Real Software Engineering

Glenn Vanderburg
InfoEther
glv@vanderburg.org

Software Engineering Doesn't Work!

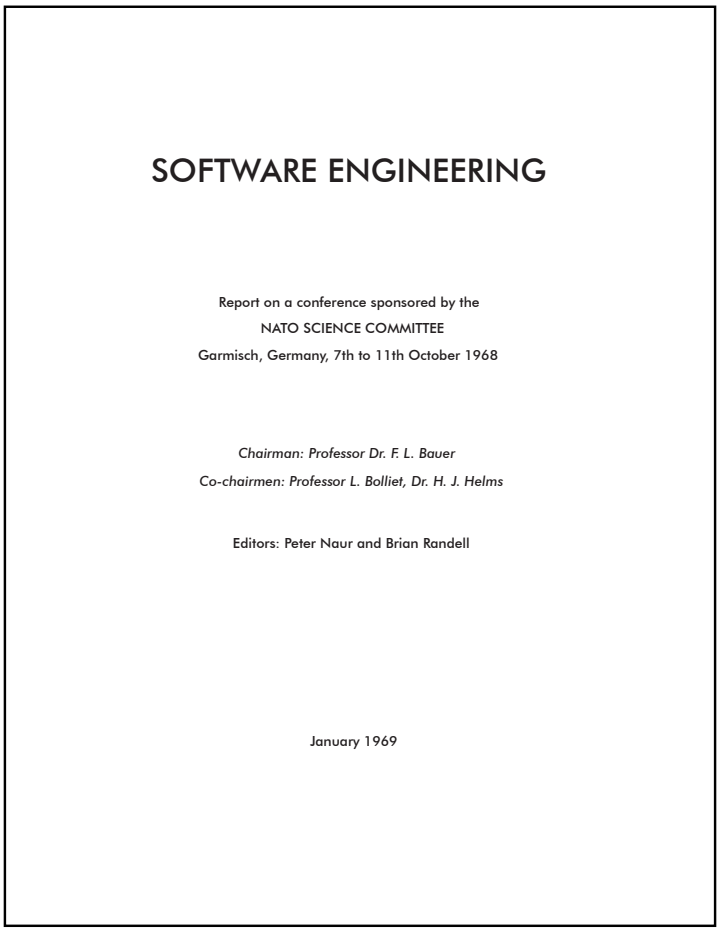
Real
Engineering

Real
Software
Engineering

Caricature of
Engineering

Real
Software

A Caricature of Engineering



SOFTWARE ENGINEERING

Report on a conference sponsored by the
NATO SCIENCE COMMITTEE
Garmisch, Germany, 7th to 11th October 1968

Chairman: Professor Dr. F. L. Bauer
Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

1. A software system can best be designed if the testing is interlaced with the designing instead of being used after the design.

2. A simulation which matches the requirements contains the control which organizes the design of the system.

3. Through successive repetitions of this process of interlaced testing and design the model ultimately becomes the software system itself. [...] in effect the testing and the replacement of simulations with modules that are deeper and more detailed goes on with the simulation model controlling, as it were, the place and order in which these things are done.

MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS

Dr. Winston W. Royce

INTRODUCTION

I am going to describe my personal views about managing large software developments. I have had various assignments during the past nine years, mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis. In these assignments I have experienced different degrees of success with respect to arriving at an operational state, on-time, and within costs. I have become prejudiced by my experiences and I am going to relate some of these prejudices in this presentation.

COMPUTER PROGRAM DEVELOPMENT FUNCTIONS

There are two essential steps common to all computer program developments, regardless of size or complexity. There is first an analysis step, followed second by a coding step as depicted in Figure 1. This sort of very simple implementation concept is in fact all that is required if the effort is sufficiently small and if the final product is to be operated by those who built it — as is typically done with computer programs for internal use. It is also the kind of development effort for which most customers are happy to pay, since both steps involve genuinely creative work which directly contributes to the usefulness of the final product. An implementation plan to manufacture larger software systems, and keyed only to these steps, however, is doomed to failure. Many additional development steps are required, none contribute as directly to the final product as analysis and coding, and all drive up the development costs. Customer personnel typically would rather not pay

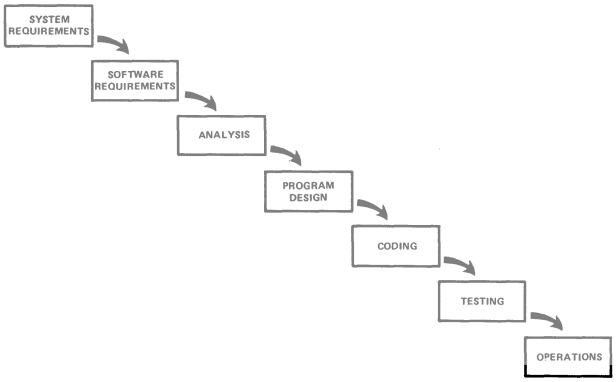


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

I believe in this concept, but the implementation described above is risky and invites failure. The problem is illustrated in Figure 4. The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.

One might note that there has been a skipping-over of the analysis and code phases. One cannot, of course, produce software without these steps, but generally these phases are managed with relative ease and have little impact on requirements, design, and testing. In my experience there are whole departments consumed with the analysis of orbit mechanics, spacecraft attitude determination, mathematical optimization of payload activity and so forth, but when these departments have completed their difficult and complex work, the resultant program steps involve a few lines of serial arithmetic code. If in the execution of their difficult and complex work the analysts have made a mistake, the correction is invariably implemented by a minor change in the code with no disruptive feedback into the other development bases.

However, I believe the illustrated approach to be fundamentally sound. The remainder of this discussion presents five additional features that must be added to this basic approach to eliminate most of the development risks.

An implementation plan ... keyed only to these steps, however, is doomed.

