

When is it good (if ever) to scrap production code and start over?

[closed]

Asked 16 years, 2 months ago Modified 12 years, 3 months ago

Viewed 7k times



55



As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, [visit the help center](#) for guidance.

Closed 12 years ago.

I was asked to do a code review and report on the feasibility of adding a new feature to one of our new products, one that I haven't personally worked on until now. I know it's easy to nitpick someone else's code, but I'd say it's in bad shape (while trying to be as objective as possible). Some highlights from my code review:

- **Abuse of threads:** `QueueUserWorkItem` and threads in general are used a *lot*, and Thread-pool delegates have uninformative names such as `PoolStart` and `PoolStart2`. There is also a lack of proper

synchronization between threads, in particular accessing UI objects on threads other than the UI thread.

- **Magic numbers and magic strings:** Some `Const`'s and `Enum`'s are defined in the code, but much of the code relies on literal values.
- **Global variables:** Many variables are declared global and may or may not be initialized depending on what code paths get followed and what order things occur in. This gets very confusing when the code is also jumping around between threads.
- **Compiler warnings:** The main solution file contains 500+ warnings, and the total number is unknown to me. I got a warning from Visual Studio that it couldn't display any more warnings.
- **Half-finished classes:** The code was worked on and added to here and there, and I think this led to people forgetting what they had done before, so there are a few seemingly half-finished classes and empty stubs.
- **Not Invented Here:** The product duplicates functionality that already exists in common libraries used by other products, such as data access helpers, error logging helpers, and user interface helpers.
- **Separation of concerns:** I think someone was holding the book upside down when they read about the typical "UI -> business layer -> data access layer"

3-tier architecture. In this codebase, the UI layer directly accesses the database, because the business layer is partially implemented but mostly ignored due to not being fleshed out fully enough, and the data access layer controls the UI layer. Most of the low-level database and network methods operate on a global reference to the main form, and directly show, hide, and modify the form. Where the rather thin business layer *is* actually used, it also tends to control the UI directly. Most of this lower-level code also uses `MessageBox.Show` to display error messages when an exception occurs, and most swallow the original exception. This of course makes it a bit more complicated to start writing unit tests to verify the functionality of the program before attempting to refactor it.

I'm just scratching the surface here, but my question is simple enough: Would it make more sense to take the time to refactor the existing codebase, focusing on one issue at a time, or would you consider rewriting the entire thing from scratch?

EDIT: To clarify a bit, we do have the original requirements for the project, which is why starting over could be an option. Another way to phrase my question is: Can code ever reach a point where the cost of maintaining it would become greater than the cost of dumping it and starting over?

Share

edited Aug 30, 2012 at 18:33

Improve this question

Follow

community wiki

9 revs, 3 users 100%


Mike Spross

3 Changed to a community wiki since it feels wrong to be getting so much rep from a subjective question.

– [Mike Spross](#) Apr 17, 2009 at 1:18

2 Well, for better or worse, we "rebooted" the product rather than refactor it. I say "reboot" rather than "rewrite" because we realized that only a few core features were actually absolutely necessary, so we dropped a lot. Based on the new specs, I was able to create a working beta version that did 90% of what the original product did in 4 days, and it was a few tens of thousands lines of code shorter (yes, partly because I didn't reimplement every feature, but it really was a *huge* reduction in the amount of code regardless). Not to say this is *the* right answer, but it worked for us. – [Mike Spross](#)

Jul 24, 2010 at 3:24 

3 Also, the reason why I was able to pull together a rebooted version in 4 days is because, ultimately, this monstrous piece of software actually was doing something fairly simple at its heart. Almost all of the original complexity was completely unwarranted and unnecessary for the problem the software was meant to solve. – [Mike Spross](#) Jul 24, 2010 at 3:29 

You might want to move your last two comments to your post. They are a valuable contribution to answering this subjective

29 Answers

Sorted by:

Highest score (default)



38

Without any offense intended, the decision to rewrite a codebase from scratch is a common, and serious management mistake newbie software developers make.



There are many disadvantages to be wary of.



