The Requirements Iceberg and Various Icepicks Chipping at It

Daniel M. Berry

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.**

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!**

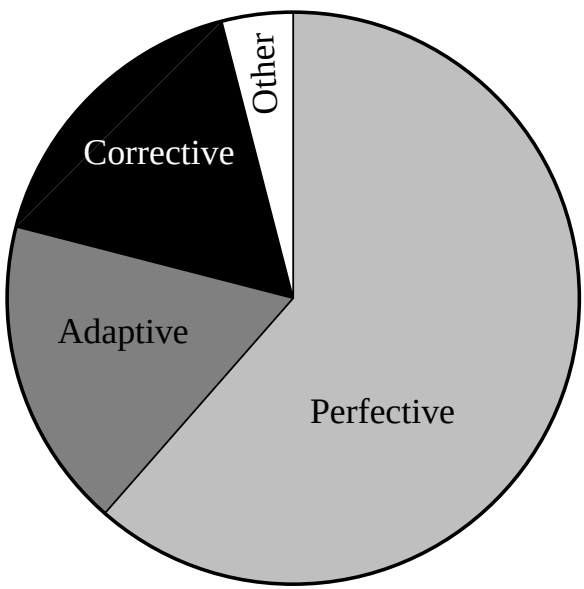
E-Type Software, Cont'd

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!

