

# The Inevitable Pain of Software Development: Why There Is No Silver Bullet

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

# The Inevitable Pain of Software Development: Including of Extreme Programming, Caused by Requirements Volatility

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

## Abstract

A variety of programming accidents, i.e., models, methods, artifacts, and tools, are examined to determine that each has a step that programmers find painful enough that they habitually avoid or postpone the step. This pain is generally where the programming accident meets requirements, the essence of software, and their relentless volatility. Hence, there is no silver bullet.

## Terminology

This talk is about building computer-based systems (CBS).

The most flexible component of a CBS is its software (SW).

We often talk about developing SW, when we are really developing a whole CBS.

“SW” and “CBS” are used interchangeably unless the distinction matters.

## More Terminology

**We talk about methods, approaches, artifacts, and tools as technology that help us develop CBSs. I use “method” to stand for all of them so I don’t have to keep saying “method, approach, artifact, or tool” in one breath.**

## Talk Based on Paper

**This talk is based on a full paper of the same title available at**

[http://se.uwaterloo.ca/~dberry/FTP\\_SITE/tech.reports/painpaper.pdf](http://se.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/painpaper.pdf)

**All the details and citations left out of this talk can be found there!**

## If You Disagree?

**What should you do if, by some strange occurrence, you disagree with something or everything I say?**

**Please be polite and allow me to finish my talk and make my point.**

## After the Talk

**After the talk, during questions and answers, you can demolish my argument to your heart’s content, or ...**

**you can send me e-mail at [dberry@uwaterloo.ca](mailto:dberry@uwaterloo.ca), or ...**

**you can write your own paper!**

## Personal Observation

**This talk is based on personal observation.**

**Sometimes, I describe an idea based solely on my own observations over the years.**

**These ideas have no citations.**

**These ideas have no formal or experimental basis.**

**If your observations differ, then please write your own rebuttal.**

## Beliefs and Feelings

**Sometimes, I give as a reason for doing or not doing something that should not be or should be done what amounts to a belief or feeling.**

**The belief or feeling may be incorrect, i.e., not supported by the data.**

**The belief or feeling is typeset in Italics**

## Programming Then and Now

**I learned to program in 1965.**

**First large program outside classroom for a real-life problem was written in 1966 in FORTRAN!.**

## Operation Shadchan

**I implemented functionality of Operation Match, adapted to use by a high school synagogue youth group dance.**

**The dance and the SW were called “Operation Shadchan”.**

**Each person’s date for the dance was selected by the SW.**

## I Remember

I remember doing requirements analysis at the same time as I was doing the programming in the typical seat-of-the-pants build-it-and-fix-it-until-it-works (BIAFIUIW) method of those days:

## BIAFIUIW Method

- discover some requirements,
- code a little,
- discover more requirements,
- code a little more,
- etc, until the coding was done;
- test the whole thing,
- discover bugs or new requirements,
- code some more, etc.

## Biggest Problem

The biggest problem I had was remembering all the requirements.

It seems that ...

each thought brought about the discovery of more requirements.

## More Requirements

They were piling up faster than I could modify the code to meet the requirements.

I tried to write down requirements as I thought of them.

## Forgotten Requirements

**But, in the excitement of coding and tracking down the implications of a new requirement, which often included more requirements, I neglected to or forgot to write many down, only to have to discover them again or to forget them entirely.**

## Guilt

**I recall feeling guilty just thinking, about requirements, rather than doing something substantial, writing code.**

**So whenever I considered requirements because I could go no further with coding, I tried to do it as quickly as possible.**

## Programming Felt Like Skiing

**Programming felt like skiing down a narrow downhill valley with an avalanche following me down the hill and gaining on me.**

**Programming gave rise to an endlessly growing avalanche of endless details.**

## Overwhelming Problem

**We have a sense of being overwhelmed by the immensity of the problem and the seemingly endless details to take care of, and we produce poorly-written software that makes stupid mistakes.**

## Nowadays

**Nowadays, we follow more systematic methods.**

**My latest program to implement stretching of Arabic letters was constructed, after extensive requirements analysis and architecture recovery, by making object-oriented and aspect-oriented extensions to a legacy program constructed using information hiding.**