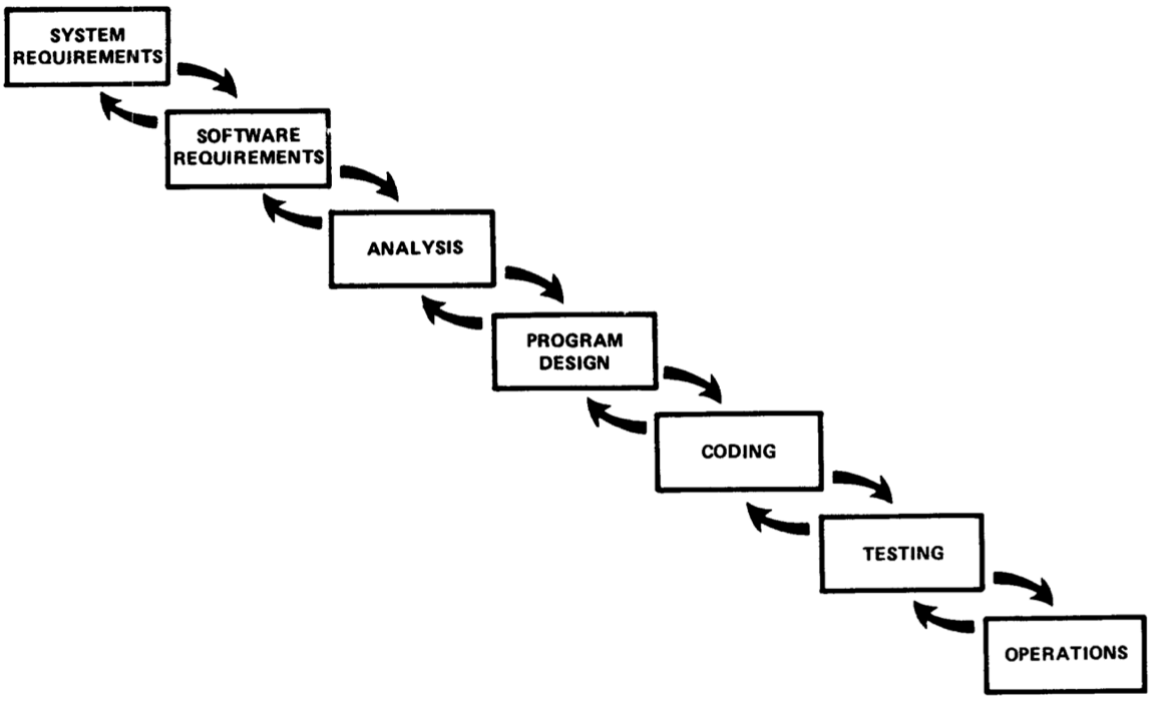


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

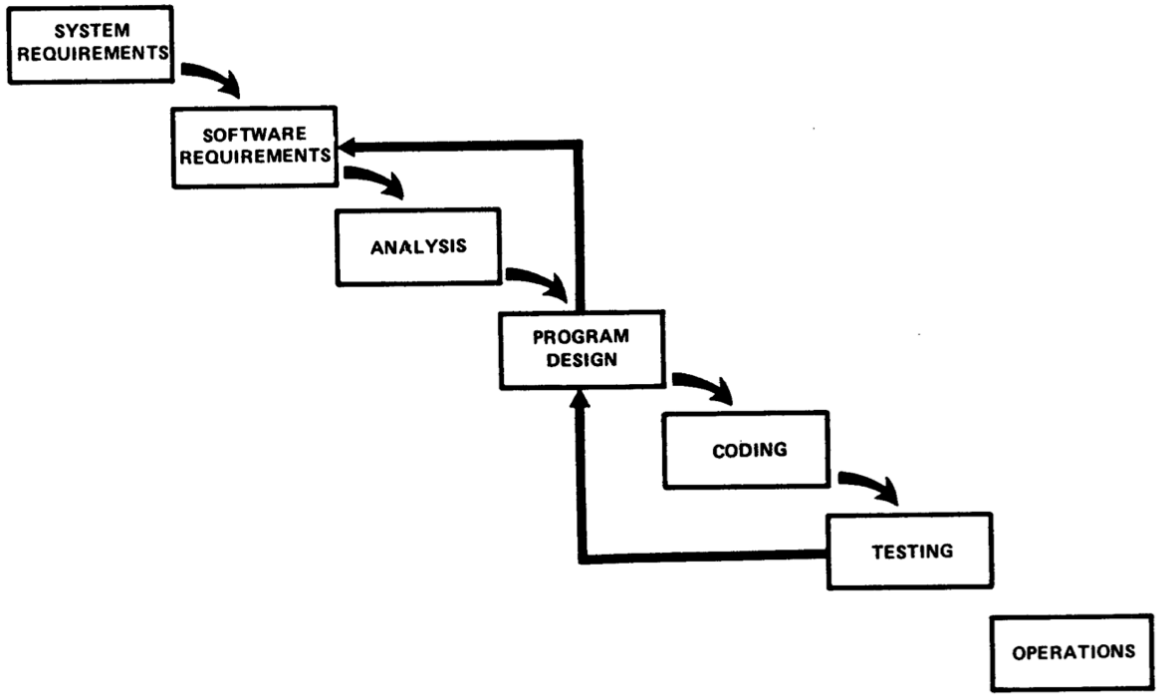


Figure 4. Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps.

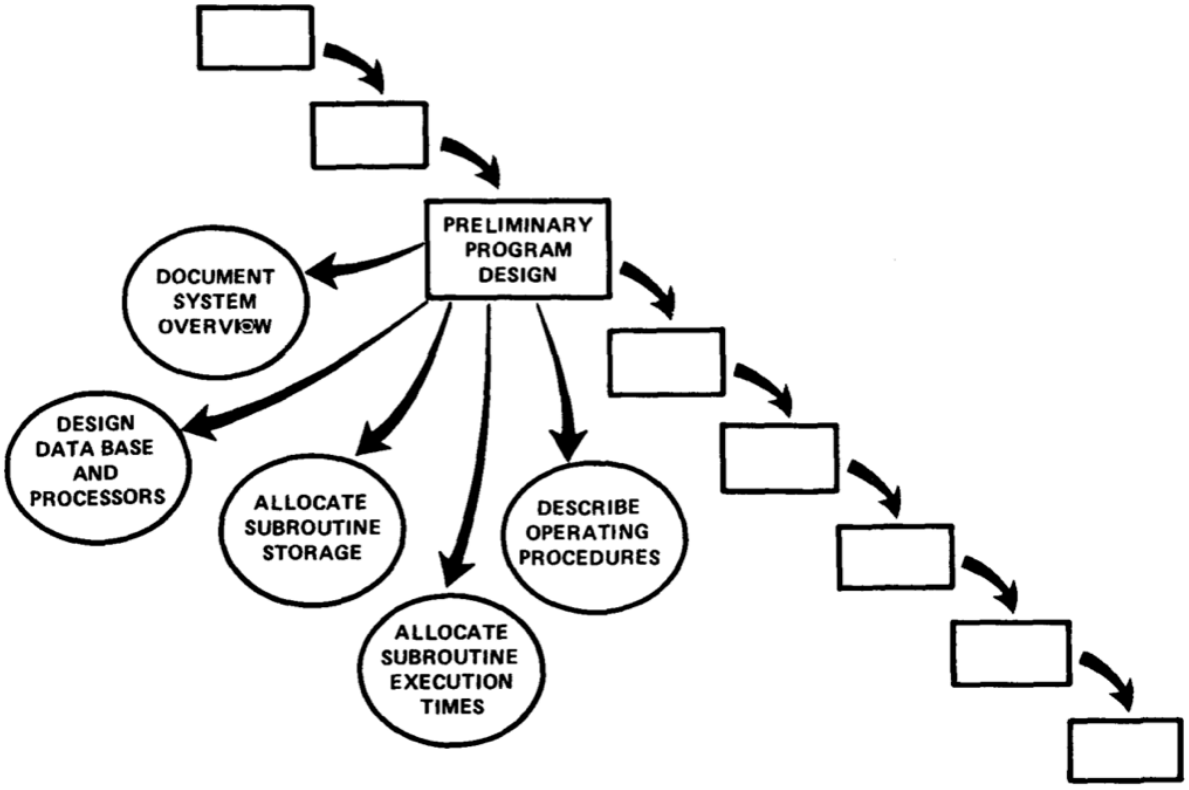


Figure 5. Step 1: Insure that a preliminary program design is complete before analysis begins.

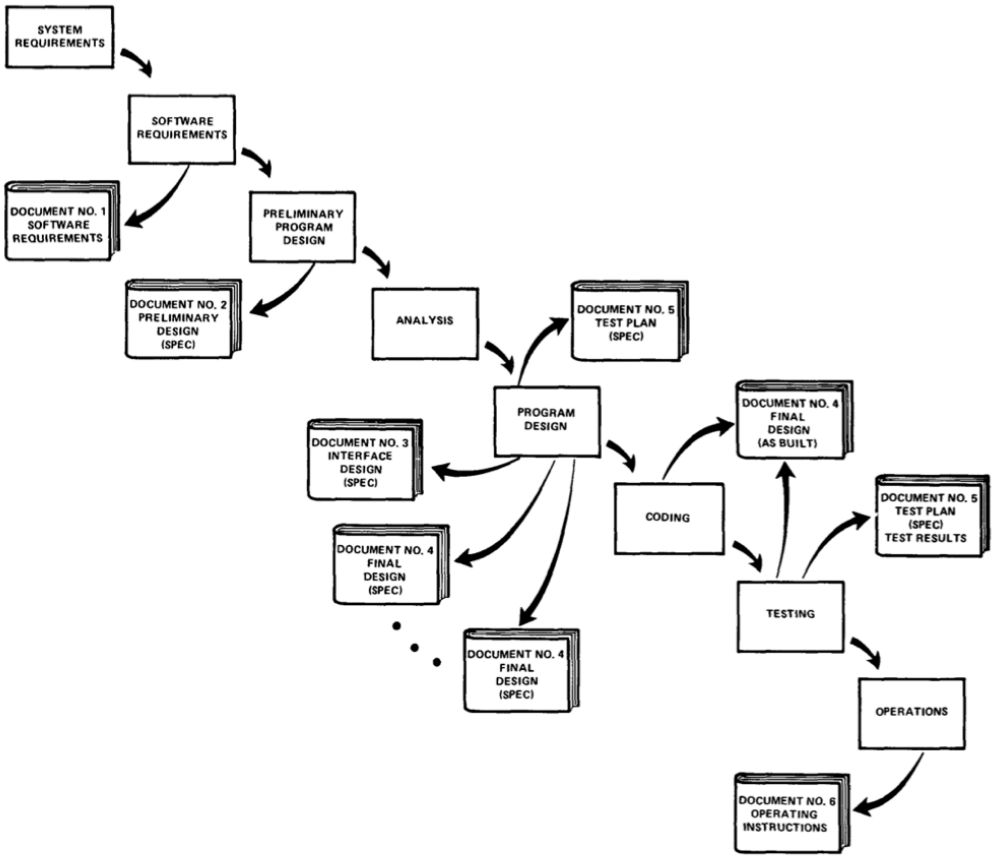


Figure 6. Step 2: Insure that documentation is current and complete – at least six uniquely different documents are required.

