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

Daniel M. Berry

School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
dberry@uwaterloo.ca
http://se.uwaterloo.ca/~dberry

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.

1 Introduction

The call for papers for this workshop says of the object-oriented programming methods introduced in the last decade, "There is no proof and no evidence that software productivity has increased with the new methods"[1] The call for papers argues that as a consequence of this and other reasons, there is an urgent need for new "post object-oriented" software engineering and programming techniques. However, there is no proof, evidence, or guarantee that any such new technique will increase productivity any more than object-oriented techniques have. Indeed, the past failures of any method to significantly increase productivity is the strongest predictor that any new technique will fare no better. In other words, what makes you, who designs new methods, think you can do better?

This paper tries to get to the root of why any given new programming technique has not improved productivity very much. This paper is, as the call for papers requires, an attempt "to analyze why some ideas were or were not successful" with the choice of "were not".

This paper is about building computer-based systems (CBS). Since the most flexible component of a CBS is its software, we often talk about developing its software, when in fact we are really developing the whole CBS. In this paper, "software" and "CBS" are used interchangeably.

This paper is based on personal observation. Sometimes, I describe an idea based solely on my own observations over the years. Such ideas carry no citation and have no formal or experimental basis. If your observations disagree, then please write your own rebuttal.

[1] Actually, this claim is a bit too strong. There is a feeling that object orientation has improved programming a bit, but is clearly not the silver bullet that it was hoped, and even hyped, to be.

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. This belief or feeling may be incorrect in the sense that it is not supported by the data. Whenever possible, I alert the reader of this mode by giving the belief-or-feeling-based sentence in Italics.

2 Programming Then and Now

I learned to program in 1965. I can remember the first large program I wrote in 1966 outside the class room for a real-life problem. It was a program that implemented the external functionality of Operation Match, a computer-based dating and matchmaking service set up in the mid 1960s. I wrote it for my synagogue youth group in order that it could have a dance in which each person's date for the dance was that picked by a variation of the Operation Match software. The dance and the software were called "Operation Shadchan".[2] I got a hold of the questionnaire for Operation Match, which was being used to match a new client of one gender with previously registered clients of the opposite gender. Each client filled out the form twice, once about him or herself and the second time about his or her ideal mate. For each new client, the data for the client would be entered. Then the software would find all sufficiently good matches with the new clients from among the previously registered clients. Presumably, the matches had to be both good and balanced; that is, the total number of questions for which each answered the way the other wanted had to be greater than a threshold and the difference between the number of matches in each direction had to be smaller than another threshold. I adapted this questionnaire for high school purposes. For example, I changed "Do you believe in sex on the first date?" to "Do you believe in kissing on the first date?".[3]

I then proceeded to write a program. 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 method of those days:

- 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.

The first requirements were fairly straightforward to identify. Since this matching was for a dance, unlike with Operation Match, each person would be matched with one and only one person of the opposite gender.[4] Obviously, we had to make sure that in the

[2] "Shadchan" is Yiddish for "Matchmaker". The "ch" in "shadchan" is pronounced as the "X" in "TEX". One person attending the dance thought the name of the dance was "Operation Shotgun".

[3] Remember, this was during the mid 1960s!

[4] It was assumed that each person wanted someone of the opposite gender.

input set, the number of boys was equal to the number of girls, so that no one would have the stigma of being unmatched by the software. The next requirements were not so easy to identify. Each boy and each girl should be matched to his or her best match. So, I wrote a loop that cycled through each person and for each, cycled through each other person of the opposite gender to find the best match. But whoa! what is a match? Ah, it cannot be just one way. It must be a mutually balanced match. But double whoa! I have to remove from the list of those that can be cycled through in either loop those that have been matched before. But triple whoa! Suppose the best mutual match for someone is among those not considered because that best mutual match has already been matched to someone else, and that earlier match is not as good. Worse than that, suppose that what is left to match with someone are absolute disasters for the someone. This simple algorithm is not so hot.

In those days and at that age, couples in which the girl was taller than the boy was a disaster, especially if the difference was big. Also, it was not considered so good if the girl of a couple were older than the boy. Therefore, to avoid being painted into a disastrous-match corner, I decided to search in a particular order, from hardest-to-find-non-disastrous matches to easiest. That is, I searched for matches for the tallest girls and shortest boys first and the shortest girls and tallest boys last. Presumably the tallest boys get assigned fairly early to the tallest girls and the shortest girls would get assigned fairly early to the shortest boys. I randomly chose the gender of the first person to be matched and alternated the gender in each iteration of the outer loop. To help avoid disastrous matches, I gave extra weight to the height and age questions in calculating the goodness of any potential match. Each "whoa" above represents a scrapping of previously written code in favor of new code based on the newly discovered requirements. Thus, I was discovering requirement flaws and correcting them during coding as a result of what I learned during coding.

The biggest problem I had was remembering all the requirements. It seemed that each thought brought about the discovery of more requirements, and these 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, 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 them all down, only to have to discover them again or to forget them entirely.

Basically, programming felt like skiing down a narrow downhill valley with an avalanche following me down the hill and gaining on me.

Nowadays, we follow more systematic methods. However, the basic feelings have not changed. Since then, I have maintained and enhanced a curve fitting application for chemists[5] I built a payroll system. I have participated in the writing of a collection of text formatting software. I have watched my graduate students develop tools for requirements engineering. I watched my ex-wife build a simulation system. I have watched my ex-wife and former students and friends involved in start ups building large CBSs. To me and, I am sure the others, programming still feels like skiing with an avalanche

[5] This was my second system, and it suffered Brooks's second system syndrome [17], as I tried to build a super-duper, all-inclusive, fancy whiz-bang general curve fitting application with all sorts of fantastic options.

following closely behind. I see all the others undergoing similar feelings and being as empathic as I am, I get the same skiing feeling. 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 seem to grow faster than the code.

