Extreme Programming vs. Interaction Design

When two development design visionaries meet, there's room for consensus—but not much.
by Elden Nelson

Posted January 15, 2002

Kent Beck is known as the father of "extreme programming," a process created to help developers design and build software that effectively meets user expectations. Alan Cooper is the prime proponent of interaction design, a process with similar goals but different methodology. We brought these two visionaries together to compare philosophies, looking for points of consensus—and points of irreconcilable difference.

The Starting Line
Let's root this conversation in real life. You've got an important development contract, but in the past, the customer has just given you a one-line description of his high-level concept, and that's about as specific as he's been. What do you do?

Beck: That was exactly the problem one of my clients faced last week. We had product managers who wanted to say things like, "force this product to OS-390." Then, three months later, they wanted to come back and have the results. So they're getting answers like, "Oh, it's going to take us three more months."

You could try and do a better job on the engineering side of getting the details of exactly what they meant by this and spending more and more time on estimation, requirements, specifications, engineering methodology, computer-aided yadda yadda yadda *forever* … and never reach anything like a satisfactory conclusion. Or you can say, "Customer, your job in this process has to be more than just giving us the bullet item; you're going to have to take a half step toward engineering, and engineering is going to take a half step toward you." You take that one bullet item and turn it into twelve things you care about and would recognize as signs of progress along the road toward delivering this product on OS-390.

If you click off one of those every week, after two weeks, you're going to have a pretty good idea whether you're going to get what you want at the end of three months. After four weeks, you're going to be absolutely certain whether you're going to get what you originally wanted at the end of four months.

It does mean that the specification of functionality on the business side has to break down one more level. And to make it as powerful as possible, it has to also come with a lot of activities that used to be thought of as post-hoc testing, where you'd say, "We want the system to do this, and that means—here's a little script by which I mean is really something automated, and that script will run when and only when the one-twelfth of our whole system has been added."

So, if you want better results, if you want more predictability out of this process, the strobe light has to flash more than once every six months. If it flashes every week, you add up a bunch of those and you're going to have a sense of how this works. And there are some trade-offs. If, for the customer, the pain of seeing into the project every week or two weeks is greater than the pain of the disappointment every six months, then I guess the status quo is OK. But if they want to have more control over this process, they're going to have to be engaged in the loop—in this week-to-two-week kind of loop.

Can XP work without that key element?

Beck: From the customer's perspective, no. I've had teams be called "whiners" because after 25 percent of the budget is spent, they're saying, "We have 10 features to add and we're going at half the speed that we expected. Which five would you like us to work on first?" And the customer says, "Oh, you whiners. Work some overtime or just get back to work or quit complaining." What do you say in a situation like that?

I don't know. What do you say?

Beck: You say, "I quit." Life's too short to work on doomed projects you already know are doomed after 25 percent of the budget is spent.

Alan, how would you handle the problem?

Cooper: I think XP has some really deep, deep tacit assumptions going on, and I think the deepest tacit assumption is that we have a significant organizational problem, but we can't fix the organization.

Essentially, the crap rolls downhill and ends up rolling right into the programmer's lap. When the product or the program turns out to be unsatisfactory, the fingers point to the programmer. XP is very well-intentioned; it's the software-development community beginning to say, "Hey, this is not only unfair to us, but it's not productive as a discipline and we can do a lot better." I applaud that sentiment and I agree with that sentiment, but then XP says, "OK, so, I can't change the organizational failings, so, I'm going to build my own internal defenses." I suppose this is probably better than nothing, but I'm interested in changing the way organizations are constructed. I believe that in order to create quality software, you have to change the organization. We can change the organization, and it strikes me that the assumption underlying XP is that the organization's structure is a given.

Beck: No, it's absolutely not. I spent my entire week explaining to the marketing people and the sales people and the customer-support people how they needed to be engaged as members of the team doing software development.

Cooper: That's not organizational change. That's just turning up the knob.

Beck: No, it most certainly is. We're changing the way people will be evaluated, changing where people sit, changing job responsibilities, creating entire new bundles of responsibilities to shift the planning process more into the customer realm, so they can see progress on things that they want. How does that not add up to organizational change?