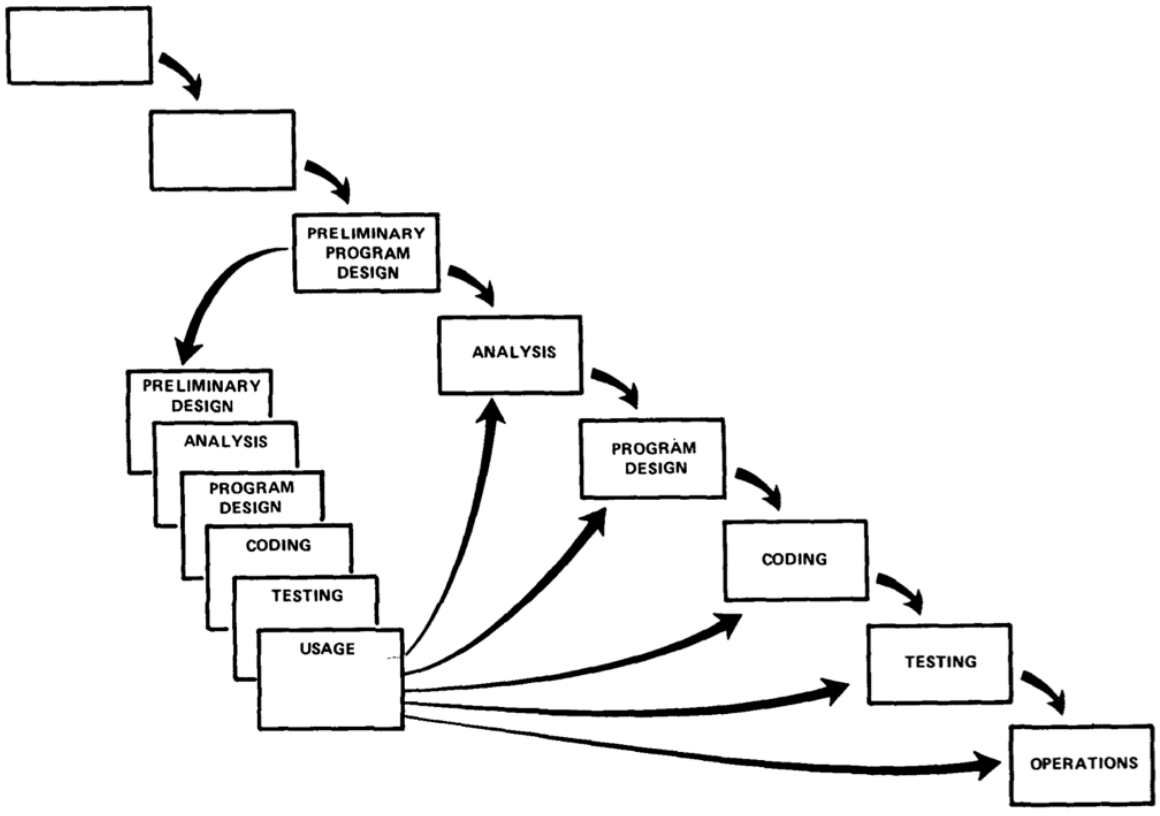


Figure 7. Step 3: Attempt to do the job twice – the first result provides an early simulation of the final product.

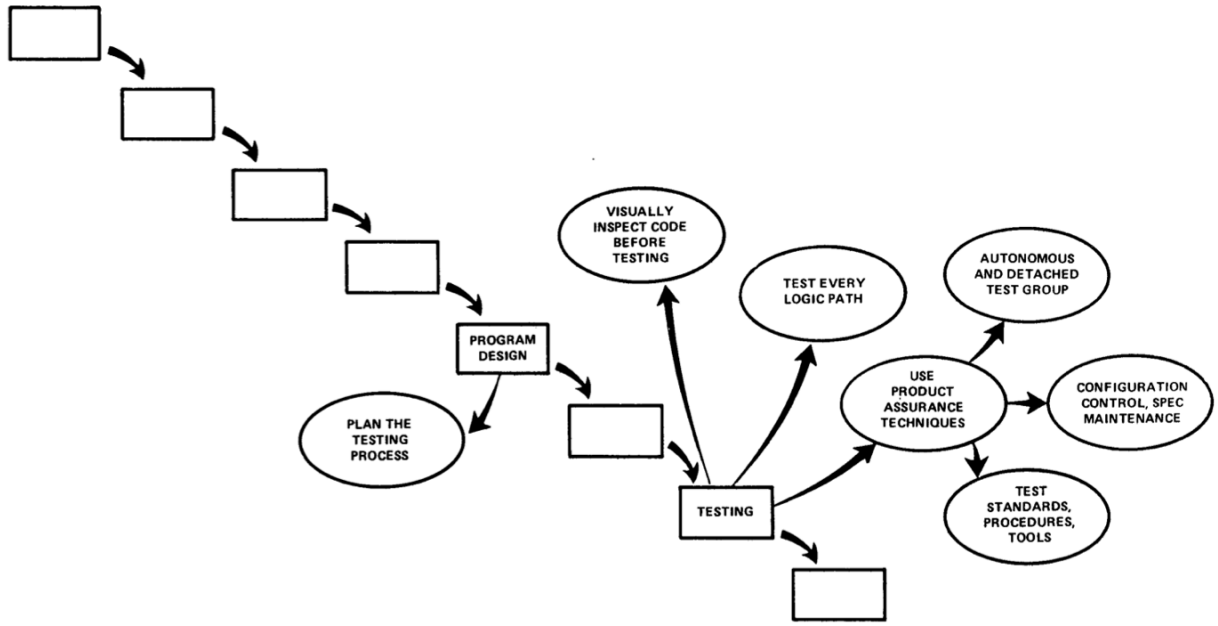


Figure 8. Step 4: Plan, control, and monitor computer program testing.

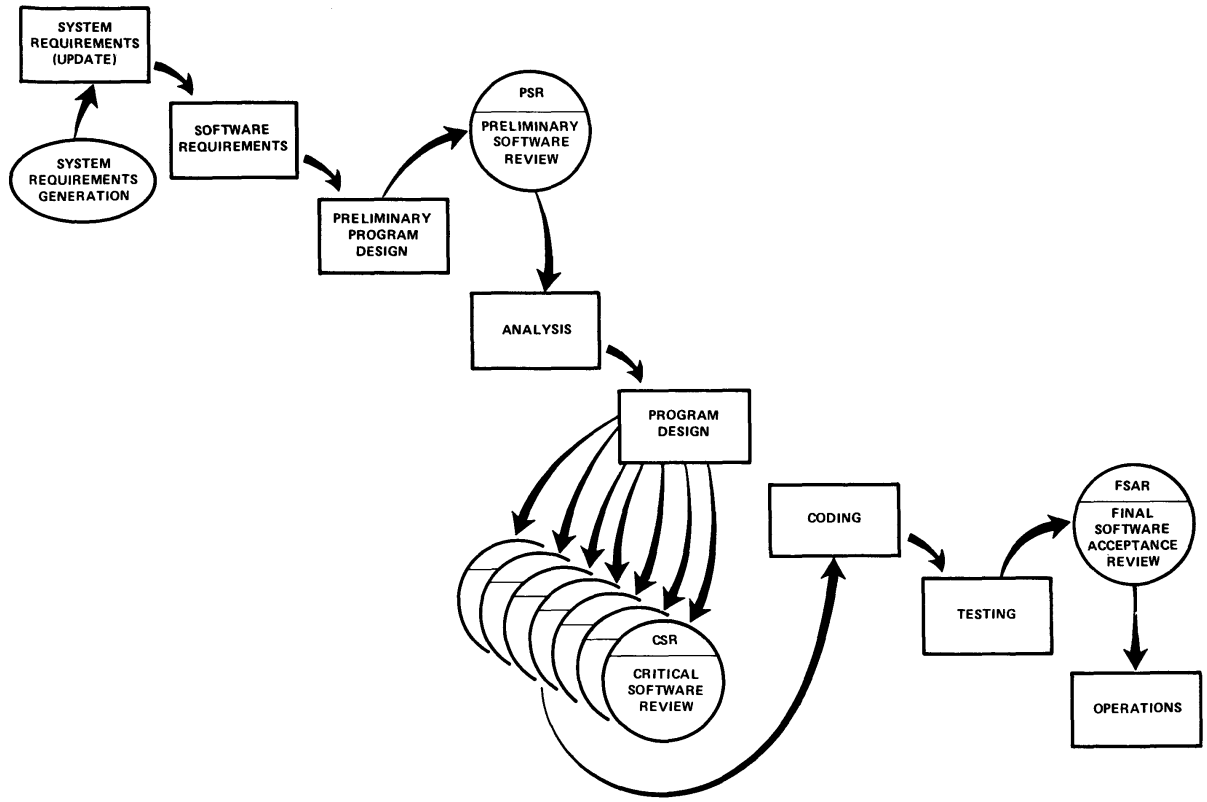


Figure 9. Step 5: Involve tne customer — the involvement should be formal, in-depth, and continuing.