The real problem of software engineering 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. I am not the first to say so. Fred Brooks and Michael Jackson, among others, have said the same for years. Let us examine their arguments.

3 The Search for a Silver Bullet

Some time ago, Fred Brooks, in saying that there is no software engineering silver bullet, classified software issues into the essence and the accidents [16]. The essence is what the software does and the accidents are the technology by which the software does the essence or by which the software is developed. That is, the requirements are the essence, while the language, tools, and methods used are the accidents. He went on to say, "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." This quotation captures the essential difficulty with software that must be addressed by any method that purports to alter fundamentally the way we program, that purports to make programming an order of magnitude easier, that purports to be the silver programming bullet we have all been looking for. Heretofore, no single method has put a dent into this essential problem, although all the discovered methods have combined to improve programming by at least an order of magnitude since 1968, the year the term "software engineering" was invented [41]. Moreover, Brooks says based on his experience, the silver bullet will never be found: "No major improvement in the software engineering area will ever appear."

The obvious question is "Why is there no silver bullet, and why can there not be a silver bullet?" The contention of this paper is that every time a new method that is intended to be a silver bullet is introduced, it does make many parts of the accidents easier. However, as soon as the method needs to deal with the essence or something affecting or affected by the essence, suddenly one part of the method becomes painful, distasteful, and difficult, so much so that this part of the method gets postponed, avoided, and skipped. Consequently, the method ends up being only slightly better than no method at all in dealing with essence-borne difficulties.

But, what *is* so difficult about understanding requirements? I mean, it should be possible to sit down with the customer and users, ask a few questions, understand the answers, and then synthesize a complete requirements specification. However, it never works out that way. Michael Jackson, Paul Clements, David Parnas, Meir Lehman, Bennet Lientz, Burton Swanson, and Laszlo Belady explain why.

4 Requirements Change

Michael Jackson, in his Keynote address at the 1994 International Conference on Requirements Engineering [33] said that two things are known about requirements:

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

The first implies that a CBS will always have to be modified, to accommodate the changed requirements. Even more strongly, *there ain't no way that requirements are not gonna change*, and there is as much chance of stopping requirements change as there is stopping the growth of a fully functioning and heartily eating teenager. The second implies that a CBS will always have to be modified, to accommodate the changes necessitated by better understanding, as the misunderstandings are discovered. Clements and Parnas describe how difficult it is to understand everything that might be relevant [44].

Meir Lehman [36] classifies a system that solves a problem or implements an application in some real world domain as an E-type system. He points out that once installed, an E-type system becomes inextricably part of the application domain so that it ends up altering its own requirements.

Certainly, not all changes to a CBS are due to requirement changes, but the data show that a large majority of them are. Bennett Lientz and Burton Swanson found that of all maintenance of application software, 20% deal with correcting errors, and 80% deal with changing requirements. Of the requirement changes, 31% are to adapt the software to new platforms, and 69% are for perfecting the software, to improve its performance or to enhance its functionality [37].

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 [7], [8]. That is, as programs undergo continual change their structure decays to the point that it is very hard to add something new or change something already there without affecting seemingly unrelated parts of the program in a way that causes errors. It can be difficult even to find all the places that need to be modified. The programmers make poor guesses and the program, if it even runs, behaves in strange and unpredicted ways. They modeled the phenomenon mathematically and derived a graph like that of Fig. 1, showing the expected number of errors in a program as a function of time, as measured by the ordinal numbers of releases during which modifications are made. In practice, the curve is not as smooth as in the figure, and it is sometimes necessary to get very far into the upswing before being sure where the minimum point is. The minimum point represents the software at its most bug-free release. After this point, during what will be called the *Belady-Lehman (B-L) upswing*, the software's structure has so decayed that it is impossible to change anything without adding more errors than have been fixed by the change.

The alternative to continuing on the B-L upswing for a CBS is to roll back to the best version, that is, the version that existed at the minimum point. Of course, rolling back assumes that all versions have been saved. All the bugs in this best version are declared to be features, and no changes are ever made in the CBS from then on. Usually not changing a CBS means that the CBS is dead, that no one is demanding changes because no one is using the software any more. However, some old faithful, mature, and reliable

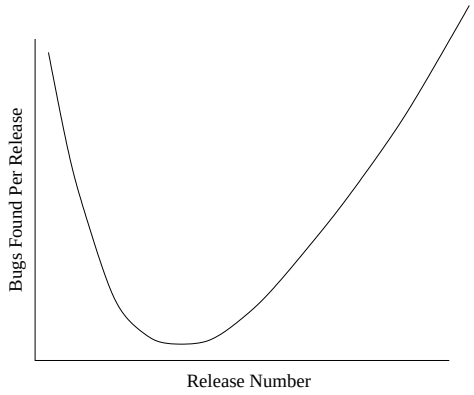

Fig. 1. Belady-Lehman Graph

programs e.g, cat and other basic UNIX applications, vi, and ditroff,[6] have gone this all-bugs-are-features route. The user community has grown to accept, and even, require that they will never change. If the remaining bugs of the 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 to make future modifications possible. Another alternative that works in some special cases is to use the best version as a feature server for what it can do well and to build a new CBS that implements only the new and corrected features and has the feature server do the features of the best version of the old CBS.

The time at which the minimum point comes and the slopes of the curves before and after the minimum point vary from development to development. The more complex the CBS is, the steeper the curve tends to be. Moreover, most of the time, for a carefully developed CBS, the minimum point tends to come in later releases of that CBS. However, occasionally, the minimum point is passed during the development of the first release, as a result of extensive requirements creep during that initial development. The requirements have changed so much since the initial commitment to architecture that the architecture has had to be changed so much that it is brittle. It has become very hard to accommodate new or changed requirements without breaking the CBS. Sometimes, the minimum point is passed during the initial development as a result of code being slapped together into the CBS with no sense of structure at all. The slightest requirements change breaks the CBS.

