

I was only a year and a half out of college when I found I no longer enjoyed programming. I had worked for several software consulting companies, and I had begun to see a common theme in their corporate cultures—slow to progress, slow to change, and slow to notice problems (let alone to fix them). When efforts were made to fix a problem, companies invested heavily in processes and tools but did little to address the *real* problems. This resulted in a lot of work

(tion). It's not just a catch phrase—it's a philosophy. As a programmer and problem solver, it is, above all else, an attitude. It's an attitude that lets you tear apart a problem, analyze it, and solve it to the best of your ability. It's an attitude that abhors hacks, shortcuts, band-aids, and workarounds. It's an attitude that sweeps away all the things that don't matter. It's an attitude that peels away the issues that keep us from a good solution and gets at the heart of a problem.

Solve the *Real Problem*



A FORMULA FOR SUSTAINABLE SOLUTIONS

by Tim Beck

that felt wasted and software projects that felt like death marches. I was looking for something that seemed elusive—the ability for a group of software developers to look at a problem, analyze it, determine the real issue at hand, and solve it in an elegant and effective way.

On my wife's advice, I decided to try one more company. On my first day, I walked into the development area of the office and knew something was different. On the wall was a big banner that read "Solve the Real Problem." I didn't fully understand what that meant at the time, but it turned out to be exactly what I was missing in my previous jobs. This simple phrase, put into action, changed the course of my programming career.

Two senior developers, Brad Spencer and David Benoit, taught me what "Solve the Real Problem" really means (see the StickyNotes for more informa-

"Solve the Real Problem" guided the work that we did. Brad and David taught me how to apply it directly to the technical challenges we faced. They went out of their way to make sure I understood not only the attitude and the theory behind it but also how to put the theory into practice.

Real Problems in Real Life

Now that I know what to look for, I often find myself in the position to solve the "real problem." One such problem occurred while I was working on an enterprise application that had been handed down to me from a programmer who had moved on to a different team. While development was already underway, the application didn't have much more completed than a basic user interface and a data access layer. To get up to speed on what I was working on, I began reading the code.



The data access layer was interesting, and at first glance, it promised to make my task much easier than I had expected. All the data was accessed through a custom library that had a generic interface. The interface provided simple, clean access to the data, without the programmer's needing to know the internal workings of the library. I could call this library and ask for the data I needed without having to go directly to the data source. The interface would return a collection of data for me that I could easily utilize.

Custom libraries exist for these kinds of things—particularly for Web applications—but in this technology, it was a pleasant surprise. It was pretty slick, so I began to use this generic data access layer. The groundwork had been done for me, so all I needed to do was use it in my own

development work and extend its capabilities when needed.

After working with it for a while, I needed new types of functions, which became progressively more complex. No problem. I extended the data classes in the data access layer for each of the new cases that I came across while maintaining the integrity of the component. This helped build my confidence in the implementation.

However, my work raised questions about the data access layer. Some of the features in the interface were written but never used. From experience I've learned that this is often a red flag. Martin Fowler calls this kind of thing a "code smell" and, more specifically, "speculative generality." Initially, we have good intentions and think we might need features in our code, but for whatever reason, we don't implement them fully. The code ends up just sitting there, unused.

In addition, there was a lot of redundancy in the library. It certainly did not follow the "Don't Repeat Yourself" principle. I would never really know if there were valid reasons the previous developer had done it that way, but this code smell added to my concerns about the data access layer as a whole.

While testing the new features I was developing, my suspicions about the data access layer were confirmed. Something was wrong. The final straw was when I noticed that, for a certain subset of data, collections were not being returned completely. Looking at source data and what I was getting back from the data access layer, it was obvious that the data access layer was truncating—or wasn't even aware of—the entire set of data. This was more than just some unused code or redundancy. This was a major issue that had a direct impact on the end-user experience. Not all issues with a software component are worth fixing, but when enough issues are uncovered, a re-evaluation is in order.

Evaluate Your Problem

At this point, it was time to stop and evaluate what I was doing. Was I trying to solve the real problem, or, as Elisabeth Hendrickson asks, was I trying to solve a symptom of the problem? (See the Sticky-

Notes for more information.) If it's merely a symptom of a larger problem, I am going to churn as I code, moving from one "problem" to the next, not addressing the actual cause. I've learned to watch for warning signs when solving a problem, and so far, I had seen unused code and my implementation of a simple interface action becoming more and more complex. However, the facts that the interface was not reliable and that I was not getting any farther ahead in my own development were stop signs.

My first thought was to add code to the data access layer to check for the special cases where data was not being returned properly. The cases were fairly obvious and generally detectable so it wouldn't be too hard. That would probably get me through this immediate problem, but what about the next new feature I needed to develop? Could the data access layer handle cases other than what I had developed to date? When I find myself in these kinds of situations, I pause and evaluate where I am. When you are deep in the details of code, it can be too easy to get stuck in a loop of fix-test-fix-test and not get anywhere on the feature you are trying to develop. It's good to step back once in a while and look at the big picture.