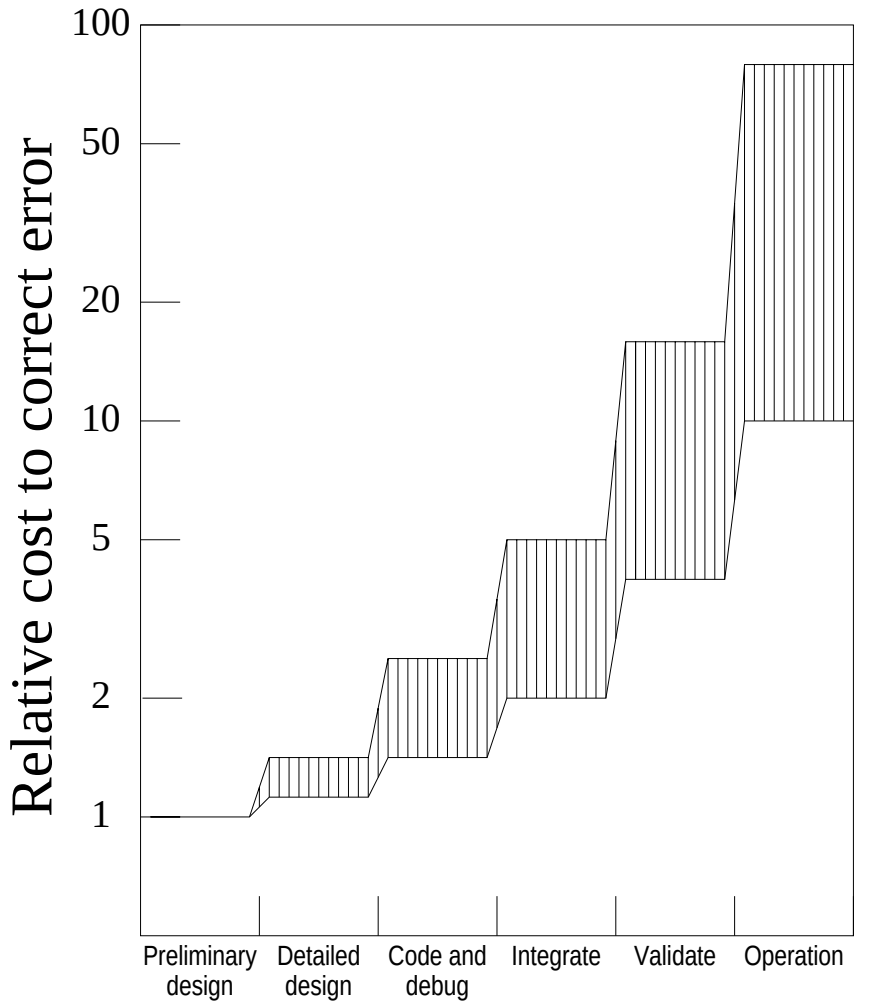


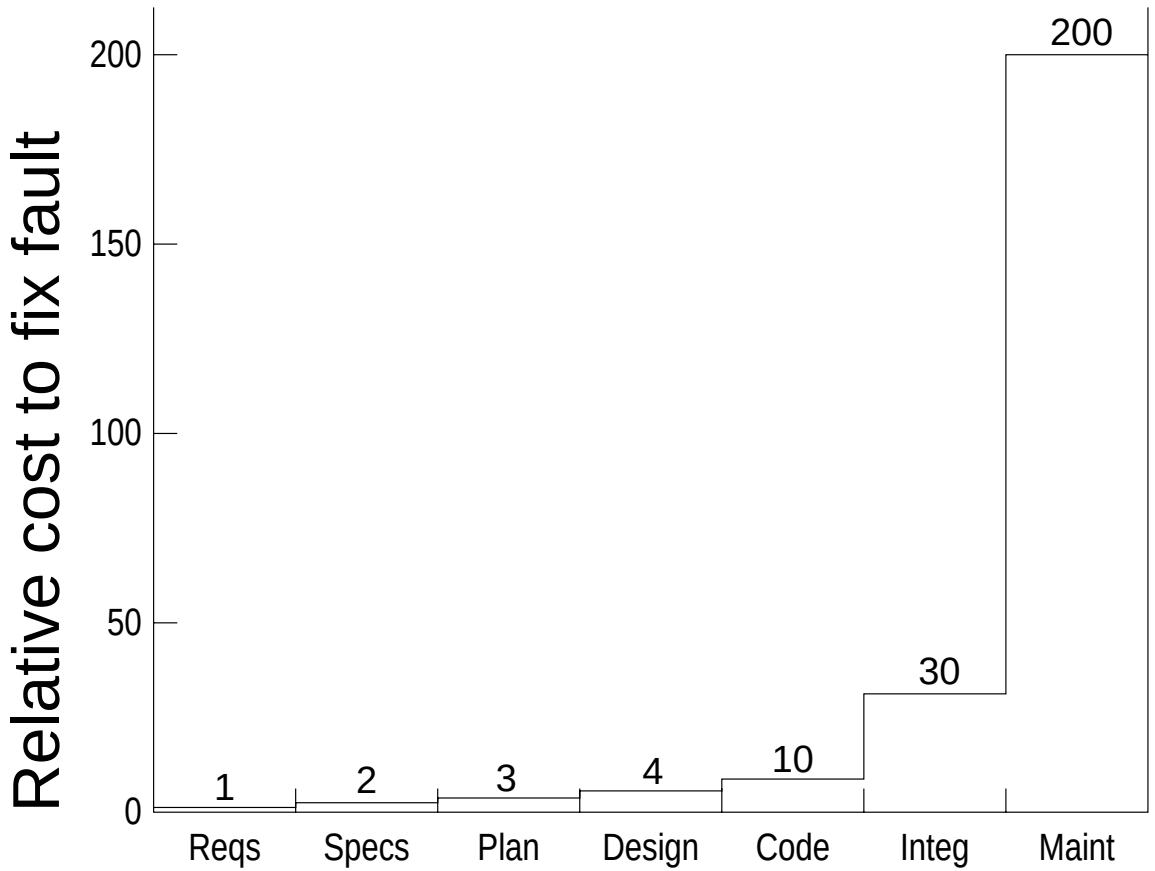
So, no matter what, we have to deal with changes.

How much do these changes cost?

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.