5 Purpose of Methods

One view of software development methods is that each method has as its underlying purpose to tame the B-L graph for the CBS developments to which it is applied. That

[6] Well, at least *I* think so! One great thing about these programs that have not been modified since the 1980s is that their speed doubles every 18 months!

is, each method tries to delay the beginning of the upswing or to lower the slope of that B-L upswing or both. For example, Information Hiding [43] attempts to structure a system into modules such that each implementation change results in changing only the code, and not the interface, of only one module. If such a modularization can be found, then all the code affecting and affected by any implementation change is confined to one module. Thus, it is easier to do the modifications correctly and without adversely affecting other parts of the system. Hence, arrival at the minimum point is delayed and the slope of the upswing is reduced.

In this sense, each method, if followed religiously, works. Each method provides the programmer a way to manage complexity and change so as to delay and moderate the B-L upswing. However, each method has a catch, a fatal flaw, at least one step that is a real pain to do, that people put off. People put off this painful step in their haste to get the software done and shipped out or to do more interesting things, like write more new code. Consequently, the software tends to decay no matter what. The B-L upswing is inevitable.

What is the pain in Information Hiding? Its success in making future changes easy depends on having identified a right decomposition. If a new requirement comes along that causes changes that bridge several modules, these changes might very well be harder than if the code were more monolithic, simply because it is generally easier on tools to search within one, even big, module than in several, even small, modules [22], [29]. Moreover, future changes, especially those interacting with the new requirement, will likely cross module boundaries.[7] Consequently, it is really necessary to restructure the code into a different set of modules. This restructuring is a major pain, as it means moving code around, writing new code, and possibly throwing out old code for no externally observable change in functionality. It is something that gets put off, causing more modifications to be made on an inappropriate structure.

The major irony is that the reason that the painful modifications are necessary is that the module structure no longer hides all information that should be hidden. Because of changes in requirements, there is some information scattered over several modules and exposed from each of these modules. Note that these changes are requirements changes rather than implementation changes, which continue to be effectively hidden. The painful modifications being avoided are those necessary to restore implementation information hiding, so that future implementation changes would be easier. Without the painful changes, all changes, both implementation and requirements-directed, will be painful. This same pain applies to any method based on Information Hiding, such as Object-Oriented Programming.

In the subsequent sections, each of a number of models, methods, and tools is examined to determine what its painful step is. In some cases, the whole model, method, or tool is the pain; in others, one particular step is the pain. No matter what, when people use these models, methods, and tools, they tend to get stopped by a painful step.

[7] For an example that requires minimum explanation, consider two independently developed modules, for implementing unbounded precision *integer* and *real* arithmetic. Each can have its implementation change independently of the other. However, if the requirement of inter-type conversion is added, suddenly the two modules have to be programmed together with representations that allow conversion.

6 Development Models and Global Methods

This section considers several development models and general programming methods to identify their painful parts. The set chosen is only a sampling. Space limitations and reader boredom preclude covering more. It is hoped that after reading these, the reader is able to identify the inevitable pain in his or her own favorite model or method. In fact, for many models and methods, the pain lies in precisely the same or corresponding activities, all in response to requirements change. The models covered are the Build-and-Fix Model, the Waterfall Model and Requirements Engineering. The general methods covered are Structured Programming, Extreme Programming, Rapid Prototyping, and Formal Methods.

6.1 Build-and-Fix Model

The non-method that I applied when I built the Operation Shadchan program was the build and fix model. As indicated by Steve Schach [49], the basic cycle is:

1. Build the first version of the software.
2. Modify the software until client is satisfied.
3. Operate program until a problem is found; when a problem is found, go back to Step 2.

In this case, there is actually no pain, unless the customer's dissatisfaction, the repeated difficulties with the software, or the feeling of skiing in front of an avalanche gives the programmer pain. On the other hand, the model is not purported to be a silver bullet, and the B-L upswing can start almost immediately and can be rather steep, unless the program is small enough to be written entirely by one person.

6.2 Waterfall Model

The waterfall model [48] is an attempt to put discipline into the software development process by forcing understanding and documentation of the requirements before going on to design, by forcing understanding and documentation of design before going on to coding, by forcing thorough testing of the code while coding each module, etc. The model would work if the programmers could understand a stage thoroughly and document it fully before going on to the next stage [44]. However, understanding is difficult and elusive, and in particular, documentation is a pain. The typical programmer would prefer to get on to coding before documenting. Consequently, some programmers consider the whole model to be a pain, because it tries to force a disciplined way of working that obstructs programmers from getting on with the coding in favor of providing seemingly endless, useless documentation. However, even for the programmers who do believe in discipline, the waterfall becomes a pain in any circumstance in which the waterfall cannot be followed, e.g., when the full requirements are learned only after significant implementation has been carried out or when there is significant repeated backtracking, when new requirements are continually discovered.

6.3 Structured Programming

I can recall the first systematic programming method I learned, used, and taught, from 1973 until it fell out of fashion in the mid 1980s, namely, Structured Programming or Stepwise Refinement [53], [20]. After having done build-it-and-fix-it programming for years, Structured Programming appeared to be the silver bullet, at last, a way to put some order into the chaotic jumble of thoughts that characterized my programming, at last, a way to get way ahead of or to the side of the avalanche that was coming after me all the time.

In fact, I found that Structured Programming did work as promised for the development of the first version of any CBS. If I used the clues provided by the nouns that appeared in more than one high-level statement [9], [11], Structured Programming did help me keep track of the effects of one part of the program on another. It did help me divide my concerns and conquer them one-by-one, without fear of forgetting a decision I had made, because these decisions were encoded in the lengthy, well-chosen names of the abstract, high-level statements that were yet to be refined. Best of all, the structured development itself was an ideal documentation of the structure of the program.