- Rewrites stop new features from being developed cold for months/years. Few, if any companies can afford to stand-still for this long.
- Most development schedules are difficult to nail. This rewrite will be no exception. Amplify the previous point by, now, a delay in development.
- Bugs that were fixed in the existing codebase through painful experience will be re-introduced. Joel Spolsky has more examples in [this article](#).
- Danger of falling victim to the [Second-system effect](#) - in summary, ``People who have designed something only once before try to do all the things they "didn't get to do last time", loading the project up with all the things they put off while making version one, even if most of them should be put off in version two as well."
- Once this expensive, burdensome rewrite is completed, the very next team to inherit the new codebase is likely to use the same excuses for doing

another rewrite. Programmers hate learning someone else's code. No one writes perfect code because perfection is so subjective. Find me any real-world application and I can give you a damning indictment and rationale for doing a from-scratch rewrite.

Whether you ultimately rewrite from scratch or not, beginning a refactoring phase now is a good way to both really sit down and understand the problem so that the rewrite will go more smoothly if truly called for, as well as giving the existing codebase an honest look to really see if a rewrite's needed.

Share Improve this answer

answered Sep 28, 2008 at 3:14

Follow



[mbac32768](#)

11.6k ● 9 ● 37 ● 41

-
- 4 I'd upvote this again if it would let me. I definitely agree with your points. OTOH, I wonder how many hard-earned "bug fixes" were added solely to fix a symptom of poor design -- if a cleaner design (with less code) can be created, then there is less chance of encountering the same bugs again.

– [Mike Spross](#) Sep 28, 2008 at 3:47

Hmmm - sounds like Mike Spross is about to invent refactoring! – [MarkJ](#) Feb 26, 2009 at 11:11

Martin Fowler agrees with this too.

martinfowler.com/bliki/StranglerApplication.html – [MarkJ](#) Mar 16, 2009 at 11:33

To actually scrap and start over?



When the current code doesn't do what you would like it to do, and would be cost prohibitive to change.

33



I'm sure someone will now link Joel's article about Netscape throwing their code away and how it's oh-so-terrible and a huge mistake. I don't want to talk about it in detail, but if you do link that article, before you do so, consider this: the IE engine, the engine that allowed MS to release IE 4, 5, 5.5, and 6 in quick succession, the IE engine that totally destroyed Netscape... it was new. Trident was a new engine after they threw away the IE 3 engine *because it didn't provide a suitable basis for their future development work*. MS did that which Joel says you must never do, and it is *because* MS did so that they had a browser that allowed them to completely eclipse Netscape. So please... just meditate on that thought for a moment before you link Joel and say "oh you should never do it, it's a terrible idea".



Share Improve this answer

edited Sep 28, 2008 at 8:46

Follow

answered Sep 27, 2008 at 23:39



DrPizza

18.3k ● 7 ● 42 ● 53

7 Of course, it could also be said that companies like Microsoft generate their own gravity. If you have the resources of Microsoft, then by all means, rewrite. Otherwise, be very cautious. – [Ryan Lundy](#) Sep 28, 2008 at 22:02

-
- 4 Kyralessa, remember that IE 3 was not a juggernaut; Netscape was at the time. – [Bernard](#) Sep 29, 2008 at 3:40
-
- 2 The problem with this answer is that it ignores the difficulty of estimating the effort involved in re-writing the code. If you knew that it would be possible to re-write the code to achieve your objectives faster than changing the existing code, it's a no-brainer. But estimating is the hard part. – [PeterAllenWebb](#) Oct 3, 2008 at 19:15
-
- 2 Perhaps. But I don't think it's necessary all that difficult, because I think that any kind of radical alteration to a program will tend to be difficult and time-consuming. Far too often I think we incorrectly veer towards "modify", when actually, software isn't as malleable as is assumed. – [DrPizza](#) Oct 4, 2008 at 10:12
-
- 4 Kyralessa's point is that Microsoft had a few *billion* in the bank from their other products, so they could pour money into a rewrite because they considered the browser very strategic. Sorry, DrPizza, but I've experienced some failed overoptimistic rewrites. It's a dangerous road to go down. – [MarkJ](#) Feb 15, 2009 at 21:10
-



14



Be very carefull with this:



1. Are you sure you aren't just being lazy and not bothering to read the code

2. Are you being arrogant about the great code you will write compared to the rubbish anyone else produced.
3. Remember tested-working code is worth a lot more than imaginary yet-to-be-written code

In the words of our esteemed host and overlord, [Joel - things you should never do](#),

it's not always wrong to abandon working code - but you have to be sure about the reason.