Cooper: It's not my definition of organizational change.

Beck: So, what is?

Cooper: It's my experience that neither users nor customers can articulate what it is they want, nor can they evaluate it when they see it.

Neither the people who buy software nor the people who use it have the capability of visualizing something as complex as the behavior of software. They also don't have the ability to analyze what appropriate behavior is.

Now, I think that there are two sides to software—there's the side that touches the hardware, the central processing unit and peripherals, and it has one type of defining behavior—you know, it's fast and error-free and deterministic. Then on the other side, there's the human side, and humans are slow and error-prone and they're inferential and judgmental and emotional. In my experience, the skills that make you good at one of those things are no help at all on the other thing.

So, what happens is you take professional logicians—programmers—and they will look at the problem from that logical point of view. They'll look at it from, "How do we structure this code? They'll look at, "What is the code that I have to build?" Then when they say, "Let's look at the human issues," they bring those same logical tools to the party.

On the other side, you have the customer and/or user, and they tend to do what we call "automating the pain." They say, "What is it we're doing now? How would that look if we automated it?" Whereas, what the design process should properly be is one of saying, "What are the goals we're trying to accomplish and how can we get rid of all this task crap?"

When you put those two constituencies together, they come up with a solution that is better than not having those two constituencies together, I grant that. But I don't believe that it is a solution for the long term; I believe that defining the behavior of software-based products and services is incredibly difficult. It has to be done from the point of view of understanding and visualizing the behavior of complex systems, *not* the construction of complex systems.

The way the industry works right now is the initial cut at a solution is generally made from the point of view of a feature list that comes from the marketing people or the in-house customer, then given to the developers, who then synthesize a solution. XP makes their communication much more efficient and clearer and robust. But neither of them is properly equipped to solve the problem. It's not a construction problem; it's really a problem of design—not interface design, but behavior design.

And companies, whether they're product companies or corporate IT structures, have no place for behavior design in their current structure, because manufactured products from the Industrial Age didn't have behavior. So when I talk about organizational change, I'm not talking about having a more robust

communication between two constituencies who are not addressing the appropriate problem. I'm talking about incorporating a new constituency that focuses exclusively on the behavioral issues. And the behavioral issues need to be addressed before construction begins.

The Point of Departure: Point of Entry

Beck: OK, wait. I agreed with you very close to 100 percent, then you just stepped off the rails. I don't see why this new specialist has to do his or her job before construction begins?

Cooper: It has to happen first because programming is so hellishly expensive and the cost of programming is like the cost of pulling teeth. I can go have my tooth pulled for about a hundred bucks. OK? But what's the cost of having your tooth pulled? You don't have a tooth there. OK? And that's what's going on in the world of software. There's enormous cost in writing code, but the real cost in writing code is that code never dies. If you can think this stuff through before you start pouring the concrete of code, you get significantly better results. You get significantly more efficient programming and you don't end up with pulled teeth.

Building software isn't like slapping a shack together; it's more like building a 50-story office building or a giant dam.

Beck: I think it's nothing like those. If you build a skyscraper 50 stories high, you can't decide at that point, oh, we need another 50 stories and go jack it all up and put in a bigger foundation.

Cooper: That's precisely my point.

Beck: But in the software world, that's daily business.

Cooper: That's pissing money away and leaving scar tissue.

Beck: No. I'm going to be the programming fairy for you, Alan. I'm going to give you a process where programming doesn't hurt like that—where, in fact, it gives you information; it can make your job better, and it doesn't hurt like that. Now, is it still true that you need to do all of your work before you start?

Cooper: What do you mean, "make my job better"?

Beck: For example, suppose there are two styles of interaction, one of which costs one one-hundredth of the other and is 95 percent as effective. Would you like—when would you like to know that? So, the return on investment for these two possibilities is dramatically different. Would you like to know that before you decided which style to use?

Cooper: Well, the cheapest thing is to just stay home and not do it.

Beck: I'll try and make it very concrete. I have some tabular presentation of data and I would like to be able to sort all by all the different columns—ascending, descending, and with a little scripting language that lets me combine all of those in any order I choose. And the programmer comes back and says, "That's very expensive." And an interaction designer says, "Oh, really, I didn't know that was expensive. Well…."