However, God help me if I needed to change something in a program developed by stepwise refinement, particularly if the change was due to an overlooked or changed requirement. I was faced with two choices:

1. Patch the change into the code in the traditional way after a careful analysis of ripple effects; observe that the documented structured development assists in finding the portions of the code affected by and affecting the change.
2. Redo the entire structured development from the top downward, taking into account the new requirement and the old requirements, all in the right places in the refinement.

The problems with the first choice are that:

1. no matter how careful I was, always some ripple effects were overlooked, and
2. patching destroys the relation between the structured development and the code, so that the former is no longer accurate documentation of the abstractions leading to the code. This disrepancy grows with each change, thus hastening the B-L upswing.[8] Thus, the new code contradicts the nice structure imposed by the structured development. Moreover, the names of the high level abstractions that get refined into code no longer imply their refinements.

Therefore, the correct alternative was the second, to start from scratch on a new structured development that recognizes the modified full set of requirements. However, then the likelihood is that the new code does not match the old code. Most structured programmers do this redevelopment with an eye to reusing as much as possible. That is, whenever one encounters a high-level statement with identical name and semantics as before, what it derived before is put there. However, the programmer must be careful to use the same high-level statements as refinements whenever possible, and he or she

[8] It is clear that there is a disrepancy, because if the abstractions had derived the new code, then the new code would have been there before.

must be careful that in fact the same semantics is wanted as before and that he or she is not self deluding to the point of falsely believing a high-level statement has the same semantics as before.

This second alternative turned out to be so painful that I generally opted to the first alternative, patching the code and thus abandoned the protection, clarity, and documentation offered by Structured Programming.

About that time, I learned the value of faking it. Do the modification by patching up the code, and then go back and modify the original structured development to make it look like the new program was derived by Structured Programming. However, faking it was also painful, and soon, unless I was required to do so for external reasons, e.g., preparing a paper for publication or a contract, I quickly stopped even faking it. Adding to the pain of faking it was the certainty of not doing the faking perfectly and being caught. It was only later that David Parnas and Paul Clements legitimized faking it [44] and relieved my guilt.

6.4 Requirements Engineering

The basic premise of requirements engineering is spend sufficient time up front, before designing and coding, to anticipate all possible requirements and contingencies, so that design and coding consider all requirements and contingencies from the beginning [13], [42]. Consequently, fewer changes are required, and the B-L upswing is both delayed and moderated. Data show that errors found during requirements analysis cost one order of magnitude less to fix than errors found during coding and two orders of magnitude less to fix than errors found during operation [15]. These economics stems from the very factors that cause the B-L upswing, namely, the fact that in a full running program, there is a lot more code affecting and affected by the changes necessary to fix an error than in its requirements specification. There are data and experiences that show that the more time spent in requirements engineering, the smoother the implementation steps are [26], [21], not only in software engineering, but also in house building [14], [51]. When the implementation goes smoother, it takes less time, it is more predictable, and there are fewer bugs.

However, for reasons that are not entirely clear to me,[9] a confirmed requirements engineer, people seem to find haggling over requirements a royal pain. They would much rather move on to the coding, and they feel restless, bored, or even guilty when forced to spend more time on the requirements. A boss with his or her eyes focused on an unrealistically short deadline does not help in this respect. I would bet that Arnis Daugulis [21], his colleagues, and bosses at Latvenergo felt the pain of not being able to move on to implementation, even though in the end, they were happy that they did not move on to implement the first two requirements specifications, which were, in retrospect, wrong.

The pain is exacerbated, and is felt even by those willing to haggle the requirements, because the requirements engineer must make people discover requirements by

[9] Then again, I always read the manual for an appliance or piece of hardware or software completely before using the appliance or piece. I return the appliance or piece for a refund if there is *any* disagreement between what the manual says and the appliance or piece does, even if it is in only format of the screen during the set up procedure.

clairvoyance rather than by prototyping. The pain is increased even more as the backtracking of the waterfall model sets in, as new requirements continue to be discovered, even after it was thought that all requirements had been found.

There appear to be at least two ways of carrying out RE in advance of CBS design and implementation, what are called in the agile software development community [1] "Big Modeling Up Front (BMUF)" [3] and "Initial Requirements Up Front (IRUF)" [2]. They differ in the ways they treat the inevitable requirements changes.

In BMUF, the CBS developers try to create comprehensive requirement models for the whole system up front, and they try to specify the CBS completely before beginning its implementation. They try to get these models and specifications fully reviewed, validated, agreed to, and signed off by the customer and users. Basically, the developers try to get the requirements so well understood that they can be frozen, no matter how painful it is. However, as demonstrated in Section 4, the requirements continue to change as more and more is learned during implementation. To stem the tide of requirements change, the powers with vested interest in the freezing of the requirements create disincentives for changes, ranging from contractual penalties against the customer who demands changes to heavy bureaucracy, in the form of a change review board (CRB). For each proposed change, the CRB investigates the change's impact, economic and structural. For each proposed change, the CRB decides whether to reject or accept the change, and if it accepts the change, what penalties to exact. Of course, the CRB requires full documentation of all requirements models and specifications and up-to-date traceability among all these models and specifications.[10]

Agile developers, on the other hand, expect to be gathering requirements throughout the entire CBS development. Thus, they say that we should "embrace change" [2]. We should explore the effects of a proposed change against the organizations business goals. If the change is worth it, do it, carrying out the required modifications, including restructuring. If the change isn't worth it, don't do it [31]. It's that simple.

