results will have a much higher value.

The highest risk projects have the biggest payoff.

And if it fails, so what? You can always find a good job!

Myth:

The new Denver International Airport (the one with the automated baggage system that was more than a year late) is a total failure!

Stay away from DIA!

(But that makes it difficult to get to good skiing in Colorado)

Reality

DeMarco, in his ICRE ’96 Keynote, observed that the failure at DIA was a lack of risk management.

They did not defend against the risk that the software might be late.

The software was on the critical path for opening and they simply did not provide any other way to open without the software.

So they were a year late.

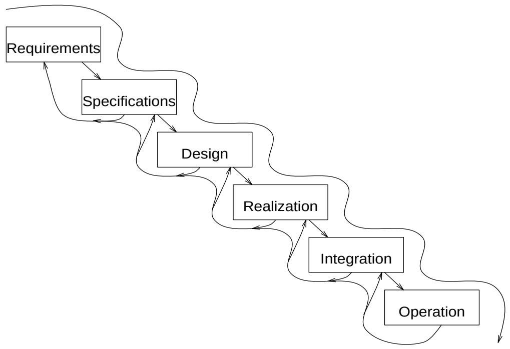
But for all the delay, for all the money lost, two years after opening, it has already recovered the losses and is making a profit.

And it’s not a bad airport (even though I *did* break my leg in the skiing I did after arriving at DIA).

Lifecycle Models

Waterfall
Problems with Waterfall
Walkerfall
Spiral

Waterfall Model:



Win Royce's Waterfall Model

Brooks about Waterfall

In ICSE '95 Keynote

Brooks 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.**
- **Even the U.S. DoD finally knows this, to wit *Defense Science Board Study*, Kaminski Committee, June 1994.**

Problems with Waterfall Model

The main overall problem is that it does not work!

No one writes software that way; no one is able to write software that way!

People make too many mistakes along the way and recognize that they have done so.

So we have to fake having followed the waterfall with the documentation.

The main problem, from the requirements point of view, of the waterfall model is the feeling it conveys of the sanctity, inviolability, and unchangeability of the requirements.

Barry Boehm produced the following picture at a 1988 workshop at SEI.

Michael Jackson Says

In the Requirements Engineering ’94 Keynote

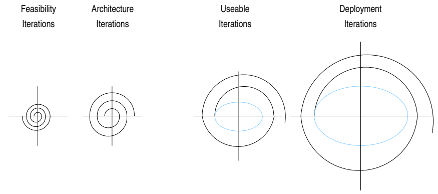
Two things are known about requirements:

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

The Walkerfall Model

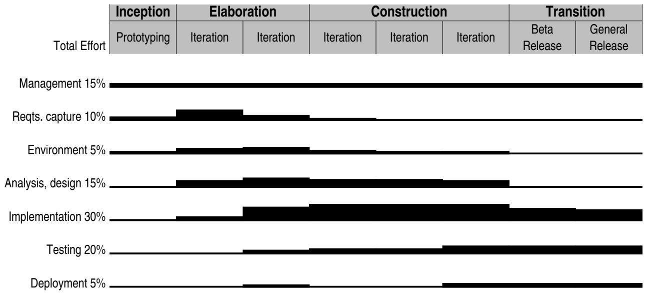
The Lifecycle Macroprocess

Development Lifecycle							
Inception	Elaboration		Construction			Transition	
Prototyping	Iteration	Iteration	Iteration	Iteration	Iteration	Beta Release	General Release



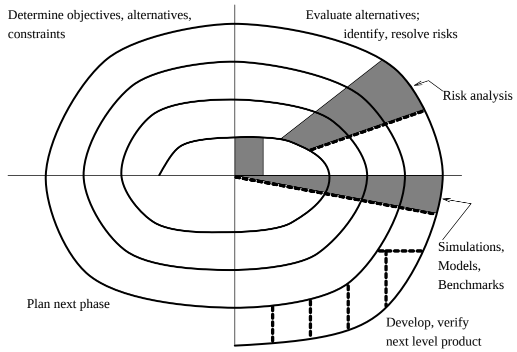
Walker Royce’s Waterfall Model, Part I

Relative Effort by Activity



Walker Royce’s Waterfall Model, Part II

Spiral Model



Barry Boehm’s Spiral Model

The spiral model solves the problem of the inappropriate sanctity of the requirements simply because it is built around ever changing requirements.

The requirements for each sweep of the spiral is derived from the requirements of the previous sweep and what was learned during the previous sweep.

The software is grown incrementally (more on this later).

Lifecycle Steps

- Conception
- Requirements
- Design
- Coding
- Testing
- Maintenance & Legacy Software
- Documentation

Conception

- Hurrying to Coding
- Costs to Repair Errors
- Crunch Mode
- Annualized Delivery
- Incremental Build
- Make vs. Buy
- Startups
- Client/Server

Myth:

You people start the coding while I go see what the customer wants.

That is, because of impending deadlines, we gotta start working on the code before we know exactly what the customer wants, and if something the customer wants is a surprise, we can always change the code later!

Reality:

There is never enough time to do it right, but there is always enough time to fix it or to do it over.

However, it always takes more time to fix it than to have done it right or to do it over (not even counting the fact that you have done it twice).

When you fix it, it is always flaky and never quite fixed (more on this later!).

What is doing it right?

It is working with client, domain experts, and technology experts until requirements are understood before designing or coding.

It is getting design worked out so that all modules are known before coding.

Why?

As requirements change, so do design and code.

As design changes, so does code (especially as a result of interface changes!).

Code is expensive to change, but design is cheaper to change, and requirements are even cheaper to change!

This does not mean not to prototype.