Share Improve this answer

edited Sep 29, 2008 at 16:30

Follow

answered Sep 28, 2008 at 0:29



Martin Beckett

96k ● 28 ● 195 ● 268

I've read the code, that's how I did the code review ;-) In this case, I think it's obvious the code could be cleaned up, but is it worth breaking the code by trying to slowly refactor, since the code is so fragile, or better to rewrite, since we can start small and build it back up again. – **Mike Spross** Sep 28, 2008 at 0:54

As for ego/arrogance, I've written my share of ugly code and then inevitably was the one who had to fix it later. At some point, you have to look at the code objectively (whether it's yours or not) and ask yourself "Does any of this make sense?" Tolerating bad code because it works seems dangerous. – **Mike Spross** Sep 28, 2008 at 0:59

Agreed, but there is a natural tendency to decide to reinvent everything yourself rather than get to grips with what's already

there, you have to be certain of your reasons.

– [Martin Beckett](#) Sep 28, 2008 at 2:46

Indeed. There need to be clear reasons (and incentive) to toss everything out. Even if we do break away and start fresh, there may still be algorithms or concepts that can be borrowed from the existing codebase, so in the end, there might be no such thing as "throwing it all away."

– [Mike Spross](#) Sep 28, 2008 at 4:11

Do it gradually, organically, by refactoring & replacing the parts that need extending. You may find in the end none of the legacy code is left, and effectively you have thrown it away. But it's a low-risk route & you're always adding value.

martinfowler.com/bliki/StranglerApplication.html – [MarkJ](#) Mar 16, 2009 at 11:45



13

A rule of thumb I've found useful is that if given a code base, if I have to re-write more than 25% of the code to make it work or modify it based upon new requirements, you may as well re-write it from scratch.



The reasoning is that you can only patch a body of code so far; beyond a certain point, it's quicker to do over.

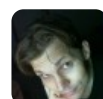


There's an underlying assumption that you have a mechanism (such as thorough unit and/or system tests) that will tell you whether your re-written version is functionally equivalent (where it needs to be) as the original.

Share Improve this answer

answered Sep 27, 2008 at 23:43

Follow



[Jason Etheridge](#)

6,897 ● 5 ● 31 ● 33

would be cool to come up with a formal mathematical model for this process – [Lance Pollard](#) Apr 3, 2015 at 20:36



9



I saw an application re-architected within 2 years of its introduction into production, and others rewritten in different technologies (one was C++ - now Java). Both efforts were not, to my mind, successful.

I prefer a more evolutionary approach to bad software. If you can "componentize" your old app such that you can introduce your new requirements and interface with the old code, you can ease yourself into the new environment without having to "sell" the zero-value (from a biz perspective) investment in rewriting.

Suggested approach - write unit tests for the functionality with which you wish to interface to 1) ensure the code behaves as you expect and 2) provide a safety net for any refactoring that you may wish to do on the old base.

Bad code is the norm. I think IT gets a bad rap from business for favoring rewrites/rearchitecting/etc. They pay the money and "trust" us (as an industry) to deliver solid, extensible code. Sadly, business pressures frequently result in shortcuts that make the code unmaintainable. Sometimes it's bad programmers... sometimes bad situations.

To answer your rephrased question... can code maintenance costs ever exceed rewriting costs... the

answer is clearly yes. I don't see anything in your examples, however, that lead me to believe this is your case. I think those issues can be addressed with tests and refactoring.