Cooper: See, this is where I come off the tracks. You're positing that your customer can tell you the correct answer. Then you can't give it to him because it's too expensive. What I'm saying is, number one, if your customer could give you a correct answer, which I absolutely believe they can't, then what are the issues about cost? I mean, it's the engineer's job to keep the cost down. It's not the engineer's job to say, I'm going to keep cost down by not giving you that. Of course, the engineer will choose the cheapest way to build it. If the engineer doesn't choose the cheapest way to build it, then the engineer is not competent at his or her job.

What you're positing is that there are practitioners out there who do stupid stuff. And there are, in fact, "interface designers" out there who do that. I'm not an interface designer, and I'm not advocating interface design. I'm advocating *interaction* design, which is much more akin to requirements planning than it is to interface design. I don't really care that much about buttons and tabs; it's not significant. And I'm perfectly happy to let programmers deal with a lot of that stuff.

In the work that we're doing, we're changing the face of organizations as much as we're changing the face of their software. We're working with companies right now that have 50 or 60 internal departments, each with a Web presence. They're trying to solve the problems of redundancy and overlap and all that stuff as

if it were an interface issue—they want commonality in the interface. In reality, it's not about commonality of the interface. The big problem is, how do you deal with those 50 or 60 divisions? How do you get them working together? If you go to the customers, have a whole series of discussions with them, then build what they want, when all is said and done, you're going to end up with a really well-constructed white elephant that doesn't solve their problem. Their problem is, they have to integrate information and their company at a much higher level. These are not constructionary interface issues.

Looking for Consensus

Beck: Well, I think I can grant you pretty much everything that you said about finding that larger picture without the need for hierarchy or phases. You say in your book that the first thing we have to do with the process is specify all the interaction design before we write a line of code. That sounds like design phase, implementation phase to me.

The interaction designer becomes a bottleneck, because all the decision-making comes to this one central point. This creates a hierarchical communication structure, and my philosophy is more on the complex-system side—that software development shouldn't be composed of phases. If you look at a decade of software development and a minute of software development, you ought to have a process that looks really quite similar, and XP satisfies that. And, secondly, the appropriate social structure is not a hierarchical one, but a network structure.

Cooper: Those are really good points, and you and I are very close on this. Let me see if I can address this. The impression I get from XP is that it addresses the problems of corporate developers. And I sense you have a deep resentment and anger and frustration at this idea of phases. And while I don't think there's anything wrong with phases *per se*, I think it's wrong when phases are abused, namely when phases have arbitrary boundaries and when there's no recourse and the people who are participating in the various phases are not working together.

You're worried that interaction design becomes a bottleneck. You've probably seen abuses in the past, and I certainly have seen abuses in the past. I see very few organizations that think in terms of interaction design as part of the requirements-planning or product-planning process. They see it as something that comes *after* product planning is done—and when you do interaction design after product planning has been done, you've got a bottleneck, regardless of who does it.

When an architect begins to define a building, he or she works very closely with the people who are buying the building to understand what the requirements are, and translate those requirements into a sketch of a solution. Then there's a lot of give-and-take between the occupants of the building and the architect in coming up with a viable solution. Usually, the architect at the sketch level will know enough not to design something that's an engineering problem. And if the architect does detect that there might be problems, he or she will consult with an engineer to make sure that he or she is not painting himself into a corner, technically speaking. At a certain point, the architect and the customer are going to achieve common ground. At that point, the architect turns those sketches into detailed drawings. Now, during the detailed-drawing phase, there's a lot more intense interaction between the architect and the engineer. The architect, you know, draws a span and calls in the engineer and says, how big a girder do I need to support this span? And there's a lot of detailed interaction between the two. This is *precisely* what happens in the world of good interaction design.