A throw-out prototype is written

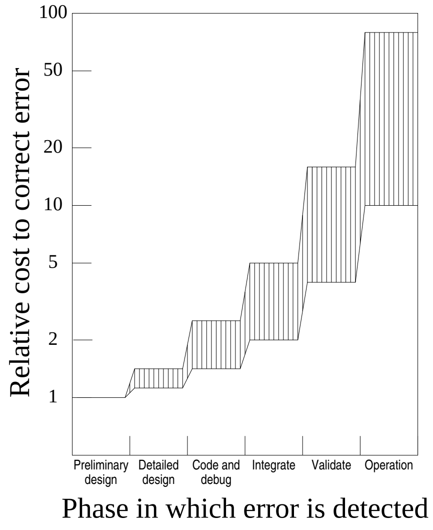
- **to help understand requirements,**
- **to get client to understand what you understand about what they said,**
- **to answer questions about user interface, algorithms, and performance.**

Remember: Boehm and others have shown that the cost to repair an error goes up dramatically as project moves towards completion and beyond ...

In graph on the next slide, note that cost scale is logarithmic, and

the graph itself looks exponential even on a logarithmic scale!!! Oy!

Graph is (*y*) relative cost to repair bug vs. (*x*) life cycle stage



Myth:

Doing it right itself is a myth and doing it wrong is reality.

Evidence shows that skimping on requirements analysis, specification, and design leads to lousy, buggy software that is brittle and very expensive to repair and enhance.

However, doing it right takes too long in these days in which the first to the market gets the whole market if the software is good enough (but not unless it is good enough!)(and who’s to define what’s good enough except the market itself!), and later, even better products are doomed to failure.

John Boddie's crunch mode does work, but you need some programmer's equivalents of Michael Jordan or Magic Johnson managed by the programming manager's equivalent of Peter Ueberroth to pull it off.

If you have talented people, and the team clicks just right, you can do it and you can even learn to manage such teams.

Richard Botting has even produced a theory showing why crunch mode works and why it has to work!

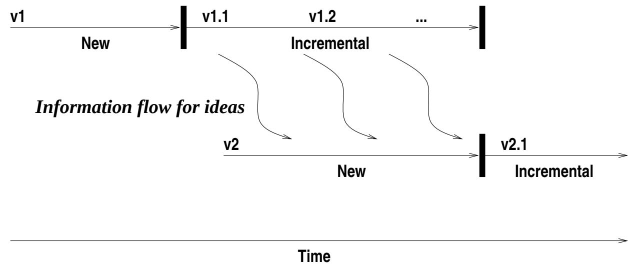
When the problems are compounded by trying to do maintenance and new development at the same time, both to keep your current customers and to meet the market demands for new features, it can be hopeless.

Steve McConnel suggests Annualized Software Delivery

You have parallel teams,

- one maintaining the current version until date *D*, and
- one aiming to release the new version on date *D*.

The current version is retired on date *D*.



Old Myth

If we try hard enough, we can get the program right the first time!

Perceived Reality

Fred Brooks says:

“*Plan* to throw one [the first one] away; you *will* anyway!”

In other words, you cannot get it right until the second time.

***This* proved to be a myth too!**

Counter-Reality

In keynote of ICSE ’95, Brooks admits that “Plan to throw one away” is wrong.

He says “Incremental Build is Better”

- **A crucial function of the designer is helping the client decide what he or she really wants**
- **The best way to decide: rapid prototyping**

Counter-Reality, cont’d

- **Growing, rather than building, software**
  - **quick to running software (You should see the effect on the morale of development team!)**
  - **early user testing**
  - **build-to-budget is possible**

Myth:

I/We can write a better X than that brain-damaged piece of ____ from ABC Co.

I/We can certainly write X for a lot less than it costs us to buy it from those jerks at ABC Co.

Reality:

Suppose X has been out *y* years and it takes you *z* years to implement X yourself, then ABC will always have *y+z* years head start in eliminating bugs and stabilizing their X.

On the assumption that ABC is charging a fair market price for X (and if they aren't, they will not be in business for long), then they are charging per copy 3 to 6 orders of magnitude less than it cost them to build it (on the assumption of selling thousands to millions of copies to recover their costs). There is no way that you are going to build X for anywhere close to what it costs you to buy it.

Furthermore, your software will always be flakier than theirs.

In general, if the product exists and is selling (and thus, the price is fair and the product is stable enough not to drive customers away), it's always a better bet to buy it rather than build it.

Furthermore, if your product W requires a functioning X, buying X, incorporating it into W, and agreeing to pay ABC a royalty for each copy of W(X) sold, allows you to get a stable W out to the market *y+z* years earlier!

Hopelessness of Startups

This all would seem to say that a startup is hopeless.

Well, it *is*! 95% of them fail—poof!

If a startup is to succeed, it needs to make something at least an order of magnitude better or an order of magnitude cheaper than what is out there; otherwise it cannot grab the market.

And you gotta make it this order of magnitude {better|cheaper} in less time than anyone else trying to make it also, in what John Boddie calls crunch mode.

Myth:

Distributed, Client/Server Systems are a major breakthrough in system architecture that will revolutionize programming and make it even easier.

Reality:

It's just another useful tool with its good points and its not so good points.

While Burton Swanson in 1996 saw a near social sweep for C/S technology, a replacement wave, Bob Glass in the same year saw deployment of C/S technology at only a 26% level with mainframes holding at 66%.

But what of the complexity of D, C/S software?

- Components of a distributed system are simpler.
- Decentralized systems as a whole are more complex.

Which dominates?

Scott Schneburger's 1997 study showed that the complexity of decentralization far outweighs the simplicity of the components.

Given that the total cost of maintenance is some 50-80% of the total cost of a system and that complexity makes maintenance harder, the D, C/S technology may end up *increasing* system costs in the long run.

Requirements

Prototyping
Requirements Difficult for Client
Requirements Volatility
Study of Requirement Errors
How Hard Are They?
Formal Methods for Requirements
Safety

More Myths:

A well-written, comprehensive requirements specification is all you need!

All of our problems would be solved if we had written a complete requirements specification.

Reality & Myth:

Interview client.

Prepare requirements specification the size of the New York City telephone directory.

Give it to the client, saying “Let me know in a week if it says what you want”.

A week later, the client says, “Yes!”

Do you believe him or her?

Reality:

Of course not!

Nonsense!

Another Way

You build a prototype and give it to the client

You walk him or her through it or you let him or her play with it for a week, saying “Let me know in a week if it does what you want”.

A week later, the client says, “Yes!”