Share Improve this answer

answered Sep 28, 2008 at 1:50

Follow



[Adrian Wible](#)

1,684 ● 1 ● 13 ● 20



8



In terms of business value, I would think it's extremely rare that a real case can be made for a rewrite due solely to the internal state of the code. If the product's customer-facing and is currently live and bringing in money (i.e. is not a mothballed or unreleased product), then consider that:

- You already have customers using it. They're familiar with it, and might have built some of their own assets around it. (Other systems that interface to it; products based on it; processes they'd have to change; staff they'd maybe have to retrain). All of this costs the customer money.
- Re-writing it *might* cost less in the long term than making difficult changes and fixes. But you can't quantify that yet, unless your app is no more complex than Hello World. And a re-write means a re-test and a redeploy, and probably an upgrade path for your customers.
- Who says the re-write will be any better? Can you honestly say your firm is writing sparkly code now?

Have the practices that turned the original code to spaghetti been corrected? (Even if the main culprit was a single developer, where were his peers and management, ensuring quality through reviews, testing, etc.?)

In terms of technical reasons, I'd suggest it could be time for a major rewrite if the original has some technical dependencies that have become problematic. e.g. a third party dependency that's now out of support, etc.

In general though, I think the most sensible move is to refactor piece by piece (very small pieces if it's really *that* bad), and improve the internal architecture incrementally rather than in one big drop.

Share Improve this answer

edited Apr 19, 2009 at 19:43

Follow

answered Oct 13, 2008 at 21:44



razlebe

7,144 ● 6 ● 44 ● 57



7



Two threads of thought on this one: Do you have the original requirements? Do you have confidence that the original requirements are accurate? What about test plans or unit tests? If you have those things in place it might be easier.



Putting on my customer hat, does the system work or is it unstable? If you've got something that's unstable you've



got an argument to change; otherwise you're best of refactoring it bit by bit.

Share Improve this answer

answered Sep 27, 2008 at 23:37

Follow



Martin Clarke

5,647 ● 7 ● 39 ● 58



7



I think the line in the sand is when basic maintenance is taking 25% - 50% longer than it should. There comes a time when maintaining legacy code becomes too costly. A number of factors contribute to the final decision. Time and cost being the most important factors I think.



Share Improve this answer

answered Sep 27, 2008 at 23:40



Follow



Shaun

716 ● 6 ● 11



6



If there are clean interfaces and you can cleanly delineate module boundaries, then it *might* be worth refactoring it module by module or layer by layer in order to allow you to migrate existing customers forward into cleaner more stable codebases, and over time, after you've refactored every module, you will have rewritten everything.



But, based on the codereview, doesn't sound like there would be any clean boundaries.

Share Improve this answer

answered Sep 28, 2008 at 0:29

Follow



Richard

184 ● 2 ● 7

-
- 1 Poor code often has more cleanly delimited module boundaries than good code, because there's little reuse of low level routines. Have a look at Michael Feathers Working with legacy code. – [MarkJ](#) Feb 15, 2009 at 21:11
-



5



I wonder if the people who vote for scrapping and starting over have ever successfully refactored a large project, or at least seen a large project in poor condition that they think could use a refactoring?

If anything, I err on the opposite side: I've seen 4 large projects that were a mess, that I advocated refactoring as opposed to rewriting. On a couple, there was barely a single line of original code that remained, and major interfaces changed in significant ways, but the process never involved the entire project failing to function as well as it originally did, for any more than a week. (And top-of-trunk was never broken).

Perhaps a project exists that is so severely broken that to attempt to refactor it would be doomed to failure, or perhaps one of the previous projects I refactored would have been better served by a "clean re-write", but I'm not sure I'd know how to recognize it.

Share Improve this answer

Follow

answered Sep 30, 2008 at 22:09



[KeyserSoze](#)

2,511 ● 1 ● 16 ● 17



4



I agree with Martin. You really need to weigh the effort that will be involved in writing the app from scratch against the current state of the app and how many people use it, do they like it, etc. Often we may want to completely start from scratch, but the cost far outweighs the benefit. I come across bits of ugly looking code all the time, but I soon realize that some of these 'ugly' areas are really bug fixes and make the program work correctly.

