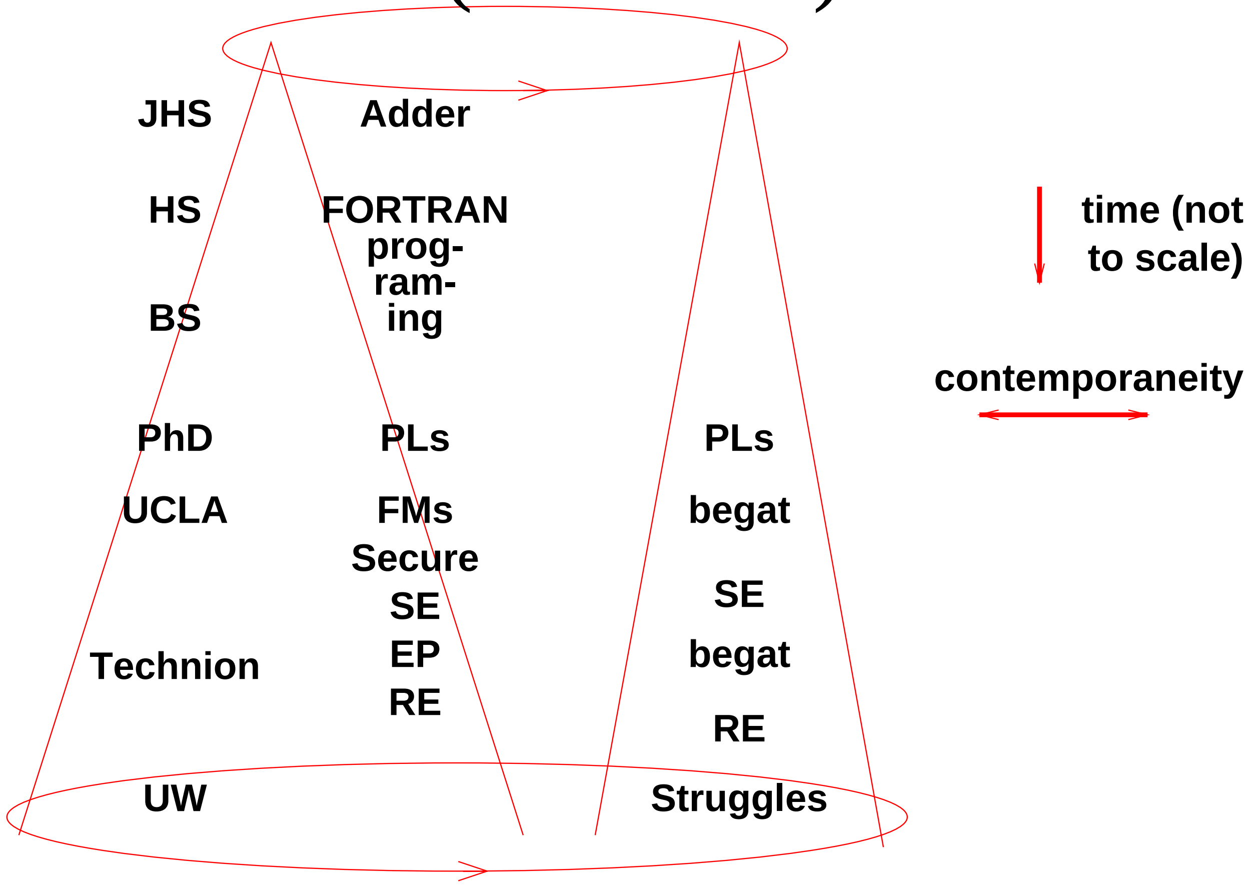
The Prehistory and History of RE (+ SE) as Seen by Me: How My Interest in FMs Helped to Move Me to RE

Daniel M. Berry
University of Waterloo, Canada
dberry@uwaterloo.ca

See Slides 39--41, 46--51, and 60--65. They discuss how much of the program that you will eventually write deals with the assumptions, exceptions, and variations that you will need to identify.