The objective of agile processes is to carry out a CBS "development project that

1. focuses and delivers the essential system only, since anything more is extra cost and maintenance,
2. takes into account that the content of the essential system may change during the course of the project because of changes in business conditions,
3. allows the customer to frequently view working functionality, recommend changes, and have changes incorporated into the system as it is built, and
4. delivers business value at the price and cost defined as appropriate by the customer." [50]

Accordingly, agile developers get the IRUF, with all the stakeholders, i.e., customers, users, and developers, participating actively at all times, in which as many requirements as possible are gathered up front from all and only stakeholders. The goals of getting the IRUF are [2]

[10] The agile development community says that traceability is a waste of resources. The cost of keeping trace data up to date must be balanced against the cost of calculating the trace data when they are needed to track down the ripple effects of a proposed change. The community believes that updating is so much more frequent than tracing that the total resources spent in continually updating outstrips the resources spent in occasional tracing.

1. to identify the scope of the CBS being built,
2. to define high-level requirements for the CBS, and
3. to build consensus among stakeholders as to what the requirements imply.

The IRUF session ideally is as short as a few hours, but it can stretch into days and even weeks in less than ideal circumstances, such as not all stakeholders being in one place or particularly tricky requirements. Then come several days of modeling sessions to produce a full set of models of the CBS as conceived by the IRUF. These models include use cases, which are eminently suitable for discussions between users and developers.

Following this modeling, the requirements are ranked by priority by all stakeholders. Business goals are taken into account during this ranking process, which typically requires about a day. Detailed modeling of any requirement takes place only during the beginning of the iteration during which it is decided to implement that requirement.

Notice that BMUF and IRUF differ in their approaches to dealing with the relentless, inevitable requirements changes. BMUF tries to anticipate all of them. IRUF does not; it just lets them come. BMUF is considered as not having succeeded totally if it fails to find a major requirement. The pain of dealing with the change with BMUF is felt during the RE process. IRUF practitioners embrace the change and decide on the basis of business value whether or not to do the change. The pain of dealing with the change with IRUF is felt in the re-implementation necessitated by the change, e.g., in the refactoring that can be necessary. See Section 6.5 for details about refactoring pain. Ultimately, neither approach can *prevent* a new requirement from appearing.

6.5 Extreme Programming

Extreme Programming (XP) [6] argues that the preplanning that is the cornerstone of each the various disciplined programming methods, such as the Waterfall model, documentation, requirements engineering, is a waste of time. This preplanning is a waste of time, because most likely, its results will be thrown out as new requirements are discovered. XP consists in simply building to the requirements that are understood at the time that programming commences. However, the requirements are given, not with a specification but with executable test cases. Unfortunately, these test cases are not always written because of the pain of writing test cases in general and in particular before it known what the code is supposed to do [40]. During this programming, a number of proven, minimum pain, and lightweight methods are applied to insure that the code that is produced meets the requirements and does so reliably. The methods include continual inspection, continual testing, and pair programming. Thus, the simplest architecture that is sufficient to deal with all the known requirements is used without too much consideration of possible changes in the future and making the architecture flexible enough to handle these changes. It is felt that too much consideration of the future is a waste, because the future requirements for which it plans for may never materialize and an architecture based on these future requirements may be wrong for the requirements that do come up in the future.

What happens when a requirement comes along that does not fit in the existing architecture? XP says that the software should be refactored. Refactoring consists in stepping back, considering the new current full set of requirements and finding a new

simplest architecture that fits the entire set. The code that has been written should be restructured to fit the new architecture, as if it were written with the new architecture from scratch. Doing so may require throwing code out and writing new code to do things that were already implemented. XP's rules say that refactoring should be done often. Because the code's structure is continually restructured to match its current needs, one avoids having to insert code that does not match the architecture. One avoids having to search widely for the code that affects and is affected by the changes. Thus, at all times, one is using a well-structured modularization that hides information well and that is suited to be modified without damaging the architecture. Thus, the B-L upswing is delayed and moderated.

However, refactoring, itself, is painful [24]. It means stopping the development process long enough to consider the full set of requirements and to design a new architecture. Furthermore, it may mean *throwing out perfectly good code whose only fault is that it no longer matches the architecture*, something that is very painful to the authors[11] of the code that is changed. Consequently, in the rush to get the next release out on time or early, refactoring is postponed and postponed, frequently to the point that it gets harder and harder. Also, the programmers realize that the new architecture obtained in any refactoring will prove to be wrong for some future new requirements. Ironically, the rationale for doing XP and not another method, is used as an excuse for not following a key step of extreme programming.

Indeed, Elssamadisy and Schalliol report on an attempt to apply a modified XP approach to a software development project that was larger than any previous project to which they had applied XP, with great success. They use the term "bad smells" to describe symptoms of poor practice that can derail XP from its usual swift track [24]. Interestingly, each of these bad smells has the look, feel, and odor of an inevitable pain.

1. In XP, to ensure that all customer requirements are met, a customer representative has to be present or at least available at all times and he or she must participate in test case construction. The test cases, of course, are written to test that the software meets the requirements. Elssamadisy and Schalliol report that over time, the customer representatives begin to "refrain from that 'toil"' and begin to rely on the test cases written by the developers. Consequently, the test cases ceased to be an accurate indication of the customer's requirements, and in the end, the product failed to meet the customer's requirements even though it was running the test cases perfectly. It appears to me that participation in the frequent test case construction was a pain to the customer that was caused by the relentless march of new requirements.
2. XP insists on the writing of test cases first and on delivery of running programs at the ends of frequent development iterations in an attempt to avoid the well-known horror of hard-and-slow-to-fix bug-ridden code that greets many software development teams at delivery time. However, as a deadline for an iteration approaches, and it is apparent that programming speed is not where it should be, managers begin to excuse developers from having to write test cases; developers refrain from just barely necessary refactorings; testers cut back on the thoroughness of test cases; and developers fail to follow GUI standards and to write documentation, all in the name of sticking to the iteration schedule. Speed is maintained, but at the cost of

[11] Remember that it's *pair* programming.

buggy code. It appears to me that all curtailed activities were painful in XP, just as they are in other methods.