[Share](#) [Improve this answer](#)

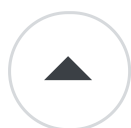
answered Sep 27, 2008 at 23:42

[Follow](#)



[Ed Swangren](#)

125k ● 24 ● 188 ● 268



4



I would try to consider the architecture of the system and see whether it is possible to scrap and rewrite specific well defined components without starting everything from scratch.

What would usually happen is that you can either do that (and then sell that to the customer/management), or that you find out that the code is such a horrible and tangled mess that you become even more convinced that you need a rewrite and have more convincing arguments for it (including: "if we engineer it right, we would never need to scrap the whole thing and do a third rewrite).

Slow maintenance would eventually cause that architectural drift that would make a rewrite more expensive later.



4



Scrap old code early and often. When in doubt, throw it out. The hard part is convincing non-technical folks of the cost-to-maintain.

So long as the value derived appears to be greater than the cost to operate and maintain, there's still positive value flowing from the software. The question surrounding a rewrite this: "will we get even more value from a rewrite?" Or alternatively "How much more value will we get from a rewrite?" How many person-hours of maintenance will you save?

Remember, the rewrite investment is once only. The return on the rewrite investment lasts forever. **Forever.**

Focus the value question down to specific issues. You listed a bunch of them above. Stick with that.

- "Will we get more value by reducing cost through dropping the junk that we don't use but still have to wade through?"
- "Will we get more value from dropping the junk that's unreliable and breaks?"
- "Will we get more value if we understand it -- not by documenting, but by replacing with something we built as a team?"

Do your homework. You'll have to confront the following show-stoppers. These will originate somewhere in your executive foodchain from someone who'll respond as follows:

- "Is it broken?" And when you say "It's not *crashed* as such," They'll say "It's not broke - don't fix it."
- "You've done the code analysis, you understand it, you no longer need to fix it."

What's your answer to them?

That's only the first hurdle. Here's the worst possible situation. This doesn't always happen, but it does happen with alarming frequency.

Someone in your executive foodchain will have this thought:

- "A rewrite doesn't create enough value. Rather than simply rewrite, let's expand it." The justification is that by creating *enough* value, users are more likely to buy in to the rewrite.

A project where scope is expanded -- artificially -- to add value is usually doomed.

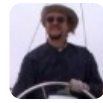
Instead, do the smallest rewrite you can to replace the darn thing. Then expand to fit real needs and add value.

Share Improve this answer

edited Sep 28, 2008 at 23:16

Follow

answered Sep 28, 2008 at 0:33



[S.Lott](#)

391k ● 82 ● 517 ● 788

-1 Martin Fowler says exactly the opposite to this. He advises gradually replacing (by expanding and improving). I know you've got a massive rep but I think you're wrong this time. martinfowler.com/bliki/StranglerApplication.html – [MarkJ](#) Mar 16, 2009 at 11:31

I'm talking about precisely the same approach Fowler is talking about -- incremental replacement. I'm suggesting that you replace pieces early and often. Eventually, the old system will be strangled out of existence. Do NOT create a huge project to replace it all at once. – [S.Lott](#) Mar 16, 2009 at 13:23



3



You can only give a definite yes to rewriting in case if you know completely how your application works (and by completely I mean it, not just having a general idea of how it should work) and you know more or less exactly how to make it better. Any other cases and it's a shot in the dark, it depends on too much things. Perhaps gradual refactoring would be safer if it is possible.

Share Improve this answer

Follow

answered Sep 27, 2008 at 23:44



[pilsetnieks](#)

10.4k ● 12 ● 50 ● 61



If possible, I typically would prefer to rewrite smaller portions of the code over time when I need to refactor a

2



baseline. There are typically many smaller issues such as magic number, poor commenting, etc. that tend to make the code look worse than it actually is. So, unless the baseline is just awful, keep the code and just make improvements at the same time you are maintaining the code.

If refactoring requires a lot of work, I recommend laying out a small re-design plan/todo list that gives you a list of things to work on in order so that you can bring the baseline to a better state. Starting from scratch is always a risky move and you are not guaranteed that the code will be better when you are finished. Using this technique, you will always have a working system that improves over time.