Outline (Pictorial)



About These Slides

The full set of slides requires 2+ hours, when you factor in my jokes & your questions, ☺ !

For this talk, I have only *X* hours; thus, the set of slides is trimmed.

You will see evidence of trimming in the logic jumps.

You may find all trimmed sets & the full set at cs.uwaterloo.ca/˜dberry/FTP_SITE/lecture.slides/HistoryOfMe_SE_FMs_RE/

Foreword

Please note that I *believed* in FMs.

I used them and still occasionally still use lightweight versions of them.

My Criticisms Are For Me

When I criticize some*thing*, I am explaining what I observed that informed my own choice of what to work and to spend my *precious* time on.

I know that I may be wrong.

Therefore, I never criticize or disrespect another *person* for observing differently and choosing to work on what I don’t work on.

Who knows, you might make a discovery that changes everything.

Vocabulary

CS = Computer Science

CBS = Computer-Based System

SW = Software

PL = Programming Language

FM = Formal Method

SE = Software Engineering

EP = Electronic Publishing

RE = Requirements Engineering

My 1960s Start in Computing

- **wrote my first real-life application, Operation Shadchan, a party 1-1 matching program based on the questionnaire of Operation Match, a 1-*n* dating program, in the Spring of 1966, age 17, for my synagogue’s youth group’s annual party,**

SOTP BIAFIUIW

Through all this, I did seat-of-the-pants build-it-and-fix-it-until-it-works (SOTP BIAFIUIW) SW development, …

simultaneous RE, design, and coding, …

not really understanding the distinction between RE, design, and coding, …

SOTP BIAFIUIW, Cont’d

thinking that all of it were just parts of programming, …

probably like a whole lot of programmers, even professionals, did.

Security, Cont'd

I consulted for the Formal Development Method (FDM) group of SDC ($\rightarrow$ UNiSYS) that was working on secure operating systems, e.g., Blacker.

I ended up publishing a paper in *IEEE TSE* showing how the theorems that the group's verifier proved about an Ina Jo formal specification of a system were sufficient to prove that the system, if implemented as specified, would meet the specified criteria.

Security, Cont’d

From all this work and from its community that included such people as Peter Neumann, I learned a lesson that goes right to the essence of RE:

There is no way to add security to any CBS after it is built; the desired security must be *required from the beginning* so that security considerations permeate the entire development lifecycle.

Importance of Ignorance in RE

RE

So in 1994, I published “The Importance of Ignorance in RE” claiming that every RE team for a CBS requires along with domain (of the CBS) experts at least one smart ignoramus of the domain, who will

- **provide out-of-the-box thinking that leads to creative ideas, and**
- **ask questions that expose tacit assumptions.**

A Realization

Then, a subset of the SE field came to the realization that the real problem plaguing CBS development was that we did not understand the requirements of the CBS we are building.

A Realization, Cont’d

Brooks, in 1975, had said it well:

“The hardest single part of building a software system is deciding precisely what to build…. No other part of the work so cripples the resulting system if it is done wrong. No other part is more difficult to rectify later.”