3. Elssamadisy and Schalliol report that there was a routine of doing unbooking and rebooking of a lease (URL). The first version of the routine did URL in the only way that was required by the first story. As a new story required a different specific case of URL, the required code was patched in to the routine as a special case. With each such new specific case of URL, the code became patchier and patchier; it became more and more apparent that a more and more painful refactoring was needed so that each specific case of URL is a specialization of a more generic, abstract URL. Elssamadisy and Schalliol

> were the ones who got stuck with fixing the problem. Also, because one of them is a big whiner, he kept complaining that "we knew this all along — something really stank for iterations and now I'm stuck with fixing it." The sad truth is that he was right. Every developer after the initial implementation knew that the code needed to be refactored, but for one reason or another ..., they never made the refactoring.
>
> This type of smell has no easy solution. The large refactoring had to be made because the inertia of the bad design was getting too high (footnote: Look ahead design would have been useful here also.) By taking the easy road from the beginning and completely ignoring the signs, we coded ourselves into a corner. So the moral of the story is this: when you find yourself making a large refactoring, stop and realize that it is probably because that you have skipped many smaller refactorings [24].

Is this ever a description of pain? It comes in a method which has been carefully designed to avoid many painful activities that programmers dislike.

The claim by agile developers is that the requirements to be addressed in the first iterations are those with the biggest business values. Any other requirements to be considered later are less important, and are even less likely to survive as originally conceived or even at all, due to the relentless change of requirements. Thus, refactoring becomes less and less important with each iteration.

However, this claim presupposes that the initial set of requirements, the IRUF, is so comprehensive that no new important requirements, forcing a major, painful refactoring, will ever be discovered. It's hard for me to imagine, and it runs counter to my experience that, any requirements modeling that is not the result of BMUF be so successful so as to effectively make refactoring completely unnecessary. Certainly, even if BMUF is done, it's hard not to ever need refactoring. Thus in the end, the potential for pain is there.

6.6 Rapid Prototyping

Rapid prototyping [4] is offered as a means to identify requirements, to try out various ideas, to do usability testing, etc. To the extent that it succeeds, it delays and moderates the B-L upswing. The pain is to throw the prototype out and to start anew when implementation starts. Often, in the face of an impending implementation deadline, instead, the prototype is used as a basis for the first implementation, thus ratifying poor design

decisions that were made for expedience in putting together the prototype rapidly. The need to throw out the prototype and start all over with a systematic development is at least the need to refactor if one is following XP. When the prototype's architecture is used for the production software, the very first production version is brittle, hard to modify, and prone to breaking whenever a change is made.

6.7 Formal Methods

There are a number of formal program development methods, each of which starts with a formal specification of the CBS to be built [52], [19]. Once a formal specification has been written it can be subjected to verification that it meets higher-level formally stated requirements, such as security criteria. Any code written for the implementation can be verified as correct with respect to the specification [32], [23]. Sometimes implementing code can be generated directly from the specification [5].

Some consider the mere writing of a formal specification a pain. Some consider writing such a specification to be fun, but consider all the verification that needs to be done to be a pain. Some consider this verification to be fun. However, my experience is that everyone considers the work dealing with changed requirements to be a pain. The specification has to be changed with the same difficulties as changing code, that is, of finding all other parts of the specification affecting or affected by the change at hand.

The worst of all is the necessary reverification. Since requirements are inherently global, formal statements about them and proofs about the formal statements are inherently global as well. Potentially every proof needs to be redone, because even though a theorem clearly still holds, the old proof may used changed formulae. Some have tried to build tools to automatically redo proofs in the face of changing specifications [39], [12]. The pain is enough to drive one to adopt a lightweight method in which only the specification is written and compiler-level checks are done and no verification is done [10].

7 Specific Methods and Techniques

Besides models and methods for the overall process of software engineering, there are a large variety of specific methods and techniques each of which is for attacking one specific problem of software engineering. Even these small methods have their pains, exactly where they brush up against changes. Again, only a small sampling of the available methods are covered in the hope that they are a convincing, representative sample.

7.1 Inspection

The effectiveness of inspection [25] as a technique for finding errors has been documented widely [27]. When applied to early documents, such as requirements or design documents, inspection helps delay and moderate the B-L upswing. However, inspection is a double pain. First, the documents to be inspected must be produced, and we know that documentation itself is a pain. Second, inspection is one of these unpopular activities that are the first to be scrubbed when the deadlines are looming [27]. Roger Pressman quotes Machiavelli in dealing with the unpopularity of inspection.

> "*... some maladies, as doctors say, at the beginning are easy to cure but difficult to recognize ... but in the course of time ... become easy to recognize but difficult to cure.*" Indeed! But we just don't listen when it comes to software. Every shred of evidence indicates that formal technical reviews (for example, inspections) result in fewer production bugs, lower maintenance costs, and higher software success rates. Yet we're unwilling to plan the effort required to recognize bugs in their early stage, even though bugs found in the field cost as much as 200 times more to correct [46].

7.2 The Daily Build

In a situation in which lots of programmers are continually modifying relatively but not completely independent code modules in a single CBS to fix bugs, improve performance, add new features, etc., a common practice is the process called the *daily build* [38]. It works best when there is tool support for version control, including commands for checking a module out, checking a module in, and identifying potential conflicts between what is checked in and what is already in.

Each programmer gets a problem, perhaps a bug to fix, performance to improve, or a feature to add, and he or she plans his or her attack. When ready, he or she downloads the latest versions of each module that needs to be changed. He or she makes the changes, he or she tests and regression tests the changed modules, he or she tests and regression tests the whole changed system, and when he or she is satisfied that the changes will work, he or she checks the modules back in.