## However

**However, programming still feels like skiing just ahead of an avalanche.**

**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 kind of poorly-written software that makes the same kind of stupid mistakes**

**as 35 years ago!**

## Others Too

**These feelings are not restricted to me.**

**I see other programmers undergoing similar feelings.**

**I have seen many people nod in agreement in previous presentations of this talk!**

## No Matter How Much We Try

**No matter how much we try to be systematic and to document what we are doing, we forget to write things down, we overlook some things, and the discoveries of things seems to grow faster than the code.**

**What are these “things”?**

**They are mostly requirements of the CBS that we are building.**

## The Real Problem of SE

The real problem of SE is dealing with ever changing requirements.

It appears that no

- model,
- method,
- artifact, or
- tool

offered to date has succeeded to put a serious dent into this problem.

## Others Agree

I am not the first to say so:

Fred Brooks and Michael Jackson, among others, have said the same for years.

## No Silver Bullet

Fred Brooks, in saying that there is no SE silver bullet (1987), classified SW issues into

- the essence and
- the accidents.

## Essence vs. Accidents

The essence is what the SW does, the requirements.

The accidents are the technology by which the SW does the essence or by which the SW is developed, the language, tools, and methods used to implement the essence.

## Requirements are Hard

**Brooks says, “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.”**

## Requirements for Methods

**This quotation captures the essential difficulty with SW that must be addressed by any method that purports to**

- **alter fundamentally the way we program,**
- **to make programming an order of magnitude easier, and**
- **to be the silver programming bullet we have been looking for.**

## Some Improvement

**No single method (accident) has put a dent into this essential problem.**

**Although all of the improved methods have combined to improve programming by at least an order of magnitude since 1965.**

**However, as programming has improved, we have taken on even more ambitious essences, leaving no net gain in our ability to deal with essences.**

## Brooks Says ...

**Moreover, says Brooks based on his experience, the silver bullet will probably never be found:**

## Will Probably Never Find SB

***“I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared to conceptual errors in most systems. If this is true, building software will always be hard. There is inherently no silver bullet.”***

## Requirements Change!

**Michael Jackson, in his ICRE'94 Keynote, said that two things are known about requirements:**

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

## Thus ...

**Thus, (1) a CBS will always have to be changed, to accommodate its changed requirements.**

**And, (2) a CBS's requirements will always have to be changed, to accommodate better understandings as misunderstandings are discovered.**

**We will see a third source of change later!**

## Understanding is Difficult

**Paul Clements & David Parnas (1986) describe how difficult it is to understand everything that might be relevant.**

**“Even if we know X, we may not know Y.”**

**(“Even if we know Y, we may not know Z.”)<sup>+</sup>**

**(Therefore, it's OK to fake having proceeded with no back up when writing documentation of your development. ☺)**

## Type E Systems

Meir Lehman classifies a CBS that solves a problem or implements an application in some real world domain as an E-type system.

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

So there is *no* hope of getting ahead of requirements.

## Most Changes are Requirements Changes

Not all changes to a CBS are due to requirement changes.

But, as early as 1978, Bennett Lientz and Burton Swanson found that 80% of maintenance changes deal with requirements changes.

## Decay of Software

As early as 1976, Laszlo Belady and Meir Lehman observed the phenomenon of eventual unbounded growth of errors in legacy programs that were continually modified in an attempt to fix errors and enhance functionality.

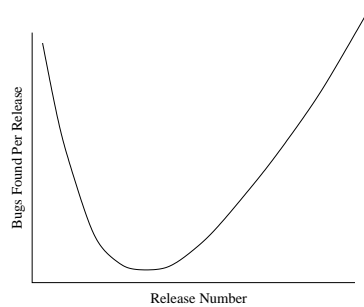
## Decay of SW Cont'd

When a change is made, it's hard to find all places that are affected by the change.

So any change, including for correcting an error, has a non-zero chance to introduce a (new) error!

## Belady-Lehman Graph

They modeled the phenomenon mathematically and derived a graph:



## B-L Graph, Cont'd

In practice, the curve is not as smooth as in the figure; it's bumpy with local minima and maxima.

It is sometimes necessary to get very far into what we will call *Belady-Lehman (B-L) upswing* before being sure where the min point is.