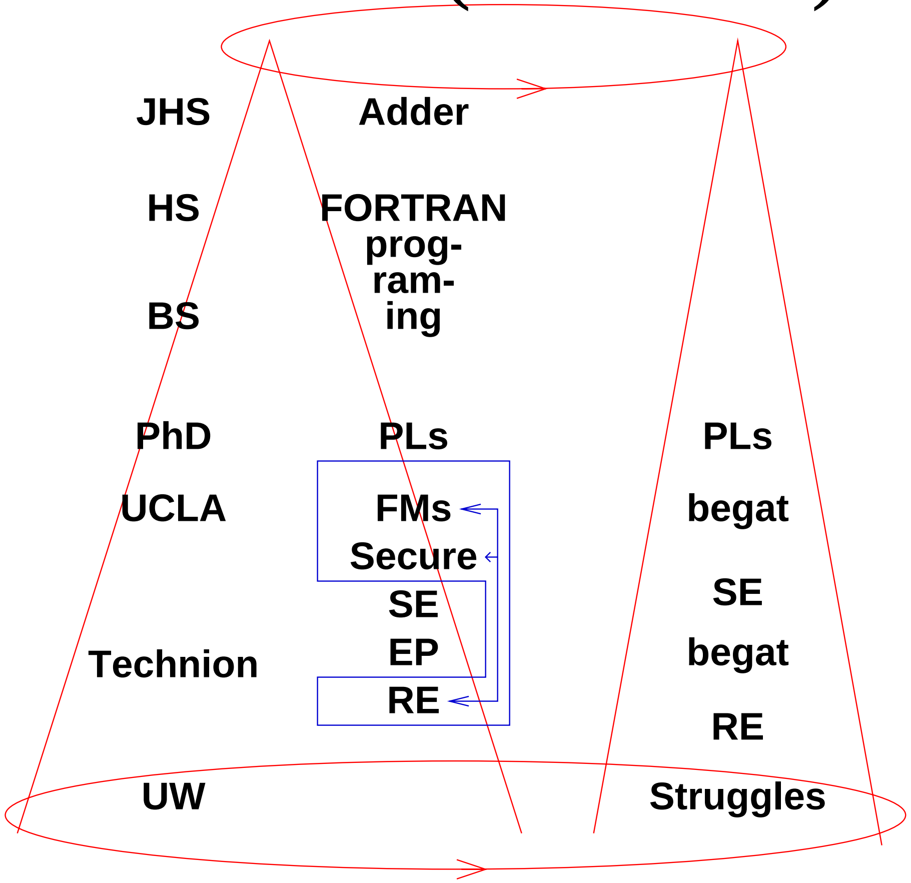
Even a FMs Person Got it

Even an initial-algebras, FMs person, Joe Goguen, came to this realization.

He ended up being a keynoter at the first RE conference in 1993.

Fast Forward

Outline (Pictorial)



Motivation to Write These Slides

I am occasionally asked to referee a FMs paper, and

I occasionally hear a FMs talk.

Motivation, Cont’d

I am struck by how little has changed from 1970s. I read or get a sense of:

- **Here’s a new approach to formalize *X*. (*X* is the same as in 1970s)**
- **If only developers would listen to us!**
- **We’re on the verge of a breakthrough that will convince developers to use FMs.**

It seems to be all the same as in the 1970s and 1980s.

Motivation, Cont’d

However, what seems to be is not reality!

There *have* been advances, e.g.,

> **SAT provers,**
> **refutation,**
> **model checking,**
> **domain-specific formalizations**
> **alloy,**
> **etc.**

Motivation, Cont'd

But, each of these advances suffered the

silver bullet $\rightarrow$ aluminum bullet

phenomenon.

Always Writing SW

I was always writing software for real-world applications:

- **medium-sized CBSs by myself or with or by my students, and**

- **large-sized CBS as part of a team**

Such as

- **matchmaking for a party (before knew about FMs)**
- **tools for regression analysis for chemists (before knew about FMs)**
- **bi-directional formatter**
- **proof updater for FDM suite of FM tools**
- **bi-directional editor**
- **tri-directional formatter**
- **letter stretching bi-directional formatter**

Never Actually *Used* FMs

I never even *considered* using FMs to develop any *real* SW …

even for the proof updater for the FDM suite of FM tools.

Knowing what I knew about developing these systems, I would have been crazy to.

But, I did use my FM-based skills of abstraction and modeling to my advantage.

Never *Used* FMs, Cont'd

FM

Neither did Val Schorre and John Scheid in developing the other tools for the FDM suite, including a verification condition generator (VCG) for Ina Jo specs, and an interactive theorem prover (ITP).

(They did use Val's compiler-compiler to deal with the syntax.)

Never *Used* FMs, Cont'd

Note that these tools *were* used in production applications of the FDM to building some half dozen verifiably secure systems at SDC for the US DOD and NSA.