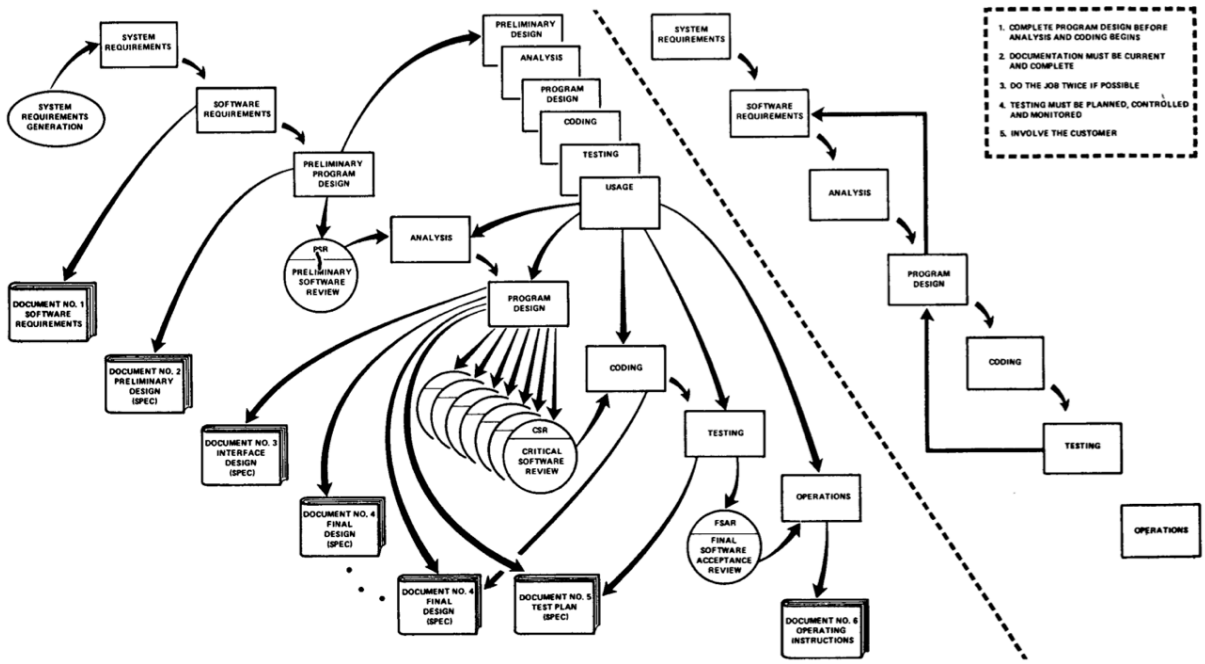


Figure 10. Summary

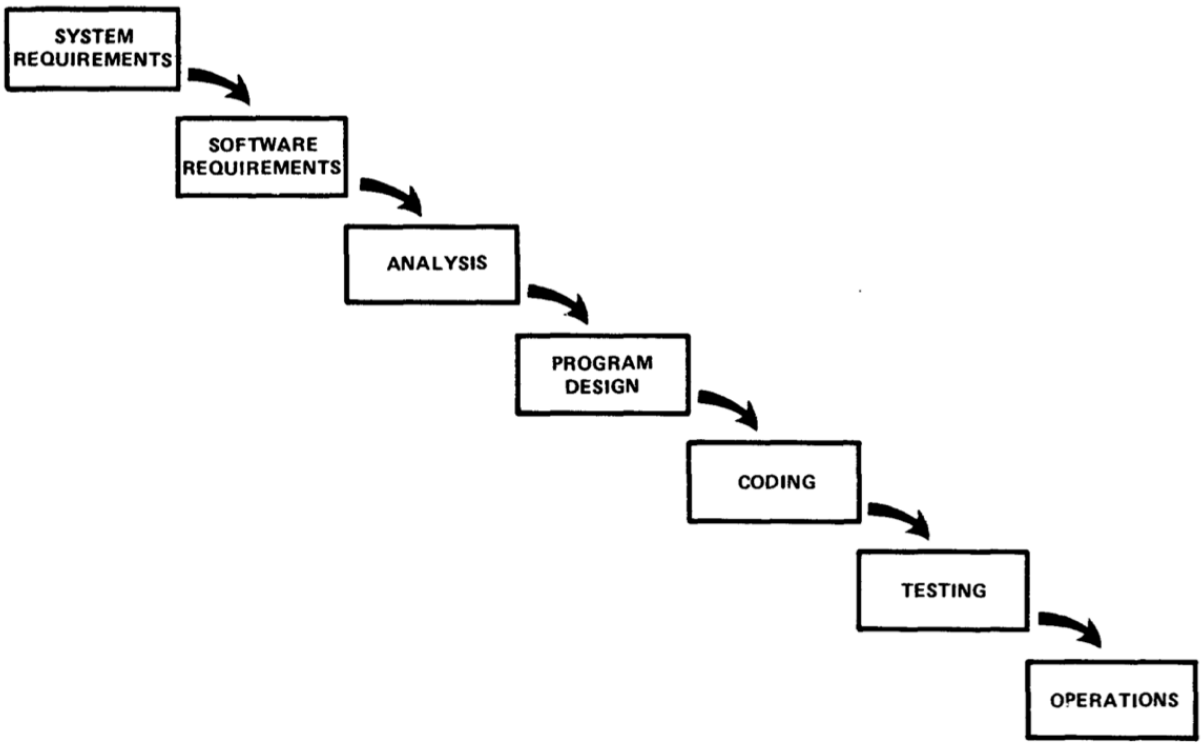


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

For every complex problem
there is a solution that is
simple, neat, and wrong.
—H. L. Mencken

The *defined process control model*
requires that every piece of
work be completely understood.
A defined process can be started
and allowed to run until
completion, with the same
results every time.

“A Rational Design Process”

A. Establish and document requirements
B. Design and document the module structure
C. Design and document the module interfaces
D. Design and document the uses hierarchy
E. Design and document the module internal structures
F. Write programs
G. Maintain

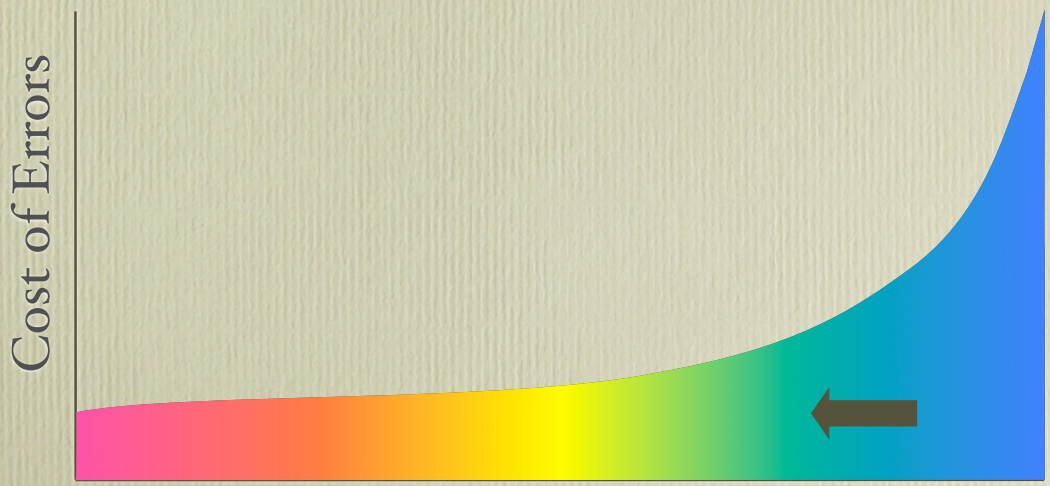
From that same paper!