Do you believe him or her?

Reality:

Far more likely!

Scott Gordon and Jim Bieman observe that users are more likely to be comfortable with a prototype than a specification, which can be *dull* reading and open to many differing interpretations; sample display output is more definitive. Thus, the prototype makes it easier for users to make well-informed decisions and suggestions.

Myths:

Several related myths (& more about a previous one):

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.

Reality:

According to Ruth Dameron (by e-mail):

The programmer who says these 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.

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.

Myth:

After the requirements are frozen, ...

When the customer is satisfied, ...

When the customer stops asking for changes, ...

Reality:

Poppycock!!

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

E-Type Systems

Meir Lehman identifies concept of E-type system.

It is a system that solves a problem or implements an application in some *real world* domain.

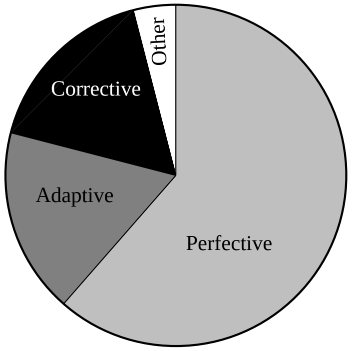
Once installed, an E-type system becomes inextricably part of the application domain, so that it ends up altering its own requirements.

- **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!**

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?

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



More Realities:

Martin & Tsai's study of requirement errors:

They conducted an experiment to identify lifecycle stages in which requirement errors are found.

An experienced user produced a polished 10-page requirements document for a centralized railroad traffic controller.

Ten 4-person teams of software engineers were given the requirements document in order to find errors in it.

The user believed that the teams would find only 1 or 2 errors.

92 errors, some very serious, were found!

The average team found only 35.5 errors, i.e., it left 56.5 to be found downstream!

Many errors were found by *only* one team!

The errors of greatest severity were found by the fewest teams!

CONCLUSIONS: Requirements are *hard* to get right!

How hard are they?

Most errors are introduced during requirements specification!

Boehm: at TRW, 54% of all errors were detected after coding and unit test; and, 65-85% of these errors were allocatable to the requirements, design, and documentation stages rather than the coding stage, which accounted for only 25% of the errors.

So most errors either are *required* or are the unplanned result of situations that are not even mentioned in the requirements specifications.

So how *do* you find the requirements?

- **Interview**
- **Observe**
- **Become a user**
- **Use imagination**
- **Prototype**
- **Validate all that you think you learn**
- **Accept that you will *not* find everything!**

Myth:

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

Reality

Yes, formal specifications 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!

Formal methods do *not* find all gaps in understanding!

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

just as they are in programs!

von Neumann and Morgenstern (*Theory of Games*) 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.

Myth:

Replace that ancient, slow, creaky electro-mechanical system that can wear out with sleek, fast, whiz-bang software software that can never wear out.

Reality:

Sure, it won't wear out, but the questions are, "Will it even be correct?" and "If it is correct at all, will it be correct at all times?"

"If it ain't broke, don't fix it!"

Therac 25 Disaster

Between June 1985 and January 1987, the computer-controlled radiation therapy machine Therac-25 massively overdosed 6 people, all of whom developed severed radiation sickness and all but 1 of whom has died (as of 1994). It was the worst accident in the history of radiation therapy machines.

A study by Nancy Leveson showed that earlier machines, the Therac-6 and Therac-20, were controlled by computer, but the computer was added after the machines had been available with electromechanical (EM) controls. In particular, the safety controls were still EM even after the addition of the computer.

In the Therac-25, designed from the start with computer control, more of the control, including the maintenance of safety, was given to the computer.

Software checks were substituted for many of the traditional hardware interlocks.

Nominally, this was a good plan; they reused code that appeared to be reliable.

The problem was that the Therac-20 was a reliable *system* !

The original Therac-20 software had a bug that just never showed up because the independent hardware interlocks prevented overdoses.

When they programmed the new checks into this buggy code, and they happened to never duplicate the error causing situation in the tests, the old bug was never discovered and reared its ugly head later with fatal results.

After much denial and protestation that the overdose was impossible, the manufacturer was forced to put the independent hardware interlocks back into the machine, just to be sure, even after they had found and fixed the bugs.

Design

Extending Prototype Reuse

Myth:

The prototype can always be extended.

Reality:

Extending a prototype is a good way to preserve lousy design decisions.

Advice:

Build the prototype in a language like LISP or Prolog so that temptation to turn it into production version is reduced!

Myth:

Reuse will save money and increase software reliability

After all, a reused module does not have to be designed, developed, inspected, and tested again. Moreover the fact that the reused module has been out there under daily use means that it is far more tested, debugged, stable, and reliable than any newly written module for the same functionality.

Yes, the module designed for reuse costs about twice what the same module designed for only one-time use. However, you amortize this extra cost by reusing the module dozens, hundreds, or thousands of times.

Note that there are many different kinds of reuse:

- **specification**
- **design**
- **source code**
- **object code**
- **macro**
- **procedure**
- **class/abstract data type**

and there are many means of customization to specific requirements.

- **parameter passing**
- **parameter passing**
- **white box modifications**
- **black box encapsulation**

Reality:

Neil Maiden and Alistair Sutcliffe point out that the human issues in reuse confound the advantages.

Finding opportunities for reuse, identifying candidate reusable components, adapting them to the current requirements, etc. may cost more than just developing from scratch.

A slight mismatch between the new requirements and the reused module’s specification may make more reliability problems than exists in immature software designed for the specific occasion.

Identifying the right reusable components, comprehending them, customizing them, all necessary for successful reuse, is a lot harder than thought.

A 1991 IBM study by Kruzela on software maintenance found that 50% of all maintenance time is spent in just understanding the code to be changed; this is what makes maintenance so much more expensive per line than writing new code.

According to Thompson and Huff, understanding unfamiliar software is complex and error prone.

How well it is done by a given programmer is a function of both skill and luck, and individual differences will come to play here.

However

However, as Ivar Jacobson, Marty Griss, and Patrik Jonsson point out, there are circumstances in which reuse works and pays well.