I don't like to move ahead and start coding a solution until I have a firm grasp of what I'm doing, so I traded in my screen and keyboard for a whiteboard. I mapped out what I still needed to develop and matched that to the data access layer. As I mapped features to the existing data access layer capabilities, it didn't take me long to understand that the design of the data access layer was my real problem—not specific isolated functions that dealt with special cases. The data access layer was a good first attempt, but the current implementation was made up of overly complex code that could only handle simple business cases. Extending it to handle the cases I needed to implement would be awkward and time consuming. Furthermore, the current implementation was so limiting that I couldn't see how I could force the code to fit the solution without introducing more problems.

Initially I thought my problem was merely a few unhandled data cases, but critical thinking and analysis revealed an

incomplete implementation. Based on the symptoms, my diagnosis seemed sound. I told my manager that I needed a few days to sort out this problem before I could implement new features, and then I went back to the whiteboard and began sketching out a new basic design for the data access layer. The original implementation had some good, usable, basic concepts, such as: providing a clean, generic, simple interface; connecting to a data source; and returning a collection of data. I would use these basic ideas as I moved forward, but I would implement something that was more robust and able to handle the complex scenarios the application required.

Define Checkpoints

I needed to have a clear idea of the problem I was trying to solve before I felt comfortable writing code, so I explored the data needs of the application. It can be tempting to write only enough code to solve the problem right in front of you. This is a popular concept in the agile/XP community, but too little planning is just as bad as too much. If you aren't looking beyond the current unit of code to where you need to go, you can paint yourself into a corner. I didn't want to end up in a situation where the current solution passed its unit tests but was still defined too narrowly. It's easy enough to do.

Once I explored the data requirements we would need for the entire project and felt I understood the differences between valid data, invalid data, and valid-yet-incomplete data, I finished sketching out my lightweight design on the whiteboard. I spent time thinking this through before I started programming and identified checkpoints for my design based on the various requirements that would be needed. Checkpoints are important to establish early, because they will serve as tests as I develop the implementation.

My design was based on simplicity and good programming practices. Once I started implementing my design in code, I used my tests to determine if my design was emerging correctly. Sometimes I found my assumptions were wrong, and my tests would reveal a problem in my design. I would revisit the problem, learn more about it, and solve it in accordance

with the overall vision. My checkpoints and tests told me whether I was meeting the overall goal.

I look for clarity in my design and code, and I resist the temptation to use tools, patterns, frameworks, and processes that add complexity. Sometimes they appear to be an easy fix, but they can distract the programmer from the real problem. When programming, I often ask myself "Is there a simpler solution?" I review my code periodically, and if I see that a design pattern helps make my solution simpler while still keeping the code true to the overall goal, I'll use it. If a fashionable design pattern looks similar, but adds unnecessary complexity, I don't implement it just for the sake of using a design pattern.

Re-evaluate When Needed

As I develop, I watch for signs along the way and iterate toward a solution that fits the overall design goals. When failing tests provide evidence that my code is missing the mark, I re-evaluate what I'm doing. The previous effort isn't wasted, because I have learned new information through the process that helps me evaluate whether I'm solving the problem. Once I have a clear idea of what is going wrong, I may try a quick proof of concept. If the proof of concept works, I briefly revisit the design and reconfigure it so that I once again have confidence to move forward programming the solution. Only then do I look at what tools and patterns I have at my disposal to help me continue solving the problem.

When programming, I use my tests to help drive out my emerging design. Are the tests too simple? That's a sign that I need to revisit my design and write more realistic test cases. Code smells are a good indication that the problem isn't being solved correctly, so I use tests to confirm


my design goals and revisit my code implementation to make sure it is true to my overall design. Design isn't static—tests and learning can drive the design in different directions, but if I have a clear overall picture, I can quickly tell if I'm still on track. In this case, I had passing tests and was happy with an emerging design.

In short order I was finished, and the new data access layer worked well. It contained less code and removed some previous hacks. It made overall data access much simpler and more reliable. In the development process, I wrote automated unit tests, cleaned up related code, and did a lot of hands-on testing. Not only did the data access layer actually work in all the cases we needed to use in the application, but there was now a firm foundation on which to make future refactoring decisions. Overall, I felt I had solved the real problem and created a solution that was solid and forward looking. In fact, months later I returned to the solution to add a caching feature to improve performance. I was able to do this quickly and with confidence, which lent credibility to my initial design.