## Min Point

The min point represents the software at its most bug-free release.

After this point, the software's structure has so decayed that it is very difficult to change anything without adding more errors than have been fixed by the change.

## Freezing SW

If we are in the B-L upswing for a CBS, we could roll back to the best version, at the min point.

Declare all bugs in this version to be features.

Usually not changing a CBS means that the CBS is dead; no one is demanding changes because no one is using the software any more.

## Exceptions to Death

However, many old faithful, mature, and reliable programs have gone this way, e.g.:

- cat, and other basic UNIX applications,
- vi, and
- ditroff

Their user communities have grown to accept, and even, require that they never change.

## Non-Freezable Programs

**IF**

- the remaining bugs of best version are not acceptable features, or
- the lack of certain new features begins to kill usage of the CBS

**THEN** a new CBS has to be developed from scratch

- to meet all old and new requirements,
- to eliminate bugs, and
- to restore a good structure for future modifications.

## Another alternative

Use best version of legacy program as a feature server.

Build a nearly hollow client that

- provides a new user interface,
- has the server do old features, and
- does only new features itself.

## Tendencies for B-L Upswing

The more complex the CBS is, the steeper the curve tends to be.

The more careful the development of the CBS is, the later the min point tends to be.

## Occasionally

Occasionally, the min point is passed during the development of the first release, as a result of

- extensive requirements creep that destroyed the initial architecture, or
- the code being slapped together into an extremely brittle CBS built with with no sense of structure at all.

## Purpose of Methods

One view of software development methods:

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

That is, the method tries

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

## Information Hiding

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

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

## IH, Cont'd

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

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

∴, B-L upswing is tamed!

## Why Is There No Silver Bullet?

**Why is there no silver bullet and why can there not ever be a silver bullet?**

**Because of the inevitable pain of SW development methods!**

## The Inevitable Pain of Methods

**My contention:**

**Every time a new method that is intended to be a silver bullet is introduced,**

**it does make many accidents of software engineering easier.**

## Methods Do Work!

**In fact, each method, if followed religiously, works.**

**Each provides a way to manage complexity and change so as to delay and moderate the B-L upswing.**

**Sometimes, just following the method *religiously* is the pain.**

## Pain Sets In!

**However, as soon as a part of the method needs to deal with the essence and its changes, suddenly the method becomes painful, distasteful, and difficult,**

**so much so that this part of the method gets postponed, avoided, and skipped.**

**Therefore, the method ends up being only slightly better than no method at all in dealing with essence and change borne difficulties.**

## The Methodological Catch 22

Each method has a catch, a fatal flaw, at least one painful step that people tend to put off.

People put off doing this painful step in their haste to get the SW shipped out or to move on to more interesting things, like writing new code.

Consequently, the SW tends to decay no matter what. The B-L upswing is inevitable.

## Pain in IH

OK, so having said this, what *is* the pain in IH?

IH's success in making future changes easy depends on having identified a right decomposition, i.e., one that isolates an implementation change into one module.

If a new requirement comes along that causes changes that bridge several modules (cross-cutting concerns), these changes might very well be harder than if the code were more monolithic.

## Pain in IH, Cont'd

It is easier for tools to search within one, even big, module than in several, even small, modules.

Future changes, especially those interacting with the new requirement, will likely cross module boundaries.

Consequently, it is really necessary to restructure the code into a different set of modules.

## Pain in IH, Cont'd

This restructuring is a major pain:

- it means moving code around,
- writing new code, and
- possibly throwing out old code

for no apparent change in functionality.

It gets put off in the rush to deliver the next version.

B-L upswing sets in!

## Irony

The painful restructuring is necessary because the module structure no longer hides all information that should be hidden.

The requirements changes have caused some implementation information that should be hidden,

- to be scattered over several modules and
- to be exposed from each of these modules.

## Irony, Cont'd

IH failed to protect against these changes because they were *requirements* changes and not implementation changes.

The painful restructuring being avoided are those necessary to restore implementation IH, so that future implementation changes will be easier.

## Irony, Cont'd

Without the painful changes, all changes, both implementation and requirements-directed, will be painful.

## Methods and Their Pains

I examine a number of of models, methods, and tools to identify their painful steps.

I give just a representative sampling of models, methods, and tools.