In an organization producing a family of related products, a carefully planned internal reuse business helps to tame the pandemonium that usually results as the organization tries

- to maintain and enhance a variety of existing related products and
- to introduce new products in this family rapidly, to stay ahead of the competition

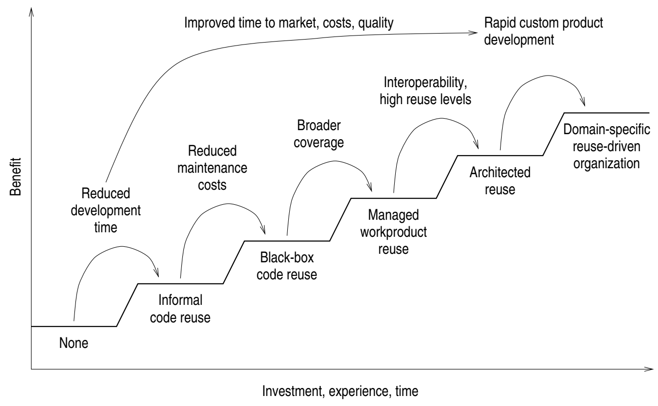
all running on a variety of platforms.

In such a situation, large portions of the code in all versions, both revisions and variations, of the products in the family are very similar, with differences that are minor in size but major in importance.

When fully functioning, the reuse business encourages developers

- to build adaptable modules for all functions and
- to use and adapt already developed modules when producing
  - new versions of existing products and
  - new products.

The reuse maturity model below summarizes the way such a reuse business works.



Such reuse businesses avoid the problems mentioned above because

- the modules subject to reuse are all for products in a single family of similar products
- the modules are designed from the beginning with reuse in mind

Much of the mystery about which modules to reuse and how to adapt them is gone.

Coding

Coding Costs
Optimization
Object Orientation

Myth:

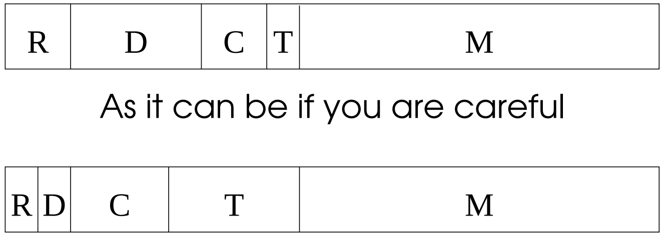
Coding is expensive.

Coding is a major part of the lifecycle.

Coding is the resource eater.

Reality:

EXPENDITURE DISTRIBUTION OVER LIFE-CYLE



As it often is

R: Requirements Definition D: Design T: Testing
C: Coding M: Maintenance

Even if you're not doing it right (i.e., are doing the top), coding is bubkes compared to requirement specification and design or testing.

What is the implication of all this?

For one thing, methods and tools directed at only the coding phase cannot have a major impact on the lifecycle.

Myth:

We gotta optimize it!

We gotta squeeze every last nanosecond out of it!

We gotta squeeze every last byte out of it!

Reality:

Today's computers are sitting there twiddling their thumbs > 95% of the time, even if the human users think they are using them all the time.

For most software, efficiency just does not matter.

The response time bottle neck is time to move cursor's image on screen to the next line or to draw the characters being displayed.

Optimization?

First Rule (Don Knuth):

DON'T!

With today's machine speeds and costs, the savings in machine time and its cost can never equal the cost of the programmer's time to optimize.

With today's underutilization of machines and for most software, there's no real need.

The lost in clarity of program and increase in difficulty to maintain are not worth it!

However, sometimes you must optimize (sigh!),

- **to fit in machine,**
- **to meet hard real-time constraints,**
- **to beat performance of competitor in commercial software.**

Second Rule (Knuth):

NOT YET!

In any case, it's useless to optimize before you have a running program.

Only then can you know what parts need optimization and, thus, where it will pay off.

80% of the execution time is spent in 20% of the code.

If you knock a microsecond off something that is executed only once, you've gained nothing, and maybe you've introduced an error!

If you knock a microsecond off something that is executed 2,000,000 times, you're really saving time!

You need to have a running, instrumented program to determine where to optimize.

Third Rule (Berry):

Very Carefully ...

from a known correct program

Hide the optimizations in as small a module as possible.

It is often better not even to optimize; find a better algorithm-data-structure combination.

Last Rule (Berry):

DON'T!

C++ is like teenage sex

It is on everyone's mind all the time.
Everyone talks about it all the time.
Everyone thinks everyone else is doing it.
Almost no one is really doing it.
The few that are doing it are:

- **doing it poorly,**
- **sure it will be better next time, and**
- **not practicing it safely.**

— Graffiti found in a toilet stall in the Faculty of Computer Science, Technion, November, 1993

C++ and teenage sex

It should say, "Programming in C++ is like teenage sex".

And even *that's* wrong!

It should really say, "Object-oriented programming in C++ is like teenage sex".

It is really inappropriate to equate OOP and C++; you can do OOP without C++, *and* you don't need to do OOP when you use C++.

Testing

Bugs
Bug Frequency
Regression Testing
Testing and Reviews
- **Walkthroughs**
- **Inspection**

Time for Inspections
Growth of Bugs
Bugs & Reliability
N-Version Programming

Myth:

Bug

The word itself is a myth

It implies that a bug is something that an otherwise healthy program gets

Maybe as a result of contagion from sitting in the same memory with other buggy programs?!

Reality:

The bug that shows up after delivery to the client was probably *required* into the software, with probability of about 60%, according to data by Boehm and others.

Myth:

"Whew! that was the last bug!"

"We found the last bug! now we'll fix it!"