Share Improve this answer

Follow

answered Sep 28, 2008 at 1:25



[dr_pepper](#)

1,587 ● 1 ● 13 ● 28

good advice... its nice to not have to learn this the hard way
– [Shawn](#) Sep 28, 2008 at 3:30

I wonder if a concurrent approach could work. Maintain the current code (bug fixes only), but work on a new version in a completely separate project, starting from scratch, but borrowing from the current codebase where appropriate/possible. – [Mike Spross](#) Sep 28, 2008 at 3:49

You might have to do that if the baseline is really bad, but I would strongly consider modifying it incrementally. One reason is because it will get less mgmt visibility, and it will force you to understand what is wrong with the current baseline. – [dr_pepper](#) Sep 28, 2008 at 4:13



2



Code with excessively high cyclomatic complexity (like over 100 in a large number of modules) is a good clue. Also, how many bugs does it have / KLOC? How critical are the bugs? How often are bugs introduced when bug fixes are made. If your answer is *a lot* (I cant remember norms right now), then a rewrite is warranted.



Share Improve this answer

answered Sep 28, 2008 at 1:28

Follow



[torial](#)

13.1k ● 9 ● 65 ● 89



2



As early as possible. Whenever you get a premonition that your code is slowly turning into an ugly beast that is very likely to consume your soul and give you headaches, and you know the problem is in the underlying structure of the code (so any fix would be a hack, e.g. introduce a global variable), then it's time to start over.

For some reasons people don't like throwing away precious code, but if you feel your better off starting over, you are probably right. Trust your instinct and remember that it wasn't a waste of time, it taught you one more way of NOT approaching the problem. You could (should) always use a version control system so your baby is never really lost.

Share Improve this answer

answered Sep 28, 2008 at 8:57

Follow



Firas Assaad

25.7k ● 16 ● 63 ● 79



1



I do not have any experience with using metrics for this myself, but the article ["Software Maintainability Metrics Models in Practice"](#) discusses more or less the same question asked here for two case studies they did. It starts with the following editor's note:

In the past, when a maintainer received new code to maintain, the rule-of-thumb was "If you have to change more than 40 percent of someone else's code, you throw it out and start

over." The Maintainability Index [MI] addressed here gives a much more quantifiable method to determine when to "throw it out and start over." This work was sponsored by the U.S. Air Force Information Warfare Center and the U.S. Department of Energy [DOE], Idaho Field Office, DOE Contract No. DE-AC07-94ID13223.)

Share Improve this answer

answered Sep 28, 2008 at 2:06

Follow



[hlovdal](#)

28k ● 11 ● 100 ● 175



1

I think the rule was...

- The first version is always a throw away



So, if you learned your lesson(s), or his/her lessons, then you can go ahead and write it fresh now that you understand your problem domain better.



Not that there aren't parts that can/should be kept. Tested code is the most valuable code, so if it isn't deficient in any real way other than style, no reason to toss it *all* out.

Share Improve this answer

answered Sep 28, 2008 at 2:55

Follow



[Andrew](#)

8,654 ● 3 ● 50 ● 73

1 -1 That rule comes from Fred Brooks famous book "The Mythical Man Month". In the most recent edition, he says it

was the biggest mistake he made in the book. – [MarkJ](#) Mar 16, 2009 at 11:28



When is it good (if ever) to scrap production code and start over?

1



Never had to do this, but logic would dictate (to me, anyway) that once you pass the inflection point where you're spending more time reworking and fixing bugs in the existing code base than you are adding new functionality, it's time to trash the old stuff and get a fresh start.



Share Improve this answer

Follow

answered Oct 13, 2008 at 21:53



[Lieutenant Frost](#)

541 ● 2 ● 7



If it requires more time to read and understand the code (if that is even possible) than it would to rewrite the entire application, I say scrap it and start over.

0



Share Improve this answer

Follow

answered Sep 28, 2008 at 0:12



[Micah](#)

116k ● 87 ● 237 ● 331



I have never completely thrown out code. Even when going from a foxpro system to a c# system.