The fun comes when each day, whatever complete collection of modules is there is compiled into a running version of the system and is tested and regression tested. If the system passes all the test, everything is fine. If not, whoever made the last change has to fix the problems, perhaps in consultation with other programmers, particularly of the modules affected by his or her changes. Consequently, the incentive is for each programmer to work quickly and to verify that the version of each module that is there just before a check in is what was checked out. If not, he or she should certainly feel compelled to re-implement and test his or her changes on that latest version. If the changes made by others are truly independent, this reimplementation is straightforward, and consists of folding in the independent changes identified with the help of diff. Obviously, the faster he or she works after checking out the modules to be changed, the less likely he or she will find them changed by others at checkin. Note that two programmers can work at cross purposes. However, if they notice that they are working on the same modules, they are encouraged to work together to ensure that their changes do not conflict.

The testing, regression testing, the checking in, and possible rework are all real pains. Most people would just as soon dispense with them to get on to the real work, programming. However, in this case, the social and work costs of failing to do these activities is very high. Basically, the one whose checkin caused the build not to work correctly is the turkey of the day to be hung by his or her thumbs, and he or she has to spend considerable time to fix a version that has multiple programmers' changes.

7.3 Open Sourcing

Open sourcing [47] is a world-wide application of the daily build, compounded from daily to continually. The main advantage is that the software has a world-wide legion of merciless inspectors and bug fixers. Whoever sees the source of a problem is encouraged by the promise of fame to fix it. "Given enough eyeballs, all bugs are shallow." The pain for each programmer is integrating his or her update in the face of almost continual change. The biggest pain is for the manager of the whole project, e.g., Linus Torvalds for Linux. Can you imagine what he has to go through to reconcile different fixes of a problem, to pick the one that he thinks fits best in a moving target. Compound this problem with the fact that he must deal with several problems at once. Goodness of fit is measured not only by how well it solves the problem but also by how well be consistent with all other changes that are being done. If the manager is not careful, the source could slide into B-L upswing oblivion very quickly.

8 Conclusions

Thus, it appears that there is no software engineering silver bullet. All software engineering 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 software engineering bullet to leave a pure painless silver software engineering bullet.

The situation with software engineering methods is not unlike that stubborn chest of drawers in the old slapstick movies; a shlimazel[12] pushes in one drawer and out pops another one, usually right smack dab on the poor shlimazel's 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.

There cannot be any significant change in programming until someone figures out how to deal, with a lot less pain, with the relentless change of requirements and all of its ripple effects. Perhaps, we have to accept that CBS development is an art and that no amount of systematization will make it less so. To the extent that we can 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. However, for any new problem, where the excitement of innovation is, there is no hope of avoiding relentless change as we learn about the domain, the need for artistry, and the pain.

The key concept in the Capability Maturity Model (CMM) [45] 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.

Perhaps now it is clear why I no longer get excited over any new language, development model, method, tool, or environment that is supposed to improve programming. It is clear also why I think that the most important work is that addressing requirements, changes, and the psychology and sociology of programming.

[12] "Shlimazel" is Yiddish for "one who always suffers bad luck".

9 Acknowledgments

I am grateful to Matt Armstrong, Andrew Malton, Daniel Morales Germán, and Davor Svetinović for useful discussions about specific methods. I thank the anonymous referees of a previous version of this paper for their careful comments. Dealing with them led to what I believe is a better paper. I was supported in part by NSERC grant NSERC-RGPIN227055-00.

References

1. Principles: The Agile Alliance. The Agile Alliance (2001) http://www.agilealliance.org/
2. Ambler, S.W.: Agile Requirements Modeling. The Official Agile Modeling (AM) Site (2001) http://www.agilemodeling.com/essays/agileRequirementsModeling.htm
3. Ambler, S.W.: Agile Software Development. The Official Agile Modeling (AM) Site (2001) http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm
4. Andriole, S.J.: Fast Cheap Requirements: Prototype or Else!. IEEE Software **14**:2 (March 1994) 85–87
5. Balzer, R.M.: Transformational Implementation: An Example. IEEE Transactions on Software Engineering **SE-7**:1 (January 1981) 3–14
6. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, MA (1999)
7. Belady, L.A.; Lehman, M.M.: A Model of Large Program Development. IBM Systems Journal **15**:3 (1976) 225–252
8. Belady, L.A.; Lehman, M.M.: Program System Dynamics or the Metadynamics of Systems in Maintenance and Growth. Lehman, M.M.; Belady, L.A. (eds.): Program Evolution, Academic Press, London, UK (1985)
9. Berry, D.M.: An Example of Structured Programming. UCLA Computer Science Department Quarterly **2**:3 (July 1974)
10. Berry, D.M.: The Application of the Formal Development Methodology to Data Base Design and Integrity Verification. UCLA Computer Science Department Quarterly **9**:4 (Fall 1981)
11. Berry, D.M.: Program Proofs Produced Practically. Tutorial Notes of Fifth International Conference on Software Engineering, San Diego, CA (March 1981)
12. Berry, D.M.: An Ina Jo Proof Manager for the Formal Development Method. Proceedings of Verkshop III, Software Engineering Notes (August 1985)
13. Berry, D.M.; Lawrence, B.: Requirements Engineering. IEEE Software **15**:2 (March 1998) 26–29
14. Berry, D.M.: Software and House Requirements Engineering: Lessons Learned in Combating Requirements Creep. Requirements Engineering Journal **3**:3 & 4 (1998) 242–244
15. Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ (1981)
16. Brooks, F.P. Jr.: No Silver Bullet. Computer **20**:4 (April 1987) 10–19
17. Brooks, F.P. Jr.: The Mythical Man-Month: Essays on Software Engineering. Second Edition, Addison Wesley, Reading, MA (1995)
18. CECOM: Requirements Engineering and Rapid Prototyping Workshop Proceedings. Eatontown, NJ, U.S. Army CECOM (1989)
19. Chan, W.; Anderson, R.J.; Beame, P.; Burns, S.; Modugno, F.; Notkin, D.; Reese, J.D.: Model Checking Large Software Specifications. IEEE Transactions on Software Engineering **SE-24**:7 (July 1998) 498–520