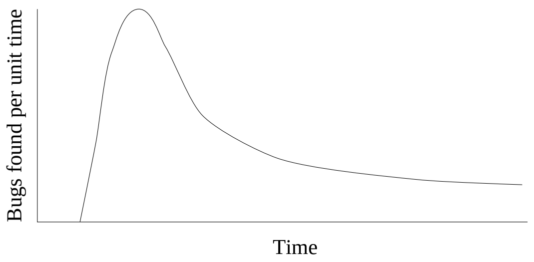
Reality:

Mills says:

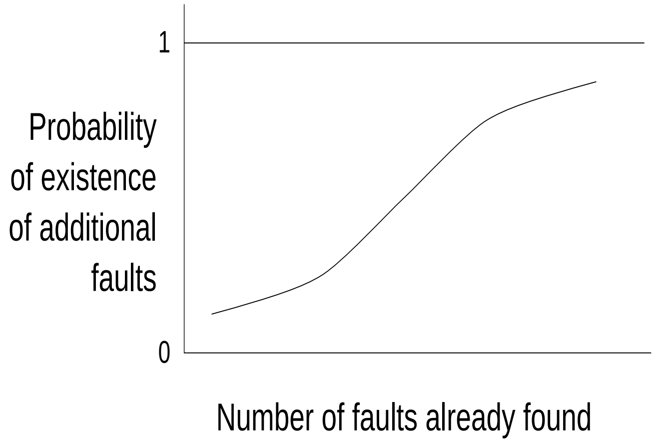
"The best way to know that you have found the last bug is never to find the first bug."

Any bug, even a tiny one, is a sign of sloppiness or lack of understanding in development, and sloppy development or lack of understanding leads to lots of bugs, never just one.

Studies by Myers and others have shown the following bug arrival graph over the testing of one release.



That is, they tend to arrive in bunches if they arrive at all.



More Reality:

Dijkstra says:

"Testing can be used to show the presence of errors, never their absence!"

This quote suggests a certain mind set for testers.

They should be trying to find errors rather than trying to show that there are none.

Goals have a bad habit of turning into reality!

Myth:

After an error is found in an unexpected place:

But, I tested that part before and didn't touch it for this new change!

Reality:

All parts of a program are connected to all other parts, *even* if you don't think so, especially if you have pointers (and who does not?).

This is why experienced testers re-run all previously run test cases whenever a new version of a program is produced.

There is even a name for this kind of testing, *Regression Testing*.

Having a program automatically running test cases and comparing actual to expected output helps a lot.

Myth:

Let's test it thoroughly.

Testing will find the problems.

We'll find the bugs later when we test it.

Testing is running a program against predetermined data for the purpose of detecting a difference between the program's output and the expected output for the purpose of finding errors in the program.

Test cases and expected outputs are determined

- **according to the specifications, exercising every feature, option, etc.**
- **according to the program structure, exercising every statement, path, etc.**

Thoroughly testing a program is impossible (requires unbounded number of test cases); so try to choose test cases that will expose *all* errors.

That's very difficult, especially since we do not know what all the errors are, and if we did, we would not need the test cases!

Reality:

A number of studies have shown testing not very effective at finding bugs.

They have compared errors located by various methods to all errors ever reported for a product over its lifetime:

- Inspections found 67 – 82% of them.
- Walkthroughs found around 38% fewer of them than inspections.
- Traditional testing found 15 – 50% of them.

The studies are described and summarized in two good books on software inspection, one by Gilb and Graham and the other by Ebenau and Strauss.

Walkthrough:

Author of a module describes the inner workings of the module to a group of people, generally from the same project, for them to spot internal errors and inconsistencies as well as interface problems with their own modules.

Inspection:

Walkthrough +

- Reviewers are given the module at least a day before meeting and are told to review the module privately ahead of time.
- Meeting moderated by facilitator.
- Minutes of meeting captured by recorder.
- Meet for exactly two hours.

- Goal of meeting is to find as many problems as possible.
- No time is wasted at meeting to consider solutions.
- Author finds solutions later and submits module again for another inspection.

Why are walkthroughs and inspections more effective?

Probably because the walkers through and the inspectors are humans who are capable of using their keppeles (noggins) to think.

When was the last time you saw a test case think?

Ah, but why cannot the human running the test cases think? He or she can, but generally there are too many test cases, very purposely to get maximum coverage, to allow time for careful thinking about the implications of each.

Also many times, the test cases are run by a program that runs each case and compares the output with the expected output, reporting only deviations; this driver does not think, and humans have fewer opportunities to think about test cases in general.

Note that the difference in the effectiveness of walkthroughs and inspections is probably due to the formality difference.

Still Pretty Bad

As good as inspections are, the situation leaves a bit to be desired.

Even with inspections, the data show that at least 18% of the bugs found in a program over its lifetime will be found by the customers and users

And of course, you know what kind of wonders this does for your company's reputation!

Myth:

We don't have time to do inspections now on* every *step!

We gotta finish the code sooner so we can test it sooner.

Reality:

If you don't have time to do the inspections then you don't have time to finish the project satisfactorily, period.

- **In any case, an error that is in the software now is not going to disappear by itself; you have to find it first.**
- **If you don't find it now, then you may or may not find it later before shipping.**

- **If you don't find it before shipping, then the customer *will* find it.**
- **In any case, finding it later means that it costs (time, money, whatever) 10 to 100 times to fix as finding it now.**
- **Finding it now means the least cost (time, money, whatever) to fix it.**
- **So if you don't have time to inspect, find it, and fix it now, then you certainly will not have time later, when it will take longer to fix it.**

Myth:

The next release will be better!

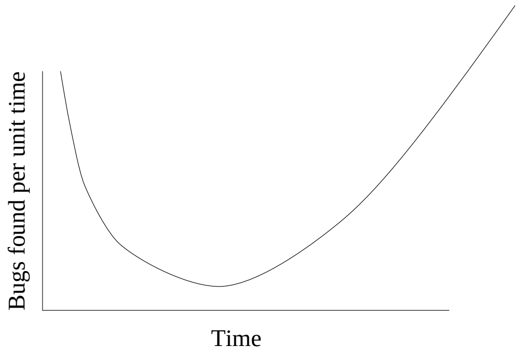
The bug will be fixed in the next release!

When the product is finally bug-free ...

Ha!

Reality:

The famous Belady-Lehman graph of bugs found over all releases of a product:



This curve is the result of a theory that attempts to explain a well-known observed phenomenon of eventual unbounded growth of errors in software that was being continually fixed, i.e., a general decay in modified software.