Never *Used* FMs, Cont’d

Apparently, neither did other developers of FM tools (at least the ones I knew).

This seemed to be one of the dirty, dark secrets among FM tool builders.

No one in his right mind would consider *using* FMs to build these tools.

The perception was that it would just take too long, and they might never finish.

FMs For Only Small Programs

So, FMs could be used only for the development of *small* programs.

Operating system kernels and trusted system kernels *are* small programs.

So some FMers began a push to get all programs to be small!

Hoare on Small Programs

Tony Hoare said (I think in late 1970s through 1980s),

“Inside every large program is a small program struggling to get out.”

I got in to the habit of trying to identify the central algorithm, the small program, at the heart of each of my programs.

Having done so, still the program was messy and the programming was hard.

Matchmaker

I did this while I was in HS, long before I knew about FMs.

Later, it proved to be a variation of the stable marriage problem, with a 50-factor bi-directional attractiveness function, based on questionnaire answers.

In retrospect, the central formal model would have accounted for less than 5% of the code.

Matchmaker, Cont’d

The rest of the code deals with

- **incorrectly filled questionnaires,**
- **the complexities of having a mix of absolute criteria and do-the-best-that-you-can criteria, and**
- **having to deal with too-picky people who did not get matched by the algorithm, but still had to be matched for the party they paid for.**

Back to the FDM ITP

In retrospect, I can see why FMs were not used to develop the ITP.

The central, formal part of the ITP was a small fraction of its code.

Back to the FDM ITP, Cont'd

The rest dealt with

implementing the really nice interaction with the user (the person trying to prove a theorem)

managing the current proof, including keeping track of what had been proved in a way that made it easy for a user to apply any of it at any time, …

and this part is tough to formalize.

What vs. How Specifications

Many times, it is much easier to express an algorithm to do something than to give an algorithm-independent description of what the something is:

- **industrial processes**
- **exceptions to a central algorithm**
- **New York bagels (chewiness vs boil-then-bake)**

Failings of FMs

Even as FMs applied to Security taught me the fundamental essence of RE,

FMs have proved incapable of

- **dealing adequately with the kinds of CBSs that we need to build, and**
- **doing what we need to do in RE.**

We explore why.

FMs Not Deal With CBSs That We Build

Let’s see what Tony Hoare says.

Tony Hoare's Reversal, Cont'd

"Ten years ago, researchers into formal methods (and I was the most mistaken among them) predicted that the programming world would embrace with gratitude every assistance promised by formalisation to solve the problems of reliability that arise when programs get large and more safety-critical. Programs have now got very large and very critical — well beyond the scale which can be comfortably tackled by formal methods.

Tony Hoare’s Reversal, Cont’d

There have been many problems and failures, but these have nearly always been attributable to inadequate analysis of requirements or inadequate management control. ***It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve.*** **[Italics are mine]”**

Hoare on Small Programs

Tony Hoare once said (in mid 1970s),

“Inside every large program is a small program struggling to get out.”

Later (in early 2000s) he added,

“the small program can be found inside the large one only by ignoring the exceptions.”

Now I Understand

Now I understand that what I was observing about the distribution of code is normal.

Distribution of Code

10–20% of the code = central approximation.

80–90% of the code = exceptional details.

99.99% of execution time is spent in the central 10–20% of the code.

It’s hard to test the exceptional details code, the 80–90% of the code, because it gets executed less than 0.01% of the execution time.

And correcting a defect in this 80-90% of the code likely involves changes in related code scattered all over the code.

FMs Not Doing What RE Needs

RE concerns validation more than verification, …

but FMs deal with …

Verification, but …

FMs have the power to put

verifying the correctness of a CBS implemention w.r.t. its specifications

on a much firmer basis than is possible with

testing the CBS w.r.t. its specifications with well-chosen test data.

…, but Not Validation

However, this power does *very little* towards

validating the specifications w.r.t. its customer’s needs and wants,

i.e., its customer’s requirements.