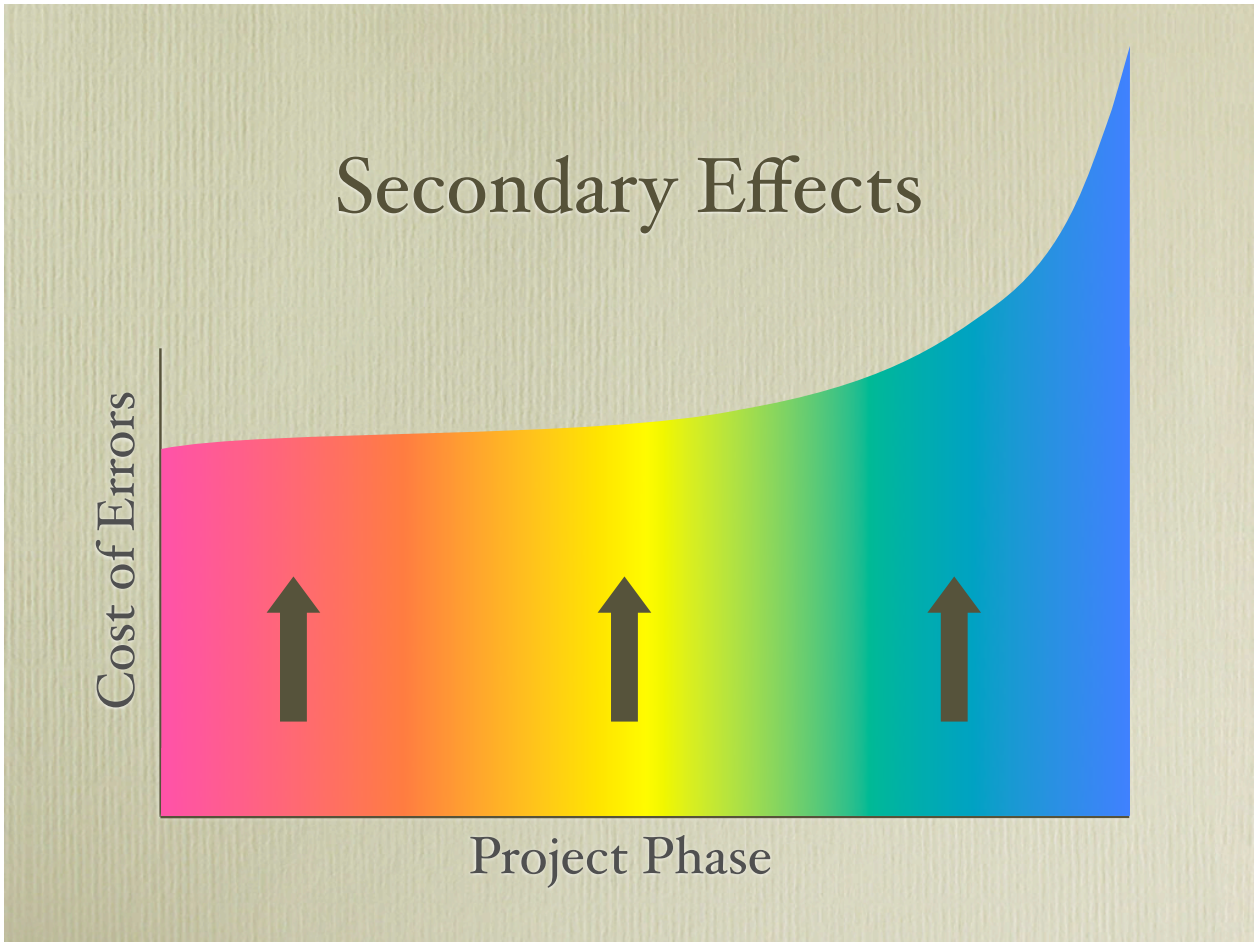
[...] the picture of the software designer deriving his design in a rational way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will.

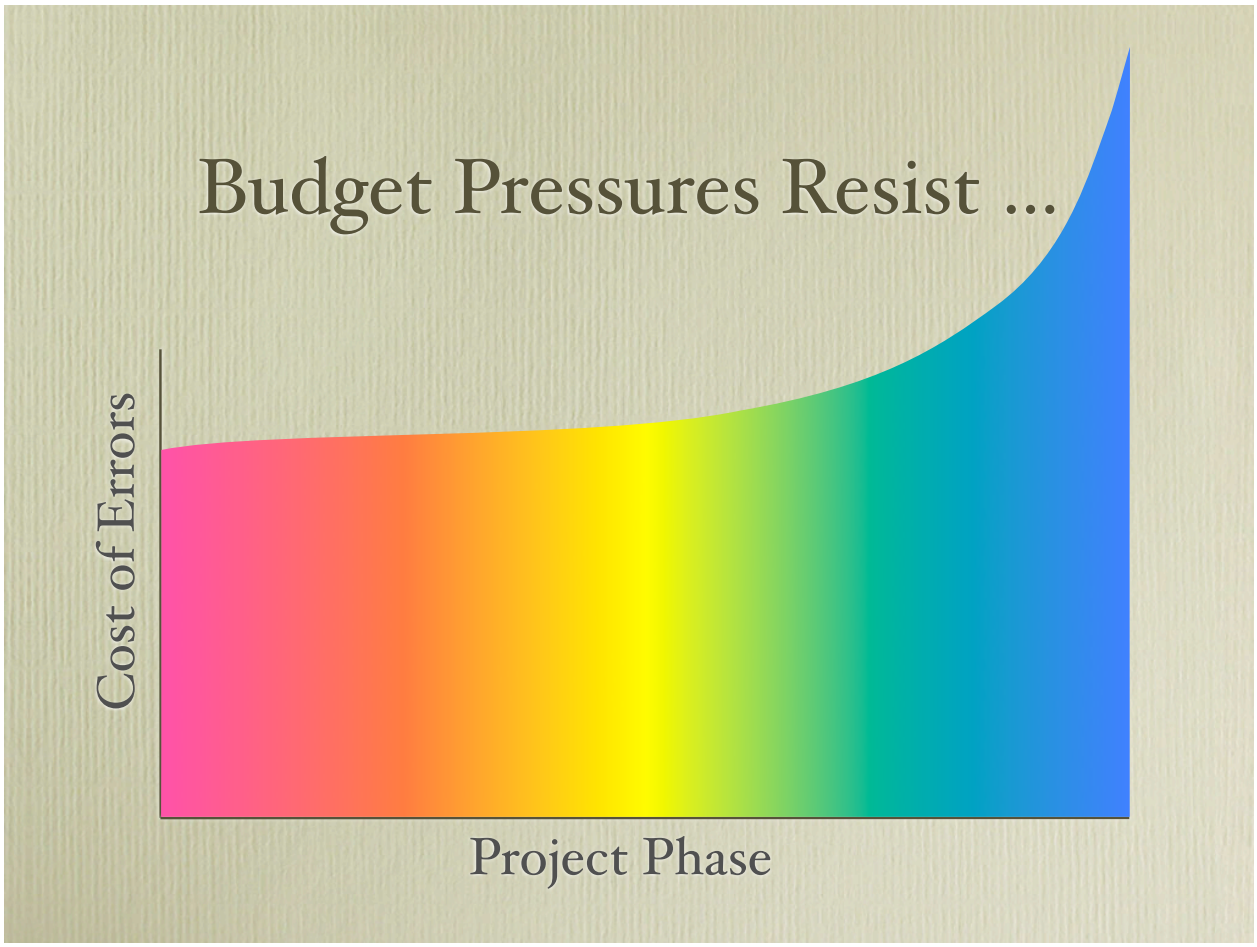
—*David Parnas*

Cost of Errors

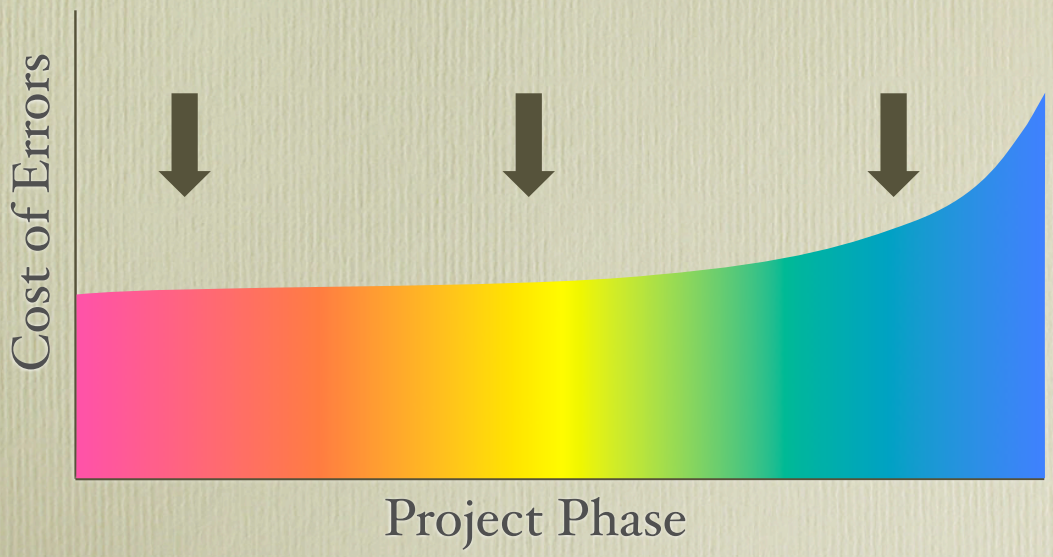
The Software Engineering Solution




Secondary Effects
Cost of Errors
Project Phase


Budget Pressures Resist ...
Cost of Errors
Project Phase

Budget Pressures Resist ...