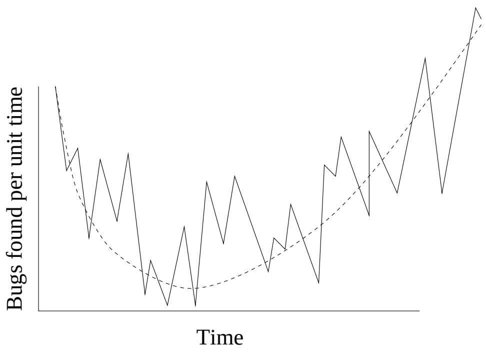
The theory assumes an $\varepsilon > 0$ probability of introducing more than one error when correcting one error.

This suggests that at a certain point, either:

- declare all current and remaining bugs to be features (Knuth has decreed that upon his death, all remaining bugs in TEX and METAFONT are features!)
- start all over with a new program development or maybe reengineer the software.

More Reality:

The real-life graph is not as smooth as the theoretically derived graph.



We will not be able to identify the real min point until we are well past it.

So this means we must keep the sources of all releases, with a configuration management system, so we can roll back to to the best release, some number of releases ago.

Myth:

The program has to be perfect.

We have to squeeze out* all *bugs.

Reality:

Reliable computations are obtainable from buggy programs, which after all, are the *only* kind of programs there are!

David Parnas observed:

I can build a reliable system with thousands of bugs, if you let me choose my bugs carefully.

Working around bugs is possible and is often required.

Both users and programs evolve to allow users to get trustable computations from programs despite bugs.

Simply, users learn not to use the bug that is, in effect, a feature (unless of course they want that feature!).

Myth:

The number of defects is a good predictor of reliability:

> ***The fewer the defects the greater the reliability***

Hence some measure defects during development as evidence of code reliability and to calculate the likely reliability.

Reality:

It’s mostly right, as we observed before, but..

Shari Lawrence Pfleeger, Ross Jeffrey, Bill Curtis, and Barbara Kitchenham caution not to count all defects the same way.

They report that Ed Adams of IBM found that 80% of the reliability problems are caused by only 2% of the defects.

We need to examine the defects carefully to see which cause reliability problems.

Remember what David Parnas says:

“I can build a reliable system with thousands of bugs, if you let me choose my bugs carefully.”

Myth:

A standard approach to hardware fault-tolerance, to have multiple versions of a unit, is a great approach to software reliability.

N-version programming will increase your software’s reliability.

Reality:

N-version programming is a good way to spend *N* times the original costs while getting no more reliability and possibly even less.

Hardware

In hardware, the problem is decaying components.

Executing multiple copies of the unit concurrently, voting, and using the majority (say 2 out of 3) result is a good way to guard against unit failure.

But note what is being protected against:

physical component failure
not
design errors

If none of the units have failed physically, then if there is a design error that causes an erroneous result on one of them, then all of them will have the same result.

The approach works because relative to the computational speed, the time until unit failure is independent of that of all other units.

Software

Assuming no failure of the underlying hardware, all copies of the same program will always produce the same result, even when they are committing an error.

Therefore, *N* voting copies of a program will always choose unanimously and obviously has the same reliability as the program itself.

For software, the reliability problem is that of design errors; programs do not wear out; that is a program presented with the same input will always produce the same result no matter how many times the program has been run and how long it has been sitting in the memory unused.

In other words, multiple identical software units are no more a protection from design errors than are multiple identical hardware units.

If *N* identical software units are useless, then perhaps *N* independently developed programs with the same intended functionality will help.

The theory shows, but it should be clear too, that this approach to protecting against design errors depends on the errors that each program can make being independent of those of every other program.

Experiments by Nancy Leveson and John Knight show that this independence assumption does not hold, that in fact, different programmers working from the same specification tend to make the same errors.

In fact *every* experiment with the N-version approach to software fault tolerance has found that independently written software routines do not fail in statistically independent ways.

People tend to make the same mistakes in the same harder parts of the problem, with the essentially the same well-known best algorithm and in the same boundary cases of the input space.

Any shared specification can lead to common failures and testers omitting the same nominal and unusual cases that the developers overlooked.

Consequently a majority of so-called independent programs might indeed vote for the same erroneous output.

Ah! so have the programmers program from independently written specifications.

But then, it is not even clear that they programs will be implementing the same requirements! We all know how hard it is to get requirements right.

Moreover, even when programming from the same requirements, there are situations in which *N* so-called independently developed programs will be less reliable than any one.

The first experiments in N-version programming were conducted in the early 1980s in my SE class at UCLA with a *simple* 5-command formatting program.

Different programs had exactly the same list of words on each line, but with different distributions of extra white space between the words on any given line.

A human would not regard these lines as different, but the original voting program did regard them as different and in fact produced fewer correct results than any one program.

Many times no majority could be found at all!

Ah! So change the definition of agreement!

But then the voting program begins to be more complex and its own reliability begins to be an issue.

The conclusion: There are far better and cheaper approaches to improving reliability than writing the same program *N* times, e.g., inspection which typically costs 15% more.

Maintenance & Legacy Software

- **Pervasiveness**
- **Dominance**

Myth:

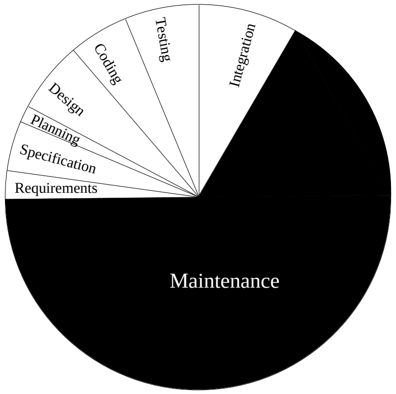
Design/coding/development **is where the** ***action/excitement/money*** **is!**

Most programmers develop new code.