And Here’s Why

The next bunch of slides are about what has become known as the Reference Model for Requirements and Specifications by Gunter, Gunter, Jackson, and Zave,

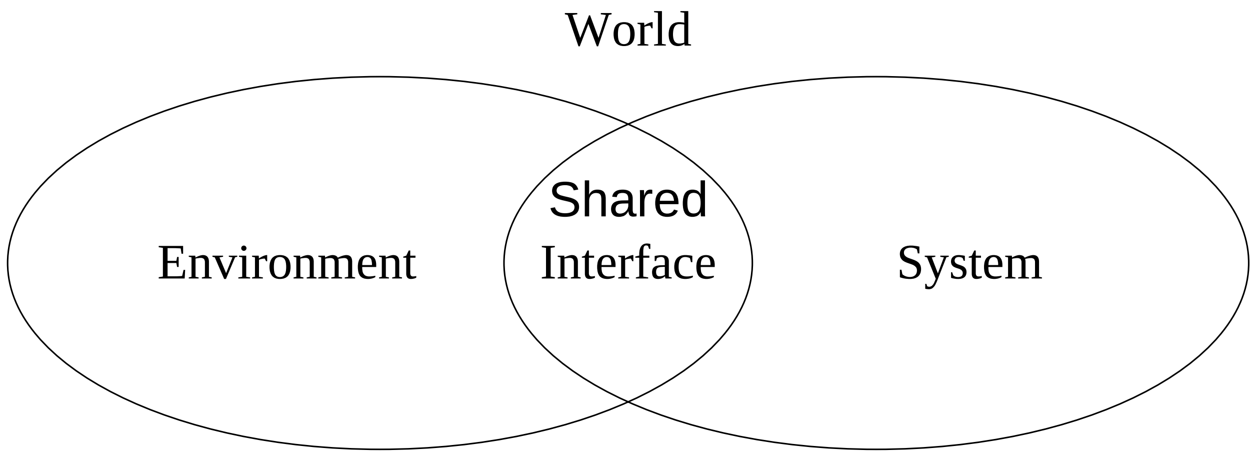
or the RE Reference Model.

The World and the CBS

The world in which a CBS operates is divided into

- **an Env, the environment affecting and affected by the CBS, and**
- **a Sys, the CBS itself, that intersect at their**
- **Intf, their Interface, and**
- **the rest of the world.**

The World and the CBS



Not Precise

While Sys, the CBS, is formal (mathematical),

the rest of the world, including Env, is *hopelessly in*formal,

and the boundaries of Env are *hopelessly fuzzy*:

Butterfly in Rio $\rightarrow$ Golden Gate Bridge

So finding all details to not ignore is hard.

Famous Validation Formula

The informality has been made formal in the Zave–Jackson Validation Formula (ZJVF):

$D,S \vdash R$

D **Domain Assumptions, in Env, informal**
S **System Spec, in Intf, can be formal**
R **Requirements, informal, in Env, informal**

Truth of each of ***D*** **and** ***R*** **in Env is** ***empirical*****.**

Sys Spec Formal?

RE

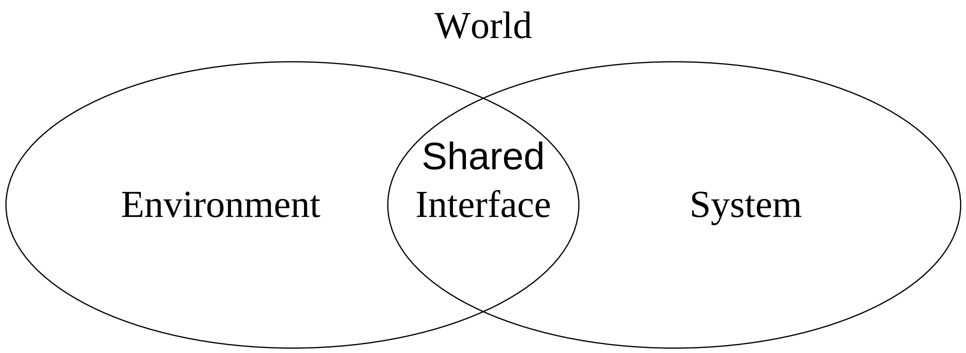
S **is formal, if it is about a program written in a PL.**

If program is molecular, then even *S* is informal, and its truth is empirical.

