

In Requirements Engineering (RE), Everything is an Aspect

Daniel M. Berry
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada

Confession

Before being invited to give this talk, I had only a superficial acquaintance with aspect orientation (AO).

It was not that I thought it was not important; rather, I have only so much time to keep up with research, reading, etc.

RE takes up all my time, and I don't have time for many other good subjects, not just AO.

And good other people are doing good things in AO.

I did some research

I did read up a bit on aspect-oriented programming (AOP), mostly introductory papers, such as Kiczales *et alii* 1997 and the current *Wikipedia* page, which seems OK, mainly because it references what appear to be seminal papers.

I am not an expert

I do not claim that preparing this talk has made me an expert. I am quite sure that you will identify many issues about which I am totally ignorant.

I was assured that my lack of expertise was OK for this talk, that you wanted someone from outside of the area to give a fresh perspective.

I am willing to learn

Even though I am giving this keynote talk, I *am* willing to be corrected, particularly when I have made a mistake.

I am even willing to adjust my opinions in the face of compelling evidence that I have overlooked.

But humor me enough to let me make my point 😊 .

My Background

I have been programming since 1965, when I learned how to program in FORTRAN.

I can see now that I had faced the problem of cross-cutting concerns from almost the very beginning, from the first time I had to change code I had written as a result of discovering new requirements ...

for whose implementation the code developed so far turned out to be poorly structured!

Early Aspects

Only, I did not call them cross-cutting concerns back then.

I called them “f---ing pains in the tukhis”.

Vocabulary

“Aspect” seems to be used in the literature in two different ways:

- 1. = “cross-cutting concern (CCC)”**
- 2. = “technique to deal with CCCs in existing software (SW)”**

I dislike unnecessary ambiguity.

I will use “CCC” for the first and “aspect” for the second.

Thanks Hermann Kaindl!

Epiphany

**I now realize that the title of my talk should be
“In RE, Everything is a Concern” ...**

**and that the published abstract got it more
right than the title! Thanks Ruzanna and
Monica!**

In Requirements Engineering (RE), Everything is a Concern

Daniel M. Berry
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada

More Epiphany

I realized also that CCCs are part of what I have described as the inevitable pain of software (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 (in Fred Brooks's sense of the word) of SW development 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 minimize the decay that tends to set in on oft-changed code.

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 (in Brooks's sense of the word) 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.

Origin of Some Slides

Some of the slides herein, such as the last 4, are stolen from my talk on that inevitable pain.

Realization

I now realize that the source of some of this pain was the CCCs introduced by some of the changes.

My First Encounter with CCCs

My first encounter with CCCs came during my first non-classroom-exercise program, i.e., my first program written for a real-world application, a match-making program, to match dates for a dancing party!

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.

Present Realization

I realize now that each new requirement for Operation Shadchan caused a CCC because the implementation of the new requirement did not fit the decomposition implicit in the code I had written.

Implications of Realization

Note that “requirement” \equiv “concern”, otherwise a requirement’s implementation could not become cross cutting (CC).

Note also that the decomposition was only implicit, because I wrote code linearly, from the bottom upward, rather than from the top downward, concern inward!

Looking Back

Looking back over all the code I have written, I see the same phenomenon, even though my programming methods improved.

I began to understand decomposition in the 70s and 80s, e.g.,

- **structured programming (SP)**
- **information hiding (IH)**
- **object orientation (OO)**

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.

The decaying of code is delayed and moderated.

No CCC is introduced!

Programming with Abstractions

After I learned about decomposition, I was programming from abstractions inwards, i.e.,

- 1. identify abstractions first, and**
- 2. implement each abstraction somewhat independently of others.**

No CCCs First Time Around

At any time, I built abstractions based on the requirements I knew, there generally were no CCCs the first time around ...

the so-called loggers and error handlers that are used in the AOP literature as examples of CCCs, notwithstanding.

These are CC, but were tolerable, because ...

in a sense, the CCC was the abstraction, and it was never more than one level.

But With New Requirements

But then along came a new requirement, the devil!