20. Dahl, O.-J.; Dijkstra, E.W.; Hoare, C.A.R.: Structured Programming. Academic Press, London, UK (1972)
21. Daugulis, A.: Time Aspects in Requirements Engineering: or 'Every Cloud has a Silver Lining'. Requirements Engineering **5**:3 (2000) 137–143
22. Dunsmore, A.; Roger, M.; Wood, M.: Object-Oriented Inspection in the Face of Delocalization. Proceedings of the Twenty-Second International Conference on Software Engineering (ICSE2000), Limerick, Ireland (June 2000) 467–476
23. Elspas, B.; Levitt, K.N.; Waldinger, R.J.; Waksman, A.: An Assessment of Techniques for Proving Program Correctness. Computing Surveys **4**:2 (June 1972) 81–96
24. Elssamadisy, A.; Schalliol, G.: Recognizing and Responding to "Bad Smells" in Extreme Programming. Proceedings of the Twenty-Fourth International Conference on Software Engineering (ICSE2002), Orlando, FL (May 2001)
25. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal **15**:3 (1976) 182–211
26. Forsberg, K.; Mooz, H.: System Engineering Overview. Software Requirements Engineering, Second Edition, Thayer, R.H.; Dorfman, M. (eds.): IEEE Computer Society, Washington (1997)
27. Gilb, T.; Graham, D.: Software Inspection. Addison Wesley, Wokingham, UK (1993)
28. Goguen, J.A.; Tardo, J.: An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications. Proceedings Conference on Specifications of Reliable Software, Boston (1979)
29. Griswold, W.G.; Yuan, J.J.; Kato, Y.: Exploiting the Map Metaphor in a Tool for Software Evolution. Proceedings of the Twenty-Third International Conference on Software Engineering (ICSE2001), Toronto, ON, Canada (June 2001) 265–274
30. Hall, A.: Using Formal Methods to Develop an ATC Information System. IEEE Software **13**:2 (March 1996) 66–76
31. Highsmith, J.; Cockburn, A.: Agile Software Development: The Business of Innovation. IEEE Computer **34**:9 (September 2001) 120–122
32. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**:10 (October 1969) 576–580,585
33. Jackson, M.A.: The Role of Architecture in Requirements Engineering. Proceedings of the First International Conference on Requirements Engineering, IEEE Computer Society, Colorado Springs, CO (April 18–22 1994) 241
34. Kemmerer, R.A.: Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs. Ph.D. Dissertation, Computer Science Department, UCLA (1979)
35. Kemmerer, R.A.: Testing Formal Specifications to Detect Design Errors. IEEE Transactions on Software Engineering **SE-11**:1 (January 1985) 32–43
36. Lehman, M.M.: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE **68**:9 (September 1980) 1060–1076
37. Lientz, B.P.; Swanson, E.B.: Characteristics of Application Software Maintenance. Communications of the ACM **21**:6 (June 1978) 466–481
38. McConnell, S.: Daily Build and Smoke Test. IEEE Software **13**:4 (July/August 1996) 144–143
39. Moriconi, M.S.: A Designer/Verifier's Assistant. IEEE Transactions on Software Engineering **SE-5**:4 (July 1979) 387–401
40. Müller, M.M.; Tichy, W.F.: Case Study: Extreme Programming in a University Environment. Proceedings of the Twenty-Third International Conference on Software Engineering (ICSE2001), Toronto, ON, Canada (June 2001) 537–544

41. Naur, P.; Randell, B.: Software Engineering: Report on a Conference Sponsored by the NATO Science Commission. Garmisch, Germany, 7–11 October 1968, Scientific Affairs Division, NATO, Brussels, Belgium (January 1969)
42. Nuseibeh, B.; Easterbrook, S.: Requirements Engineering: A Roadmap. Finkelstein, A. (ed.): The Future of Software Engineering 2000, ACM, Limerick, Ireland (June 2000)
43. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM **15**:2 (December 1972) 1053–1058
44. Parnas, D.L.; Clements, P.C.: A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering **SE-12**:2 (February 1986) 196–257
45. Paulk, M.C.; Curtis, B.; Chrissis, M.B.; Weber, C.V.: Key Practices of the Capability Maturity Model. Technical Report, CMU/SEI-93-TR-25, Software Engineering Institute (February 1993)
46. Pressman, R.: Software According to Niccolò Machiavelli. IEEE Software **12**:1 (January 1995) 101–102
47. Raymond, E.: Linux and Open-Source Success. IEEE Software **26**:1 (January/February 1999) 85–89
48. Royce, W.W.: Managing the Development of Large Software Systems: Concepts and Techniques. Proceedings of WesCon (August 1970)
49. Schach, S.R.: Software Engineering. Second Edition, Aksen Associates & Irwin, Boston, MA (1992)
50. Schwaber, K.: Agile Processes — Emergence of Essential Systems. The Agile Alliance (2002) http://www.agilealliance.org/articles/
51. Wieringa, R.: Software Requirements Engineering: the Need for Systems Engineering and Literacy. Requirements Engineering Journal **6**:2 (2001) 132–134
52. Wing, J.M.: A Specifier's Introduction to Formal Methods. IEEE Computer **23**:9 (September 1990) 8–24
53. Wirth, N.: Program Development by Stepwise Refinement. Communications of the ACM **14**:4 (April 1971) 221–227