See my paper of the same title for a larger sampling.

## Fatal Flaw

I have become convinced that each method that deals with the entire CBS development has its painful step, its fatal flaw, when it comes to dealing with requirements changes. I call these total methods.

The same can be said for some but not all methods that deal with specific parts of CBS development. I call these partial methods.

## Coverage

Covered are:

- Build-and-Fix Model,
- Waterfall Model,
- Structured Programming,
- Requirements Engineering,
- Extreme Programming,
- Rapid Prototyping,
- Formal Methods,

## Coverage, Cont'd

- Inspection,
- Regression Testing,
- Daily Build,
- Open Sourcing,
- Documentation, and
- Tools and Environments.

## Coverage, Cont'd

With each, I indicate whether the method is applied to the total lifecycle or to parts of it.

With each, I give its fatal flaw (FF), the step that is a pain to do and that keeps getting put off.

## Build-and-Fix Model

**Total**

**FF: the first overlooked requirement**

**The B-L upswing can set in during first iteration.**

## Waterfall Model

**Total**

**FF: the whole model**

**OR: in any circumstance in which the WF cannot be followed, and heavy backtracking sets in**

## Structured Programming

**Total**

**FF: redoing SP from initial abstract statement of the whole problem**

**OR: faking it properly with *in situ* patches in all places affected by change**

## Requirements Engineering

**Partial**

**FF: finishing RE before going on to design and coding**

**OR: delaying design and coding until after RE is done**

## Extreme Programming

**Total**

**FF: refactoring**

## Rapid Prototyping

**Total or partial**

**FF: scrapping throw-away prototype to start all over**

## Formal Methods

**Total**

**FF: FM itself**

**OR: dealing with changed requirements**

## Inspection

**Partial**

**FF: inspection itself in face of impending deadlines**

## Regression Testing

**Partial**

**FF: full testing in the face of small change and impending deadlines**

## Daily Build

**Partial**

**FF: fixing the system if your change breaks the system at the next build**

## Open Sourcing

**Total**

**FF: reconciling all the slightly conflicting, but nevertheless useful and cool versions coming in from the rest of the world without insulting anyone who contributed**

## Documentation

**Partial**

**FF: keeping documentation up to date in the face of relentless changes and impending deadlines**

- **Waterfall**
- **Structured Programming**
- **Information Hiding**
- **Tracing**
- **Requirements Specification**

## Tools and Environments

### Partial

**FF: remembering to use them because of the work to patch things up after failure to use tools for one or more iterations of the software**

**It's easy enough to forget to check out a module that you want to make just a small change to, and merging independently changed modules is made more difficult.**

## Tools and Environments, Cont'd

- **Configuration Management**
- **Version Control**

## Conclusions—No SB

**It appears that there is no SE silver bullet.**

**All SE bullets, even those that contain some silver, are made mostly of lead.**

**It is too hard to purify the painful lead out of the real-life SE bullet to leave a pure painless silver SE bullet.**

**This observation applies even to XP, which tries to avoid painful processes entirely.**

## Like Slapstick Drawers

**Situation with SE methods is not unlike that stubborn chest of drawers in the old slapstick movies.**

**A shlimazel pushes in one drawer and out pops another one, usually right smack dab on his knees or shins.**

**If you find a new method that eliminates an old method's pain, the new method will be found to have its own source of pain.**

## Deal with Changes

**There cannot be any significant change in programming until we figure out how to deal, with a lot less pain, with the relentless change of requirements and all of its ripple effects.**

## CBS Development is an Art

**Perhaps, we have to accept that CBS development is an art and that no amount of systematization will make it less so.**

## Well Known Domains

**If we know a domain so well that production of software for it becomes almost rote, as for compiler production these days, we can go the engineering route for that domain, to make building software for it as systematic as building a bridge or a building.**

## New Domains

**However, for any new problem, where we have the excitement of innovation, there is no hope of avoiding**

- **relentless change as we learn about the domain,**
- **the need for artistry, and**
- **the pain.**

## It's about Maturity

The key concept in the Capability Maturity Model (CMM) is *maturity*.

Getting the capability to do the recommended practices is not a real problem.

The real problem is getting the maturity to stick to the practices despite the real pain.