Overcome Roadblocks

General problem solving can also benefit from this kind of thinking. In systems, roadblocks appear over time that distract us from solving problems as well as we might. Roadblocks can seem so powerful that they preclude anyone from challenging them. Instead of accepting these roadblocks, we need to have the courage to question them.

One common roadblock is the "no time" myth. We've all experienced times we didn't want to cut corners or implement a hack. Unfortunately, due to pressure—either explicit or implicit—we succumb to the temptation just to slap together something that works. We are either being told



"I look for clarity in my design and code, and I resist the temptation to use tools, patterns, frameworks, and processes that add complexity."

or telling ourselves that there is no time to do it properly. There are many side effects that come from the “no time” attitude, but one that greatly impacts programmers is the limitation it puts on the quality of solutions they create. Without time to try to solve problems correctly, a company’s development team is in danger of causing expensive mistakes. I’m not talking about the strategic, short-term, risk-managed decisions that businesses sometimes have to make. I’m talking about the long-term, chronic attitude that there is never enough time to solve a problem correctly. If there is time to solve the problem incorrectly and deal with the aftermath of having solved the problem incorrectly, then there is surely time to solve the problem correctly.

Fear is another roadblock. It takes courage to stand up and solve the real problem. Determining problems from symptoms can be difficult, but with practice and continuous improvement we can get

better and better at it. We can’t expect to always implement the best solutions, but with the right attitude we can continue to improve over time if we learn and adapt. Software development is an evolutionary process that asymptotically approaches the production of a perfect system. We may never get there, but if we constantly improve our problem solving, we will get that much closer.

Conclusion

“Solve the Real Problem” is not limited to software development issues. It is a good philosophy to guide any kind of problem-solving activity. In many situations, it’s easy to be distracted by fashionable processes, tools, and practices. “Solve the Real Problem” serves as a reminder to focus on the unique problems we face in our own contexts. It’s important to remember that every situation is different. There is no single process that will be successful in all situations in life let

alone in software development. My challenge to you is to work toward finding your own solutions to the unique problems you face and improve on them. A good place to start is to print out for your team a large banner that reads:

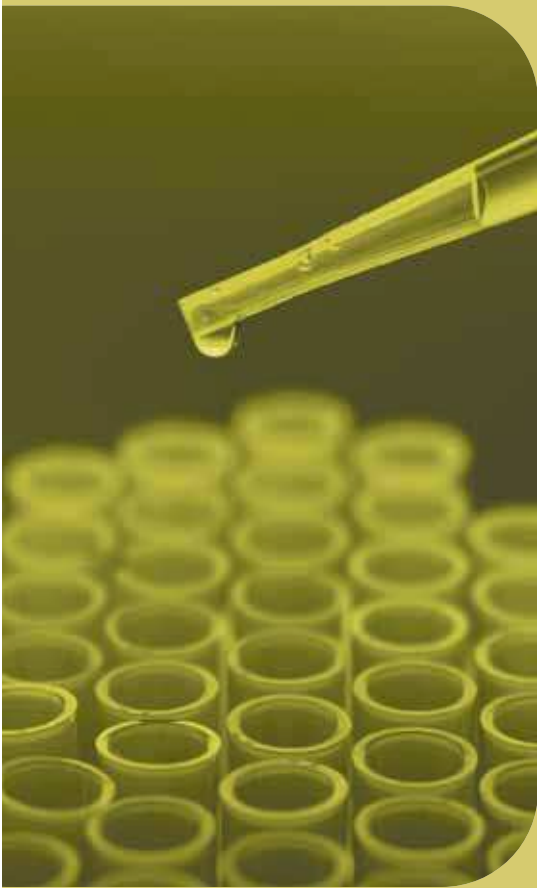
“Solve the Real Problem.” Then, solve the real problem. Period. **{end}**

Tim Beck is a software developer and entrepreneur living in Calgary, Alberta, Canada. He is the founder of the Pliant Alliance. Check out more of his ideas at <http://pliantalliance.org>.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware.

- Brad Spencer
- David Benoit
- Elisabeth Hendrickson



How to Solve the Real Problem

Here are some ideas I use when trying to solve the real problem:

- Spot problems when they occur. This is where learning from experience comes in. Learn from problems you have seen in the past and how to spot similar ones as they occur.
- Don’t start trying to solve the problem in front of you until you have a clear idea of what you are doing.
- Don’t give in to pressure and fear when you are uncomfortable with the status quo. It’s OK to say no.
- Use checkpoints to see if the implementation you develop matches the goals of the design. Different kinds of test cases are helpful checkpoints.
- Be thoughtful and watch for signs along the way. If things go wrong, take the knowledge you have gained and build on it now that you have a better idea of what the real problem might be.
- Look for clarity and elegance in the implementation of the design—avoid complex solutions to simple problems, and try not to get distracted by fashionable design patterns, frameworks, tools, and processes.
- Strive for improvement.