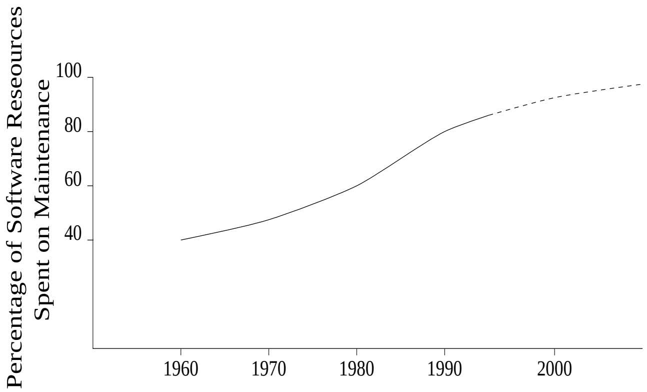
Most of a programmer's work is developing new code.

Reality:

In 1980s,



Per David Parnas, 1994



We are heading to the day in which nearly all of our work will be modifying existing software.

Even today, a new field is springing up, that of Legacy Software, existing software that is

- **too valuable to scrap,**
- **too difficult to modify or extend without error,**
- **too expensive to rebuild, but**
- **inadequate in its current form.**

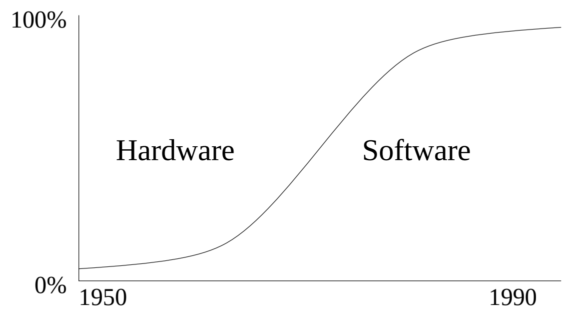
Jim Horning observed:

Hardware is the part you can replace.

Software is the part you have to keep,

because it's so expensive and nobody understands it any more!

Relative Hardware-Software Costs



The last two realities suggest that the key to keeping software costs down is to write code that is easily modified.

Choose programming methods whose guiding concern is making *inevitable* modifications easier.

For example, Parnas's information hiding method tries to decompose a system into modules such that whenever the implementation of one concept is changed, only the one module implementing that concept is affected!

The Y2K Problem

You've all heard of the famous Y2K problem, the Year 2000 Problem?

Myth:

The problem with the Y2K is that all of the data structures are big enough for only the last two digits of the year. Therefore, come the year 2000, we're going to need more digits or else 2000 will look smaller than 1999.

Reality:

Given the cheap cost of memory these days, the space problem is minor, compared to the problem of having to suddenly work with different algorithms that take into account the changes, i.e.,

- **00 is really more than 99,**

or

- **some dates are 4 digits and others are 2,**

or

- **...**

More Reality:

The different algorithms problem is minor compared to the fact that everything will have to be recompiled with new data structure definitions and new date calculation algorithms.

Even More Sobering Reality:

All of the space problems and all of the funny new algorithms and all of the recompilation required are bubkes (zilch, nada, rien, nichts, klum) compared to the REAL problem with the Y2K...

What is it??????

The REAL problem:

The REAL problem is that all of the code to deal with dates is scattered all over all of the legacy programs, very often not identified by comment and very often disguised as arithmetic or shifting or other obscurities.

Compared to finding *all* the code to change, and you have to find it *all* or else the program bombs on 1 January 2000, all the other problems are a piece of cake, a puzzle designed for 2 year olds, putting a round peg into a round hole, etc. etc.

Solutions

All solutions that attack only the data structure and algorithm problems are doomed to repeat history.

No matter what size you set the data to be, there will be a maximum representable date, although maybe very far into the future, and at that date, *X* you will have a Y*X* problem.

The only REAL solution is one that attacks the REAL problem.

- **Build an encapsulated *Date* abstraction.**
- **Hide the data structure and algorithm inside of it.**
- **Rewrite all the code, replacing code that uses the date data structure directly by calls to proceudres of the *Date* abstraction.**

You still pay for the data structure and algorithm changes, the recompilation, and the finding of all date accessing code...

BUT you never again face the REAL problem.

At Y*X*, you change the inside of the *Date* abstraction and recompile; you do not have to *find* the using code again.

That is, we were pretty stupid about the dates once, but we will have learned from our mistake not to repeat history.

Documentation

Myth:

We'll document that later when we have time. Now, we gotta finish up the coding to meet the deadline.

We'll update the documentation next Tuesday after we get this module out of the way.

Reality:

Documentation that is put off never seems to get written!

Either,

- **there never is enough time to write it, or**
- **when there is enough time to write it, we've forgotten what we wanted to say**

Guindon, Krasner, and Curtis have studied designers and have concluded:

Many design breakdowns occur because of cognitive limitations:

- Designers forget to return to design goals they have postponed.
- While working on one part they cannot record opportunistic design ideas that affect another part.

Sound familiar?

Likewise with documentation!

Also, if one is documenting during programming, these breakdowns are less likely.

“The job is not done until the paper work is done”

The problem with documentation is that it is perceived as a necessary evil which is done only after the fact

So it must be done during the programming!

Remember, there never is enough time to do it right!

But it is not going to be done during programming unless there is an incentive to do so.

Also any technological or managerial scheme to force documentation can be subverted by unwilling programmers.

Theory

Mathematics vs. Programming
Knuth Discovers
Mushiness
NP-Completeness

Myth:

Mathematics is harder than programming.

Proving theorems is harder than programming.

The true sign of intellectual achievement is being able to prove theorems, not programming; programming is trivial.

Reality:

Each new program is a new formal system, modeling a real-world system, and this formal system is built from the ground (or library) up!

Each program is a formal system in the sense that one can reason logically about its behavior.

The data of a program form a model of the relevant real-world domain, and the operations of the program transform this model according to a related model of real-world transformations.

Correspondences:

Basic Statements	Axioms
Constructors	Rules
Data	Definitions
Invariants	
Statements	Theorems
Functions	

So, programming is at least as difficult as developing a mathematical theory.

But, not all programs build a new theory!

True, but not all mathematics is new either!

Actually, programming is harder in several very important senses!

First, consider the audience of the work:

Theorems People
- UKWIM works
- Can accept imprecision

Programs Computers
- UKWIM does not work
- DWIM does not work
- Cannot accept imprecision

Mathematicians will make simplifying assumptions to keep the math tractable; the goal is usually good math, not always modeling reality, e.g., Euclidean geometry.