Budget Pressures Resist ...

Modeling and Math Envy

"In engineering ... people design through documentation."

OverrideDose

OverF

item? ∈ { p_dose, p_time }
overridden' =
  if dose ∉ **dom** overridden
  then overridden ⊕ { dose ↦ accumulated dose }
  else { dose } ⩤ overridden

OverrideF

Condition							Value N(T))
N(p(T)) = Pass							Pass
N(p(T)) = Fail							Fail
N(p(T))≠ Pass ∧ N(p(T))≠ Fail	r(T)= N(p(T))	N(p(T)) = L					Pass
		N(p(T)) ≠ L					N(p(T))+1
	r(T)≠ N(p(T))	r(T)≠ esc	r(p(T))≠ N(p(p(T)))	r(p(T))= esc	p(p(T))= _		N(p(T))
					p(p(T)) ≠_	r(p(p(T)))= N(p(p(p(T))))	N(p(T))–1
						r(p(p(T))) ≠ N(p(p(p(T))))	N(p(T))
				r(p(T)) ≠ esc			N(p(T))
			r(p(T)) = N(p(p(T)))				N(p(T))-1
		r(T)= esc	r(p(T))=esc	N(p(p(T))) = esc			N(p(T))
				N(p(p(T))) ≠ esc			Fail
			r(p(T)) ≠ esc				N(p(T))

CASE Tools

Booch

OOSE

Z

UML

VDM

MDA

OMT

This is not some kind of engineering where all we have to do is put something in one end and turn the crank.

—*Bruce Eckel*

Real Engineering

There is no kind of engineering
where all we have to do is
put something in one end
and turn the crank.

Different Engineering Disciplines are *Different*

- Different materials, physical effects, forces
- Different degrees of complexity in requirements, designs, processes, and artifacts
- Varied reliance on formal modeling and analysis vs. experimentation, prototyping, and testing
- Varied use of defined and empirical processes

The *defined process control* model requires that every piece of work be completely understood. A defined process can be started and allowed to run until completion, with the same results every time.

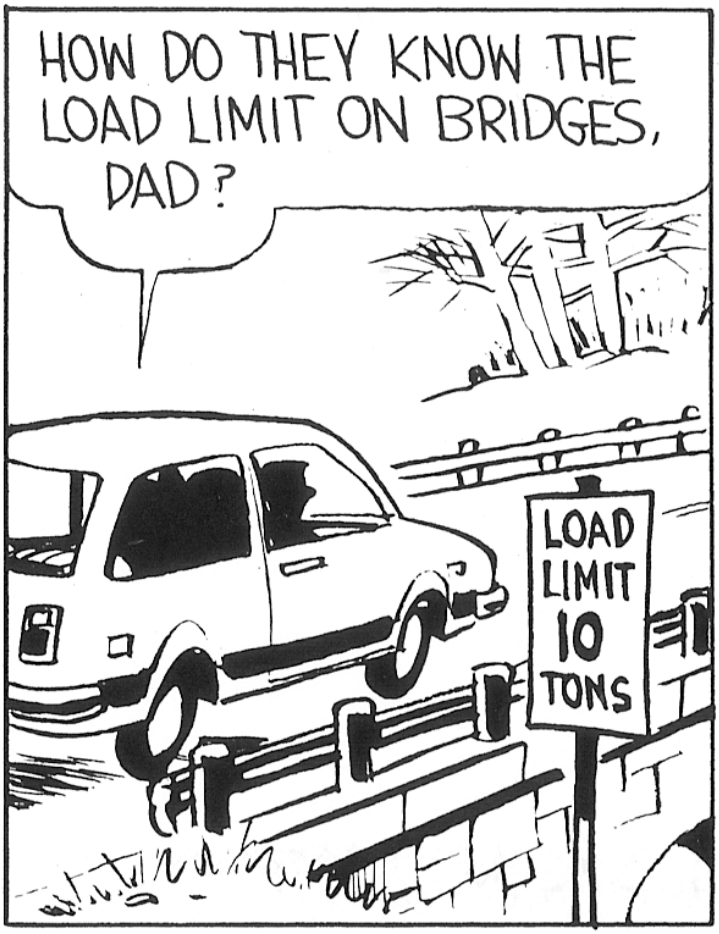
The *empirical process control model* provides and exercises control through frequent inspection and adaptation for processes that are imperfectly defined and generate unpredictable and unrepeatable outputs.

Cost is always an object.

Engineering is not the art
of constructing. It is rather
the art of not constructing:
or, it is the art of doing well
with one dollar what any
bungler can do with two.
—*Arthur Mellen Wellington*

Advances come from practitioners.

Mathematical modeling
was introduced as a
cost-saving measure.


HOW DO THEY KNOW THE LOAD LIMIT ON BRIDGES, DAD?
LOAD LIMIT 10 TONS


THEY DRIVE BIGGER AND BIGGER TRUCKS OVER THE BRIDGE UNTIL IT BREAKS.


THEN THEY WEIGH THE LAST TRUCK AND REBUILD THE BRIDGE.
WATTERSON


OH. I SHOULD'VE GUESSED.
DEAR, IF YOU DON'T KNOW THE ANSWER, JUST TELL HIM!

Structural engineering is
the science and art
of designing and making,
with economy and elegance,
[...] structures so that they
can safely resist the forces to
which they may be subjected.