Sometimes, I was lucky. Its implementation was not CC.

More often than not, its implementation was CC! Oy!

(All this without knowing the term “CCC”!)

Should Have Redesigned

I realized then that I should have redesigned the implementation from scratch, ...

but that's *PAINFUL!*

Why throw away a perfectly good running implementation?

The key is that the implementation is *not* really perfect; it does not have the new requirement.

Patching

My typical response was to patch, increasing code brittleness until eventually ...

patching or its aftermath was more painful than the perceived pain of redesign from scratch.

We now call this redesign “restructuring”.

Pain in IH and Restructuring

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, i.e., that causes CCC, 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.

Code decay 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.**

A classic non-benign CCC!

Irony, Cont'd

IH failed to protect against these changes because they were unpredictable, surprising *requirements* changes and not more predictable 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.

A Gut Feeling

It's my gut feeling (i.e., I cannot prove it) that ...

no matter the set of concerns, ...

if I know about them before I determine the architecture (\equiv determine the modules) (\equiv determine the decomposition) of the code, ...

A Gut Feeling , Cont'd

I can find a decomposition for it such that no concern is CC beyond the tolerable limit of the designed CC modules for logging, error handling, etc.

I have never encountered a counter example.

Go4 Patterns

The ingenuity of the Gang of Four in the architectures shown in the famous *Design Patterns* book strengthens this gut feeling.

After Adding New Concern

But after adding a new requirement, ...

Oy! All bets are off!

CCCs begin to come out of the woodwork and continue to do so with each new requirement until I break down and restructure.

Perhaps, using aspects to deal with CCCs would help delay the necessity to restructure.

Anticipatory Design

Of course, in every design — first time or restructuring — I try to anticipate future concerns that might become CC.

Sometimes I get lucky.

Most times I do not! sigh.

Agile Manifesto

One of the arguments for agile methods is that new requirements are always coming along to destroy the cleanliness of any decomposition.

Ergo, why bother do RE or design? (I don't buy their conclusion.)

In RE

So it seems to me that in RE, there are no CCCs, just concerns.

Of course, in this statement, there is an underlying assumption that ...

**RE means to continue to gather requirements
≡ concerns without deciding on any
implementation architecture.**

(Later, I will consider other variations of RE.)

Choosing an Architecture

The moment you choose an architecture, even if no concern is CC in this architecture, the next concern you gather runs the risk of being CC.

Note that structuring requirements is *not* necessarily determining an architecture.

Requirements Structuring

Requirements can be organized by

- **goals,**
- **use cases vs. misuse cases,**
- **viewpoints,**
- **stakeholders,**
- **functional vs. nonfunctional,**
- **win conditions vs. lose conditions, and**
- **whatever.**

Apparent CCCs

That a concern is CC with respect to the chosen organization, ...

doesn't mean that the concern is going to be CC with respect to the implementation.

Apparent CCCs, Cont'd

The concern is CC with respect to the implementation only if you structure the implementation like you've structured the requirements or in any way that makes the concern CC.

So you really should postpone all architectural design until the requirements are worked out in detail.

Until RE is Finished?

But that is NEVER!

True, and that's why I did not say "until RE is finished".

Moreover, that's the idea behind the iterative and agile lifecycle models.

RE is Never Finished

However, following these lifecycle models are too often used as excuses for not working out details that *are* knowable before beginning design and implementation, such as alternatives, exceptions, and errors to known use cases, scenarios, and functions *before* jumping into coding.

Coding is exciting, working out details in writing is boring!