Software cannot make simplifying assumptions that can cause deviations from reality that invalidate functionality, e.g., in process control of fast and dangerous situations.

Therefore, program models tend to be more complex than models that mathematicians study.

The notions of correctness in mathematics and programs are different.

A mathematical model must be consistent; it need not match reality (be correct), and it need not be complete (in the formal sense).

A program model must be consistent; it must match reality; and it must be complete (in the sense that it reacts gracefully to *all* inputs).

For example,

- $\sqrt{}$ is not defined for input < 0
- *sqrt* must deal with *all* input

Social processes for mathematics and programs are different.

Theorems written by mathematicians for publication

- undergo scrutiny of other interested mathematicians,
- are interesting; otherwise, why bother?

As a result, errors are found and corrected.

Programs

- are not looked at by other programmers,
- are boring.

Therefore, there are no error-finding social processes.

Program proofs verify only consistency with specification and not correctness, and are meaningless if the specification is not what is intended.

Testing shows only the presence of errors and not their absence.

Therefore, it is much harder to ensure the correctness of programs than theorems.

Note that a key point of inspections is to try to introduce to programming the social processes that are so successful at finding errors in theorems.

Knuth and Software

Donald E. Knuth, one of the premier computer *scientists* in the world, who has done such mathematically respectable work as:

- **Member of Algol 60 Committee**
- **Attribute Grammars**
- **3-Volume Encyclopedia on Algorithms**
- **Knuth-Bendix Algorithm**

has spent ten years of his life developing two major programs for document typesetting, TEX and METAFONT.

These projects started out as a *brief* attempt to make sure that all of his subsequent books, to be printed with computer-driven typesetters, would look as good as his earlier hand-typeset books.

They *mushroomed* into 10-year efforts yielding, to date, two releases of each.

I recall Knuth's giving a lecture at UCLA, about one year after starting, in which he said that he had hoped to have TEX fully running by tonight—as if another day or so would have cracked the problem that was preventing it from running!

It took him another 10 years to finish, and he still is not *finished*.

Knuth published a paper in 1989, *The Errors of TEX*, describing this effort and listing the 867 errors found by users in the 14,000 line Pascal program.

Knuth gave a keynote address at IFIP ’89 “Theory and Practice”.

He explained that one of the lessons learned from the development of his typesetting software is that “software is hard.”

What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD.... From now on I shall have significantly greater respect for every successful software tool that I encounter.

During the past decade I was surprised to learn that the writing of programs for TEX and for METAFONT proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks.

The remark that writing software is more difficult and requires greater accuracy than proving theorems merits close examination.

The difference is the audience.

The programmer is writing for an incredibly stupid and literal audience, the computer, that cannot tolerate minor incompleteness.

The mathematician is writing for a highly intelligent audience, the professional mathematicians, that can be counted on to fill in on missing or wrong details.

But note that for Knuth, proving theorems is hard too.

He, above all, knows how easily published theorems contain, plainly and simply, mistakes.

Knuth has had to publish two consecutive corrections to a published proof of a theorem about attribute grammars.

Myth:

Software engineering (as an academic discipline) is so soft and mushy. How can you work in it?

Reality:

Bill Curtis says

“Whenever I discuss human issues in software engineering someone always says, ‘Right, that’s the soft side of computer science.’ ... Nevertheless, with all the allusions to soft, mushy material, hard programs continue to be written by humans and not by machines, expert systems, automatic program generators, and other objects worthy of federal funding.”

It was computer scientists that observed that there is a serious problem in software development. To my mind, if we have a problem that we have to solve, we gotta go where the problem takes us and explore whatever solutions may work. If those solutions are non-technical and not based on theory, so be it.

The fact is that for nearly 30 years now, we have attempted to solve the problem with technology, theory, etc. These have made some advances, but the problems remain and seem to have gotten worse, in some sense (perhaps due to ambition fueled by successes). Recently, we have begun to recognize that the problems will not be cracked unless we also consider non-technical issues.

Note word “also”.

Myth:

The problem is NP-complete! Therefore,

- ***we can give only small inputs to the program (for the problem),***
- ***we have to use heuristic methods,***
- ***we have to accept only an approximate solution,***
- ***we have to accept the possibility of no solution being found,***
- ***we may have to wait an unacceptably long time.***

Reality:

Actually, usually the above is *not* a myth, but occasionally we have *Technological Trivialization* of the problem.

Ultimately any problem can be thought of a finite problem, i.e., for any problem, we can build a finite state machine that will solve all the problem for all input no larger than a given maximum.

However, usually, the number of states of the machine is so large as to be impractical; the memory of real machines is too small or real machines are not fast enough to complete the computation in a reasonable time even though the complexity of the solution is linear (with a BIG multiplicative constant).

So we have gotten into the habit of considering the problem unbounded and finding algorithms that handle all possible inputs (the Turing Machine game).

Many times these algorithms have exponential or higher growth.

So we work on heuristic, probabilistic, and approximate solutions.

However, surprise, surprise, surprise!!

For some of these problems, machine sizes and speeds have grown to the point that for all possible inputs that happen in real life, exploring all possible states is feasible.

We can suddenly write programs that give complete, exact solutions in a reasonable amount of time for all possible inputs.

For example, the problem of optimal floor plan layout according to specified criteria is normally NP-complete.

However, recently a representation for a floor plan was discovered such that:

For all floor plans that are possible for a normal house (e.g., < 20 rooms),

> it is possible to generate all unique-up-to-symmetries floor plans and to evaluate all according to the criteria in less than 1 hour.

Suddenly, all the heuristic, approximate, and probabilistic solutions are unnecessary.

Same thing happened with relational databases, which were only a nice theory until mid 80s.

Conclusions

Truth:

A good software engineer is a lazy one.

What kind of laziness?

- **planning ahead to avoid work**
- **reuse**

Truth:

We software engineers need humility, humility to recognize our limitations,

- **the need for help, and**
- **the limitations of the help, both methodological and technical.**