If program uses machine learning, then *S* is effectively informal, and its truth is dependent on the learning set in ways that defy formalization.

Where Are the Exceptions?

From where is that 80–90% of the code = exceptional details?



From the Env, but not from the outside World!

But are we sure that it's not from the outside World? Well... um...

Example: Airplane

Sys = airplane

Env = the sky

World = everything not relevant

Are the following in the Env:

- **flying bird?**
- **something in the hand of someone on the ground?**

The boundaries of Env are *hopelessly fuzzy*.

Two Types of Requirements

There are two types of requirements:

1. **scope determining**
2. **scope determined**

E.g., for a pocket calculator with $+$, $-$, $\times$, $\div$,

1. $\ln$ **and x^y, are scope-determining requirements.**
2. **"that $d \neq 0$ in $n \div d$" is a scope-determined requirement.**

Difference Between Types

A pocket calculator without one particular scope determining requirement is just a less useful and less attractive calculator.

A pocket calculator without one particular scope determined requirement is a flawed calculator, which will give the wrong result or fail for some inputs.

FMs and the Two Types

FMs help discover scope-determined requirements.

FMs offer little help discovering scope-determining requirements, …

because each scope-determining requirement is independent of the others.

“If no one happens to think of it, it just ain’t gonna be there.”

Value of RE Reference Model

The RE RM has become extremely valuable as a …

lightweight, informal version of a FM …

that is able to answer many questions that come up during RE for a CBS.

Value of RE RM, Cont’d

The RE RM is used to help

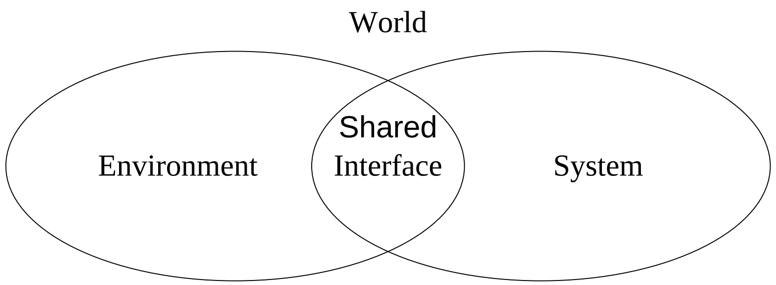
- **partition the World, i.e., to decide for each of Env, Intf, and Sys, what is in it and is not, …**

  sometimes to shuffle an entity among Env, Intf, and Sys

Value of RE RM, Cont'd

- **decide *What* vs. *How*:**

  ***What* is in the vocabulary of Env**
  ***S* is in the vocabulary of Intf**
  ***R* is in the vocabulary of Env**
  ***How* is in the vocabulary of Sys–Intf**



Value of RE RM, Cont’d

- **permanently tolerate an inconsistency *I* between *R* and *S* and the World,**

  by lying in *D* that *I* is not a problem, …

  e.g., for the Airplane CBS, permanently tolerate that a bird’s meeting an airplane in the air can crash the airplane, by lying in *D* that there are no birds in the air.

Programming as a FM

Programming itself is a FM in the sense that writing a formal specification is a FM!

Remember that programming is building a theory from the programming language and library of abstractions (the ground) up, just like making new mathematics.

But there are some fundamental differences between a program and a math model, as it’s usually done.

Math Model vs. Program

Each is a model of the real world.