More RE Leads to Faster Coding

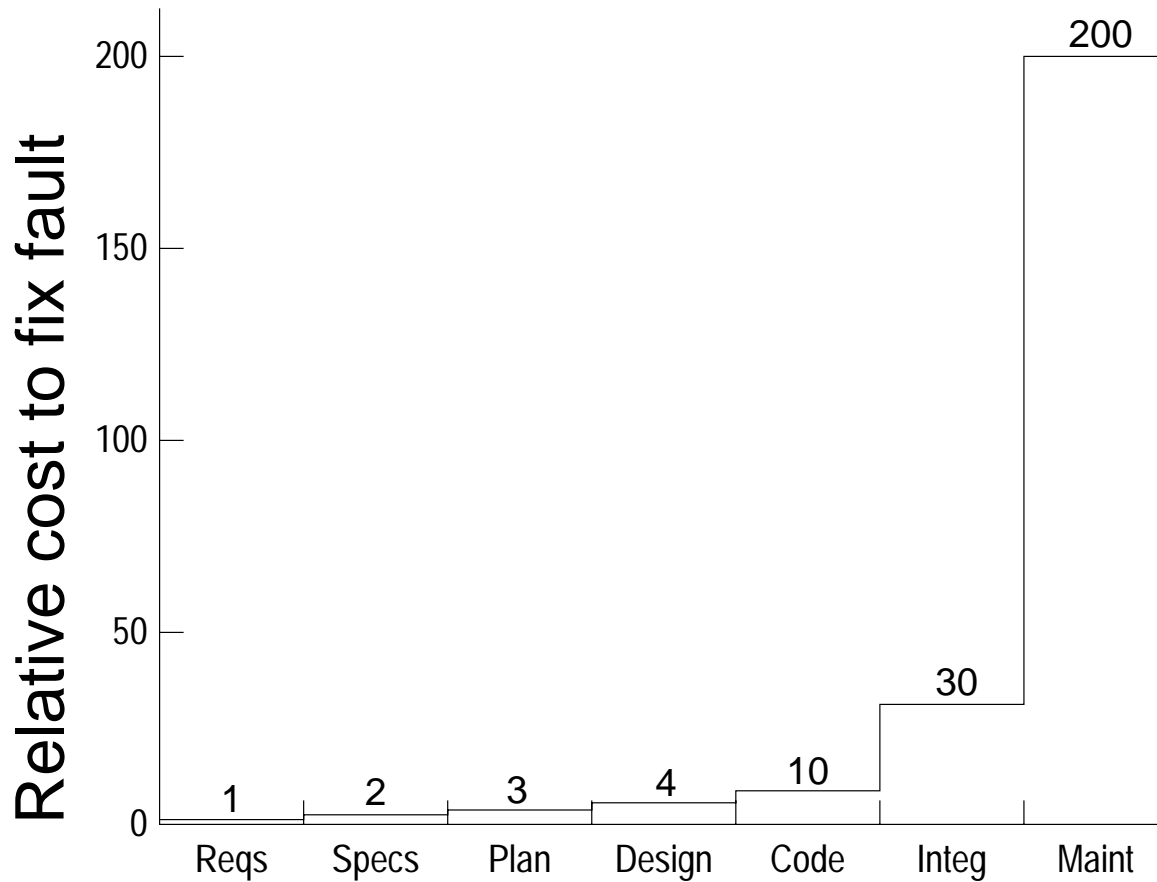
But we know that the more you postpone coding to work out details, the faster, more correctly, and cheaper the coding goes, as

- **errors that would be expensive to fix later never even happen, and**
- **the implementation staff does not have to do any on-the-fly RE, and**
- **and unit and integration testing go much more smoothly.**