0

If the old system worked then why just throw it out?



I have come across a few really bad system. Threads being used where not needed. Horrible inheritance and abuse of interfaces.



It is best to understand what the old code is doing and why it is doing it. Then change it so that it is not confusing.

Of course if the old code doesn't work. I mean can't even compile. Then you might be justified in just starting over. But how often does that actually happen?

Share Improve this answer

answered Sep 28, 2008 at 0:43

Follow



ElGringoGrande

638 ● 6 ● 13



Yes, it totally can happen. I've seen money be saved by doing it.

0



This is not a tech decision, it's a business decision. Code rewrites are long term gains, while "if it ain't *totally* broke..." is a short term gain. If you are in a first year startup that is focused on getting a product out the door, the answer is usually to just live with it. If you're in an established company, or the errors with the current systems are causing more workload, therefor more company money.. then they might go for it.



Present the problem as best as you can to your GM, use dollar values where you can. "I don't like dealing with it" means nothing. "It'll take twice the time to do everything until this is fixed" means a lot.

Share Improve this answer

answered Sep 28, 2008 at 1:15

Follow



Brent

23.7k ● 10 ● 47 ● 49



I think there are a number of issues here that depend largely on where you are at.

0



Is the software working well from a customer perspective? (If yes be very careful about changes). I would think there would be little point re-witting unless you were expanding the feature set if the system was working. And are you planning to expand the features and customer base of the software? If so then you have much more reason to change.



As much as anything just trying to understand some else's code even if well written can be difficult, when badly written I would imagine almost impossible. What you describe sounds like something that would be very difficult to expand.

Share Improve this answer

answered Sep 28, 2008 at 1:21

Follow



David L Morris

1,501 ● 1 ● 12 ● 20



0



I would take into consideration if the application does what it is intended to do, is required for you to ever make modifications, and are you confident that the app has been thoroughly tested in all scenarios that it will be used in.

Do not invest the time if the app does not need alterations. However, if it doesn't function as you need and you need to control the hours and time invested to make corrections, scrap it and re-write to the standards that your team can support. There's nothing worse than terrible code that you have to support / decipher but still have to live with. Remember, Murphy's Law says it will 10 at night when you'll have to make things work, and that is never productive.

Share Improve this answer

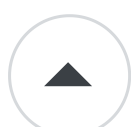
Follow

answered Sep 28, 2008 at 1:36



[David Robbins](#)

10k ● 7 ● 54 ● 83



0



Production code always has some value. The only case where I would truly throw it all out and start again is if we determine the intellectual property is irrevocably contaminated. For example if someone brought large amounts of code from a previous employer, or a large percentage of the code was ripped from a GPLd codebase.

answered Sep 28, 2008 at 2:21

Share Improve this answer

Follow



DGentry

16.3k ● 8 ● 53 ● 66



0



I'm going to post this book every time I see a discussion on Refactoring. Everyone should read "Working Effectively with Legacy Code" by Michael Feathers. I found it to be an excellent book - if nothing else, it's a fun read, and motivational.



Share Improve this answer

answered Sep 30, 2008 at 3:07



Follow



Matt Cruikshank

2,948 ● 21 ● 24



0



When the code has reached a point that is not maintainable or extensible anymore. Is full of short-term hacky fixes. It has lots of coupling. It has long (100+lines) methods. It has database access in the UI. It generates a lot of random, impossible to debug errors.



Bottom line: When maintaining it is more expensive (i.e. takes longer) than rewriting it.



Share Improve this answer

answered Oct 3, 2008 at 19:02

Follow



Ricardo Villamil

5,097 ● 3 ● 32 ● 26



0

I used to believe in just re-write from scratch, but it is wrong.



<http://www.joelonsoftware.com/articles/fog0000000069.html>



Changed my mind.



What I would suggested is figuring out a way to properly refactor the code. Keep all existing functionality and test as you go. We have all seen horrible code bases, but it is important to keep the knowledge over time you application has.

Share Improve this answer

answered Mar 16, 2009 at 20:48

Follow



David Basarab

73.2k ● 43 ● 130 ● 157