—*Structural Engineer's Association*

Real
Software Engineering

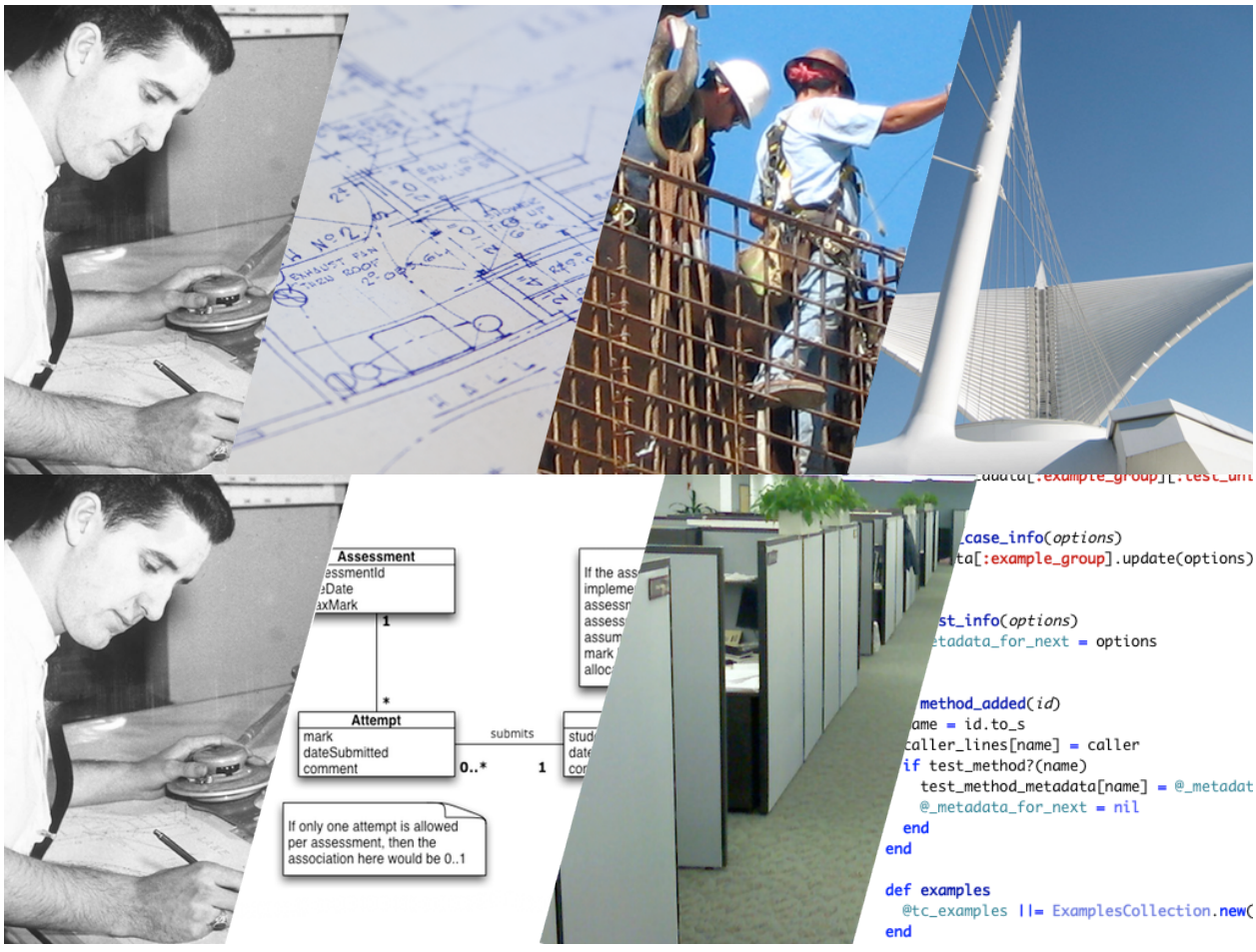
Software engineering is
the science and art
of designing and making,
with economy and elegance,
[...] systems so that they can
readily adapt to the situations to
which they may be subjected.

Software engineering will be *different* from other kinds of engineering.

The testing phase ... is the first event for which timing, storage, input/output transfers, etc., are *experienced* as distinguished from *analyzed*. These phenomena are not precisely analyzable.

Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required. In effect the development process has returned to the origin and one can expect up to a 100-percent overrun in schedule and/or costs.

—*Winston Royce*



What is needed is not **classical** mathematics, but **mathematics.** Systems should be built at levels and modules, which form a mathematical structure.
—Friedrich Bauer

We, in the Netherlands, have the title Mathematical Engineer. Software engineering seems to be the activity for the Mathematical Engineer *par excellence*. [...] our basic tools are mathematical in nature.
—Edsger Dijkstra

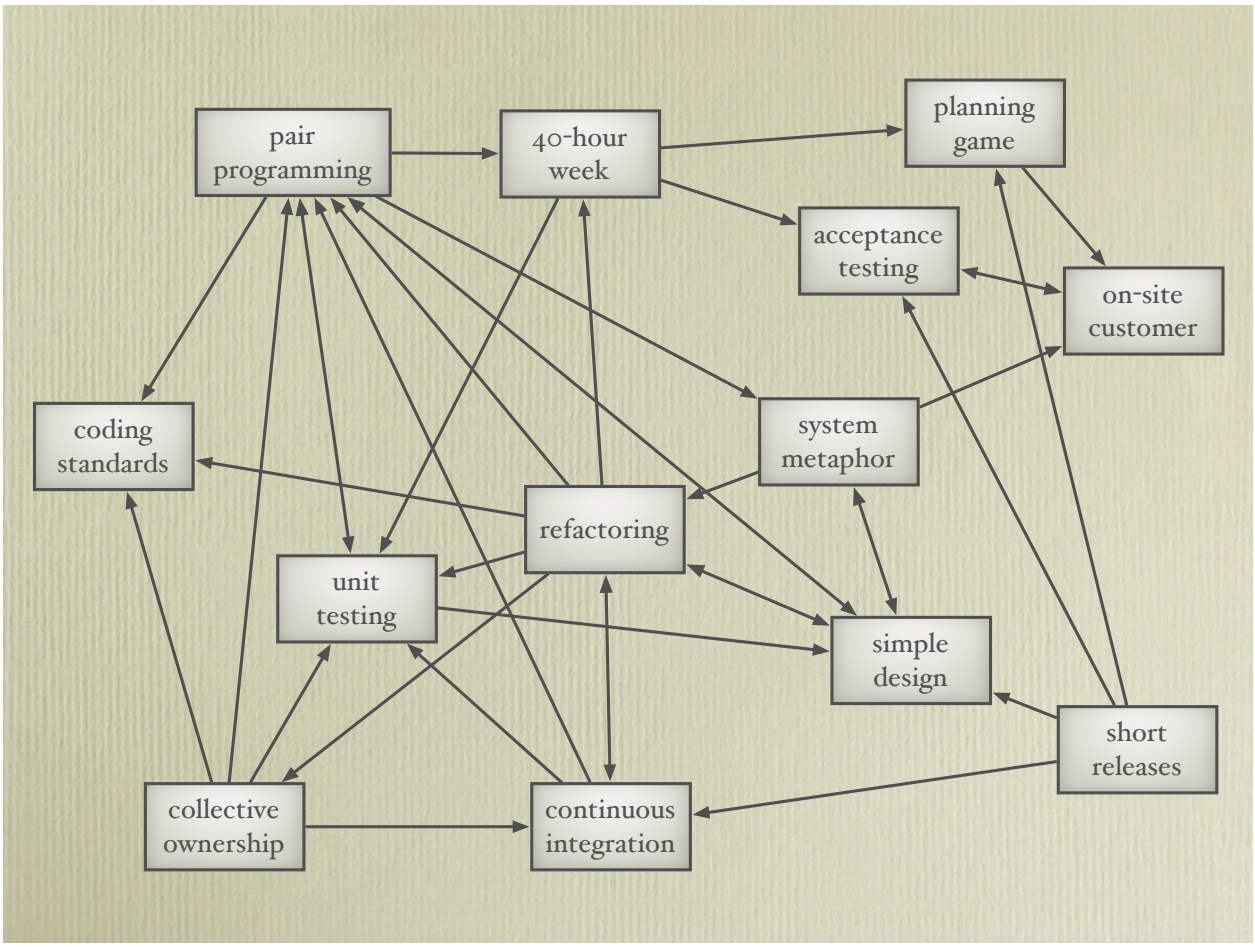
divided by	2	3	4	7
9	4.5	3	2.5	1.29
10	5.0	3.33	2.5	1.43
11	5.5	3.66	2.75	1.57
12.6	6.3	4.2	3.15	1.8
22	11.0	7.33	5.5	3.14
100	50.0	33.33	25.0	14.29