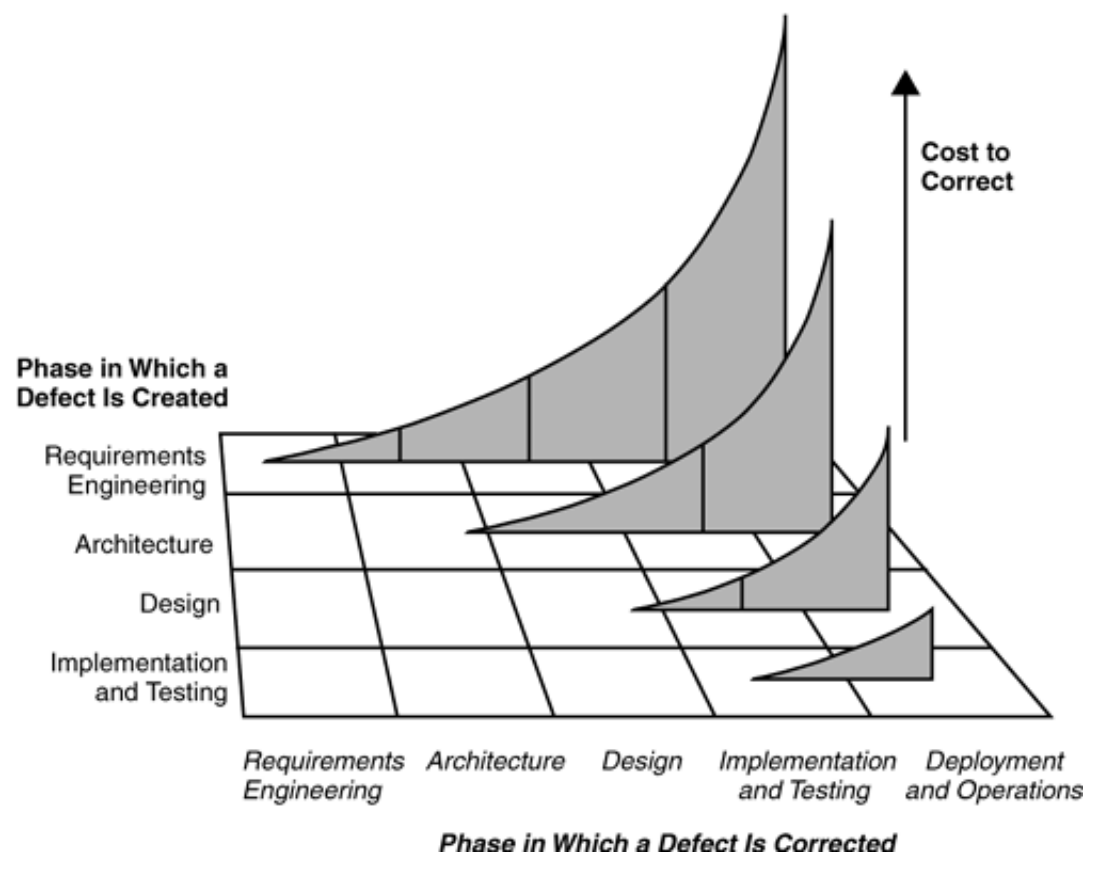

Relative cost to correct error
100
50
20
10
5
2
1
Preliminary design
Detailed design
Code and debug
Integrate
Validate
Operation
Phase in which error is detected


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!

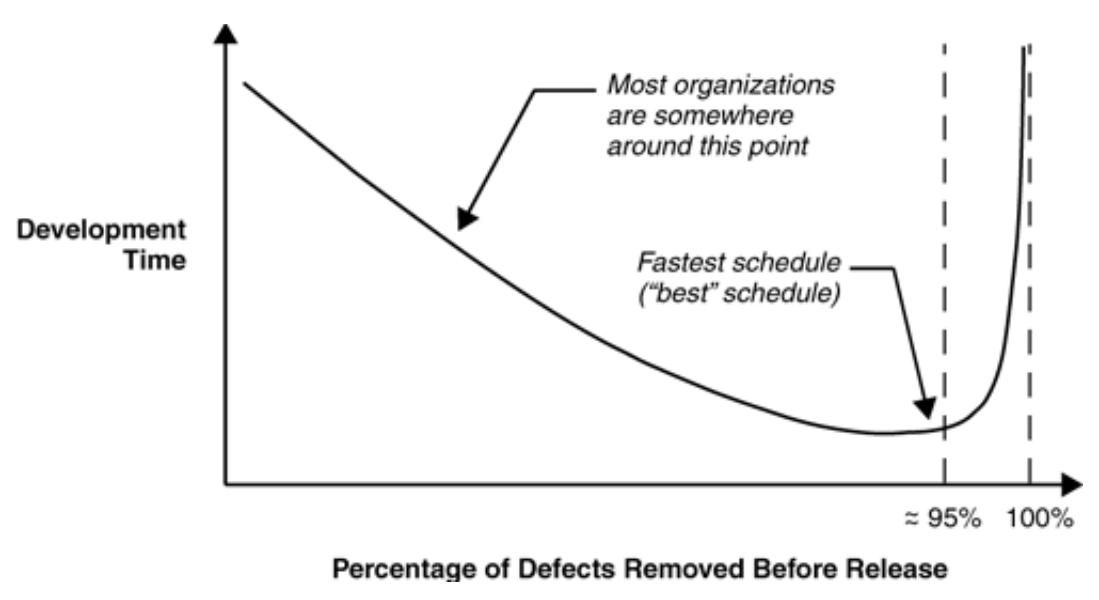
Conclusion, Cont'd

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

Technical debt (also known as design debt or code debt) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy [cheaper] solution now instead of using a better approach that would take longer [and appears to cost more].

— result of googling “technical debt” with my additions in brackets

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.