You notice that the person who's going to live in that building doesn't talk to the builder about the size of the girders. The architect makes the job of the builder much easier because the builder isn't going to have all those facts and figures. If he made these decisions, he might say, "Okay, there has to be an open span here, but I don't know how big it is, so I'll build it 30 feet." Then the user comes in and says, "I think it needs to be bigger." What happens then is you end up with all this scar tissue inside your building. You've got to pull out the smaller beams and put in the bigger beams, but that's too expensive, so you end up scabbing a slightly bigger beam onto the slightly smaller beam, and then when you want to add a story later on, you know, you can't do it.

Beck: This scar-tissue thing keeps coming up. In the XP team that I've worked with—and those that I admire that I haven't worked with—the system becomes *more* capable over time, not less capable.

One of my goals in XP was creating teams that even in highly constrained and emotionally charged business environments could produce programs that didn't *lose* their ability over time but, in fact, *gained*

capacity over time. As components are factored out, as configurations emerge from the customer team—which you might call the interaction team—the program is still getting better and better in five years. The team gets the attitude where they go back to the customers and say, "You're giving us easy stuff. Give us a hard one." The customer team is actually challenged, over time, to think bigger and bigger thoughts as development goes on.

That's one of the base assumptions that seems to be very different in our thinking—that you can build a program that improves with age instead of having it born powerful and gradually degrading until you just think, "Oh, this is crap and we have to start over."

Can We Get Along?

Beck: I would love to find a team where you got to coach the customer side and I got to coach the engineers. I think we could kick some serious ass doing that.

Cooper: Yeah, except with a slight difference. You would coach the development team and I would coach the interaction design team, and the interaction design team would act as a bridge between the customer and the developers. The thing is, the customer is just not going be imaginative and will not visualize a "bigger thoughts"; their "bigger thoughts" are going to be very small.

Beck: OK, don't do that. How about if we were to build a product and we had 60 people to do it. You get to direct the efforts of 30 and I get to direct the efforts of 30. Here's the constraint—I give you one week to tell us the first thing you'd like to see.

Cooper: I wouldn't know what to do with 30 people. We've worked with many different sizes of design teams and we've settled on the optimum size—it's two. So, I would want only two people to do the design work. However, I want more than a week.

Beck: Yeah, I know you do, but that that's why I gave you that constraint. We can't let those phases leak back in.

Cooper: Look. During the design phase, the interaction designer works closely with the customers. During the detailed design phase, the interaction designer works closely with the programmers. There's a crossover point in the beginning of the design phase where the programmers work for the designer. Then, at a certain point the leadership changes so that now the designers work for the implementers. You could call these "phases"—I don't—but it's working together. Now, the problem is, a lot of the solutions in XP are solutions to problems that don't envision the presence of interaction design—competent interaction design—in a well-formed organization. And, admittedly, there aren't a lot of competent interaction designers out there, and there are even fewer well-formed organizations.

Beck: Well, what we would call "story writing" in XP—that is, breaking a very large problem into concrete steps from the non-programming perspective still constitutes progress—that is out of scope for XP. But the balance of power between the "what-should-be-done" and the "doing" needs to maintained, and that the feedback between those should start quickly and continue at a steady and short pace, like a week or two weeks. Nothing you've said so far—except that you haven't done it that way—suggests to me that that's contradictory to anything that you've said.

Cooper: I agree with that, and I think that that's the primary focus of interaction design. I think that the—there's a single point where Larry Constantine and I diverge, and that is that he asserts that this is an engineering discipline and I don't.

The problem is that interaction design accepts XP, but XP does not accept interaction design. That's what bothers me.

Beck: For instance?

Cooper: Yeah. The instant you start coding, you set a trajectory that is substantially unchangeable. If you try, you run into all sorts of problems, not the least of which is that the programmers themselves don't want to change. They don't want to have to redo all that stuff. And they're justified in not wanting to have to change. They've had their chain jerked so many times by people who don't know what they're talking about. This is one of the fundamental assumptions, I think, that underlies XP—that the requirements will be shifting—and XP is a tool for getting a grip on those shifting requirements and tracking them much more closely. Interaction design, on the other hand, is not a tool for getting a grip on shifting

requirements; it is a tool for damping those shifting requirements by seeing through the inability to articulate problems and solutions on management's side, and articulating them for the developers. It's a much, much more efficient way, from a design point of view, from a business point of view, and from a development point of view. Do you see that?