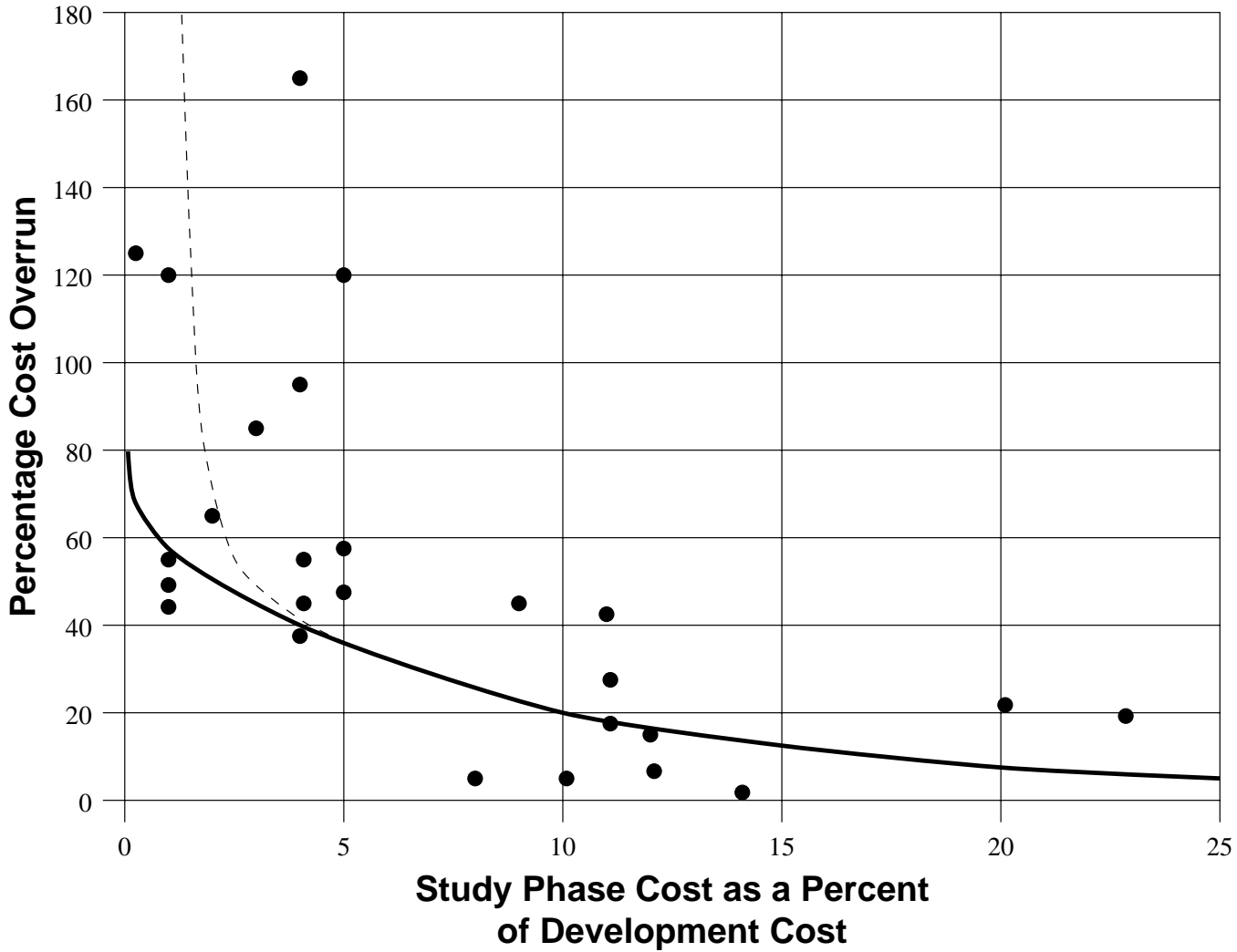
Data on Costs

We have data

- **relating error correction costs to the stage in which that error is found [Schach 1992 & Boehm 1981], and**
- **relating cost overrun to the percentage of the lifecycle spent in studying the problem at hand [Forsberg & Mooz 1997].**



Phase in which fault is detected and fixed



Therefore, No CCCs in RE

So that's why I believe that in RE, there are no CCCs and that everything is just a concern and ...

if you are getting CCCs at RE time, you're deciding architecture too soon!

If Architecture is a Requirement

Of course, *if* a system's architecture is part of the system's requirements, e.g.,

- you are working on a product in a product line,
- you are working with a legacy system, or
- you are in a second or later iteration in an iterative lifecycle and you are not restructuring in this iteration,

then CCCs are unavoidable, but

then nevertheless, the concerns are CC the architecture and not the other requirements.

What I am NOT doing

Note that I am not saying that AOP has no place.

AOP *has* a place and is important in providing systematic ways to deal with CCCs as they inevitably occur.

All I am saying is that if you believe that CCCs are present during RE, then you are not doing only RE, you are doing RE mixed with other activities, which you may have a requirement to do.