Different audience:

- **math model read by smart human; can deal with “YUWIM”**
- **program read by dumb computer; cannot deal with “YUWIM”**

Math vs. Program, Cont'd

Because of difference in audience,

- **math model can get away with simplifications and approximations for tractability;**

- **program must deal with every detail, with *no approximation*, or else program fails at exception conditions, e.g., plane crashes.**

Fickas on Outliers

Steve Fickas once said,

“Sciences ignore outliers.”

But, robust software cannot.

Central Math Model in Code

In a program based on a mathematical model of some real-world phenomenon, …

the mathematical model amounts to 20% of the code, and the code to deal with the outliers, the approximations, the exceptions, etc. amounts to 80% of the code.

Code as Math Model

So, code is a much more complete mathematical model than most mathematical models produced by mathematicians or scientists.

Even then, as we saw with the World Model and the ZJVF, it cannot be a *perfect* model.

What *Does* Work?

Good people, not good methods!

Success Stories of FMs

The typical success story describes a FM person convincing a project to apply some particular FM.

The deal is that the FM person joins the team and either does or leads the formalization effort.

Success Stories, Cont’d

The reported experience shows the FM person slowly learning the domain from the experts by asking lots of questions and making lots of mistakes.

The end result is that the application of the FM found many significant problems earlier and the whole development was cheaper, faster, etc. than expected.

Real Value of FMs

Perhaps the real value of FMs is that they attract really good people, the FMers, who is good at dealing with abstractions, who is good at modeling, etc., the smart ignoramus, into working on the development of your CBS.

Managers know that the success of a CBS development project depends more on personnel issues than on technological issues.

Flawed Experiment

“Formal Methods Application: An Empirical Tale of Software Development”, by Ann E. K. Sobel and Michael R. Clarkson, *IEEE Transactions on Software Engineering* 28:3, 157–161, March 2002

Attempt to empirically prove the effectiveness of FMs in producing quality software.

FMs vs. No FMs

They arranged two groups of teams of university students

Each team in group number

1. **learned FMs and used them in a term-long project to develop a program**

2. **did not learn FMs and did term-long project to develop same program**

Results

1. **100% of programs produced by FM teams passed all of a set of 6 test cases.**

2. **Only 45.5% of programs produced by nonFM teams passed all of same set of test cases.**

Wow!!

Conclusions

Sobel and Clarkson’s Conclusions:

Since teams did not differ by all sorts of academic measures, the successes were due to the use of FMs

Wrong!

Walter Tichy and I independently spotted the flaw in the experiment (We ended up writing a joint note).

Voluntary Selection!

Only students who had voluntarily taken an optional course on FMs were in FMs teams.

NonFM teams consisted of only students who had *not* taken this FMs course.

No Control

Also, there was *no* control over whether the FM teams actually *used* FMs in the development.

Might be that the FM teams took advantage of skills, e.g., abstracting, logical thinking, etc., used in FMs, to improve their programming without actually doing any FM.

Not enough information to know.

Alternative Explanation

Berry and Tichy offered an alternative theory for results:

The reason for the success was presence of the people who were interested in, and presumably skilled in, in FMs, abstract thinking, etc.

They program better naturally!

Alternative …, Cont’d

The teams consisting of FMs users, whose programs passed all the tests, were just plainly and simply *better programmers* than the teams not containing any FMs users, whose programs did not pass all the tests.

No surprise there!

Lesson Learned

Good FMers make good programmers.

So if you're managing a SW development, hire FMers to be your programmers!

My Message to FMers

Forget about proving programs, i.e., code, correct; it's not cost effective:

- **it increases development cost by an order of magnitude;**
- **only 15–25% of all errors are introduced by coding; and**
- **numerous experiments show that inspection does a good job of eliminating coding errors for only 15% overhead.**

My Message, Cont'd

Focus on getting correct & complete requirements specs, where 75–85% of the errors occur:

- **FMs applied to make the specs more correct, i.e., to eliminate errors of commission & discover missing scope determin*ed* requirements**
- **FMer applied to make the specs more complete, i.e., to eliminate errors of omission & discover new scope determin*ing* requirements**