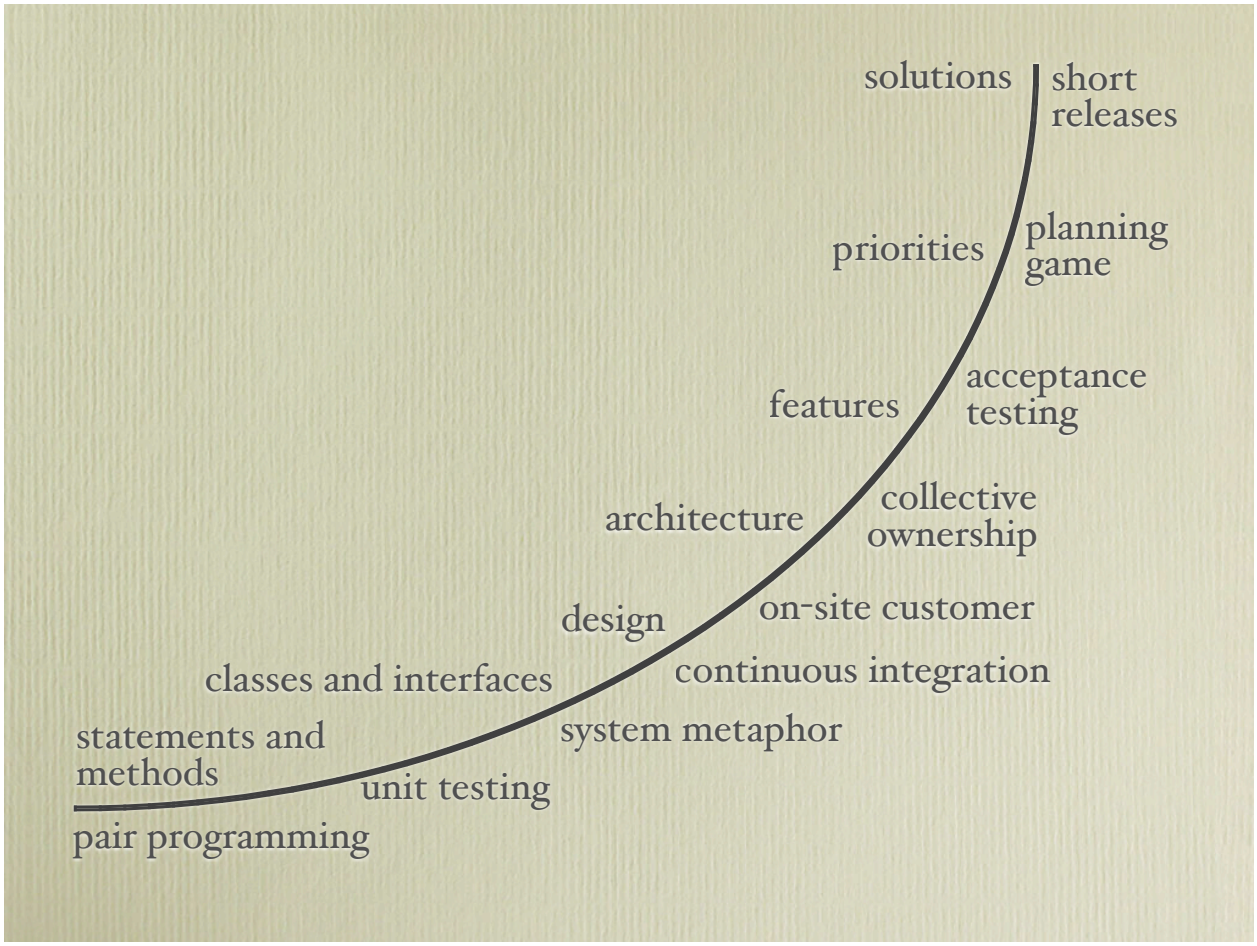
```
Feature: Addition
  In order to avoid silly mistak
  As an error-prone person
  I want to divide two numbers

  Scenario Outline: Divide two nu
    Given I have entered <input_
    And I have entered <input_2>
    When I press "divide"
    Then the result should be <r

  Examples:
    | input_1 | input_2 | result  |
    | 10      | 2       | 5.0     |
    | 12.6    | 3       | 4.2     |
    | 22      | 7       | ~=3.14  |
    | 9       | 3       | <5      |
    | 11      | 2       | 4<_<6   |
    | 100     | 4       | 32      |
```

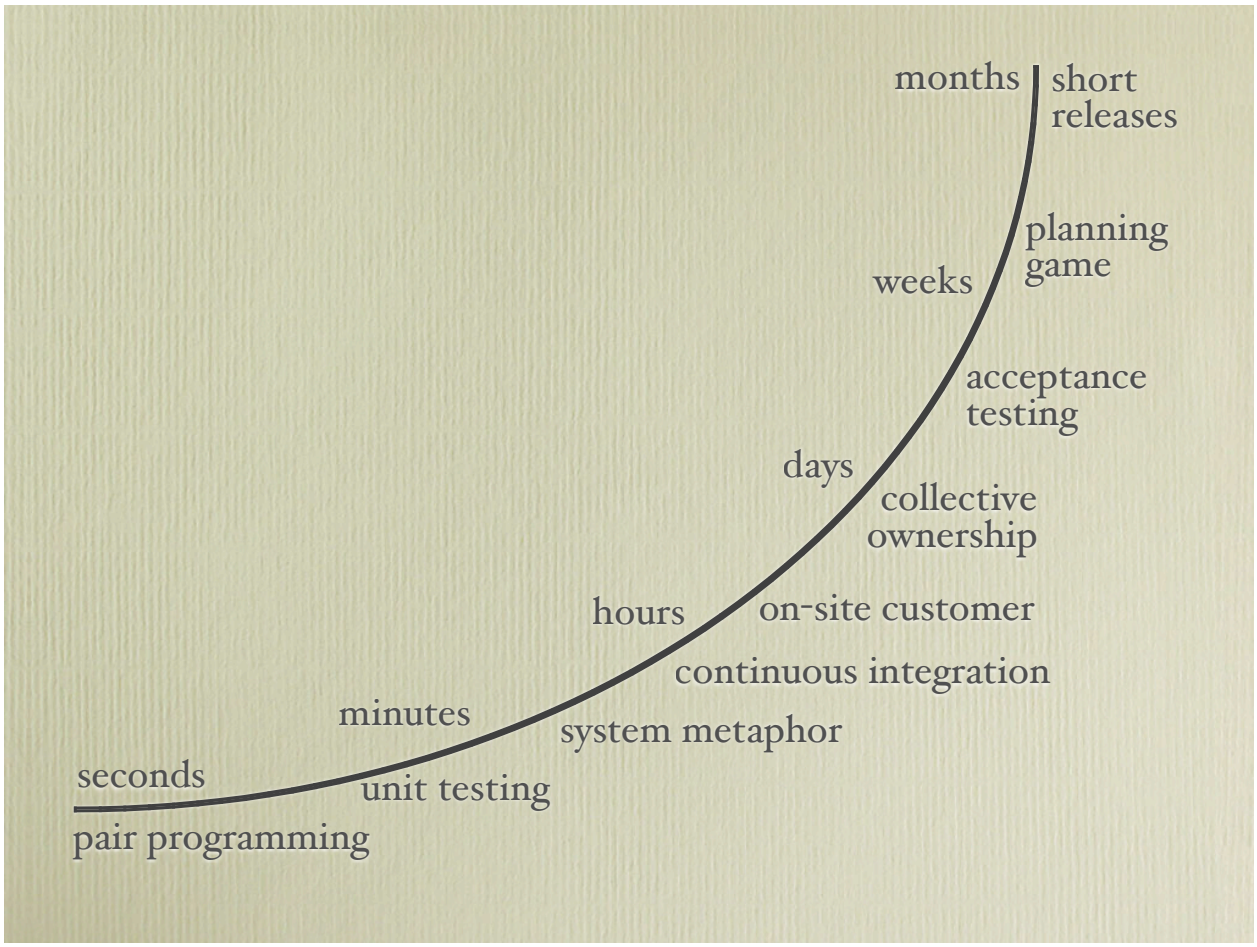
eg.Division		
numerator	denominator	quotient?
10	2	5.0
12.6	3	4.2
22	7	~=3.14
9	3	<5
11	2	4<_<6
100	4	32 expected / 25 actual

```
    assert_in_delta  3.1
    assert           5
    assert           4..
    assert_equal     32,
  end
end
```

```
ers" do
 do
 .should == 5.0
 .should == 4.2
 .should be_close(3.14, 0.01)
  (9 / 3)    .should < 5
  (11/2)     .should satisfy{|n| n > 4 && n < 6 }
  (100 / 4) .should == 32
 end
end
```




solutions
short releases
priorities
planning game
features
acceptance testing
architecture
collective ownership
design
on-site customer
classes and interfaces
continuous integration
statements and methods
system metaphor
unit testing
pair programming


months
short releases
planning game
weeks
acceptance testing
days
collective ownership
hours
on-site customer
continuous integration
minutes
system metaphor
seconds
unit testing
pair programming

Assumptions Once True, But No Longer

- Code is hard to read
- Code is hard to change
- Testing is expensive

Assumptions Once Believed But Never True

- All engineering is like structural engineering
- Programming is like building
- Modeling and analysis are about correctness

The Reality of Software Engineering

- Software is very unlike bridges and buildings.
- Additional complexity hinders requirements, design, and approval.
- Source code *is* a model.
- Building and testing our interim designs is effectively free.
- Empirical processes are rational for software.

Agile software development, far from being anti-engineering, represents the responsible application of engineering ideals to the field of software.

Real Software Engineering

Glenn Vanderburg
InfoEther
glv@vanderburg.org
http://spkr8.com/t/3398

Photo Credits (all from Flickr):

Seattle Municipal Archives:	/photos/seattlemunicipalarchives/2713475713/
Will Scullin:	/photos/wscullin/3770015203/
Bill Jacobus:	/photos/billjacobus1/122497422/
kke:	/photos/kesta/1818986646/
Carol Shergold:	/photos/carolshergold/1921464196/
Bill Abbott:	/photos/wbaiv/3236672907/