Beck: Yeah, but I would say exactly the opposite. For all its Dilbertesque qualities—and I'm still proud of having said "Embrace Change" in the title of the first XP book. I've consciously decided to give up the ability to predict the future.

Cooper: I see that in XP. It's an abdication.

Beck: Is Zen an abdication?

Cooper: XP assumes the people you're working with are going to jerk your chain, so you embrace that and work with it. And I actually think XP is a pretty good methodology for dealing with that. I'm saying, "Let's go to the root of the problem, which is the people who run our companies and jerk our chains back and forth, because they don't understand software." You can't teach them to understand software.

Rather than "embrace change," I would say, "embrace quality." If you think things through, you solve problems. Look at how something like the Palm Pilot just sliced through the marketplace; that's because it was designed—and it just sliced through the marketplace.

Beck: If we were in logic class, you would have just gotten a demerit. You can't say that the success of the Palm Pilot proves that design is the most successful way to do things. The Palm Pilot would have been better if it *had* been evolved. If they had cut code the first week, we would have had a much better product.

With all this back and forth, I still wonder: Kent, can interaction design make sense inside XP?

Beck: XP says that you have a conversation between the "customer team" and the "engineering team." The customer team's job is to take a big problem and say which step forward we'd like to take. They use "stories"—little pieces of functionality that are doable in a short cycle, like a week or two weeks. You can write an automated test that allows you to succeed or fail on the basis of the presence or absence of that story.

What I've noticed is, the teams that write the automated tests in advance of implementation do a much better job of both thinking about the thing and tracking their progress. Interaction design, to me, has a role to play from the moment of inception of a project, as a way of coming up with the metaphors for the system. In discovering those metaphors. I would love to have Alan on a team of mine in that story writing.

Cooper: What you have defined is the role of interface designers, not the role of interaction designers. Interaction designers don't do metaphors and don't do ways of information presentation—what they do is deal with behavior and with what the solution is going to be. "Interaction designer" is not title inflation for "Interface designer."

Cooper: The interaction designers would begin a field study of the businesspeople and what they're trying to accomplish, and of the staff in the organization and what problems they're trying to solve. Then I would do a lot of field work talking to users, trying to understand what their goals are and trying to get an understanding of how would you differentiate the user community. Then, we would apply our goal-directed method to this to develop a set of user personas that we use as our creation and testing tools. Then, we would begin our transformal process of sketching out a solution of how the product would behave and what problems it would solve.

Next, we go through a period of back-and-forth, communicating what we're proposing and why, so that they can have buy-in. When they consent, we create a detailed set of blueprints for the behavior of that product.

As we get more and more detailed in the description of the behavior, we're talking to the developers to make sure they understand it and can tell us their point of view.

At a certain point, the detailed blueprints would be complete and they would be known by both sides. Then there would be a semiformal passing of the baton to the development team where they would begin

construction. At this point, all the tenets of XP would go into play, with a couple of exceptions. First, while requirements always shift, the interaction design gives you a high-level solution that's of a much better quality than you would get by talking directly to customers. Second, the amount of shifting that goes on should be reduced by three or four orders of magnitude.

Beck: I'll divide what Alan is talking about into two things: a set of techniques, and the larger process into which they fit. While I'm 100 percent with the techniques themselves, I'm 100 percent against the process that he described for using them. The techniques are optimized for being thoughtful in a cognitively difficult, complicated area where you're breaking new ground, and the thinking that's embedded in the practices is absolutely essential to doing effective software development.

The process, however, seems to be avoiding a problem that we've worked very hard to eliminate. The engineering practices of extreme programming are precisely there to eliminate that imbalance, to create an engineering team that can spin as fast as the interaction team. Cooperesque interaction design, I think, would be an outstanding tool to use inside the loop of development where the engineering was done according to the extreme programming practices. I can imagine the result of that would be far more powerful than setting up phases and hierarchy.

To me, the shining city on the hill is to create a process that uses XP engineering and the story writing out of interaction design. This could create something that's really far more effective than either of those two things in isolation.