Losing the wait: A "wait loss" program for software development lifecycles

Level: Introductory

Laura Rose, Quality Engineering Manager, IBM

15 Aug 2006

from The Rational Edge: This article describes a variety of ways that software development organizations, particularly testing teams, can work more efficiently. Personal fitness and weight loss principles are used as a metaphor to illustrate these concepts throughout.

Like many people, many software development teams would benefit from lifestyle (or should I say "lifecycle") changes to help make them more lean, fit, and agile. Using metaphors from the realm of personal fitness and weight loss, this article explores concrete, effective ways to achieve those goals.

The inspiration for exploring the parallels between weight loss and "wait loss" in our software development projects began with my own personal fitness program. Like many software applications, my personal project will be in maintenance for quite some time. But I'm happy to say the initial release was successful -- I feel both fit and fabulous.

Meanwhile, I recently came across an article in the IEEE Computer Society publication IEEE Software*, entitled "Are You Fit?,"*[1] *a review of the book* Fit for Software Development: Framework for Integrated Tests *by Rick Mugridge and Ward Cunningham, in which the fitness of Olympic athletes is compared to the world of software development. I'd already noted that various software quality conferences publish "project health reports" as a way to track the "wellness" of our software projects. In fact, it seems there's recently been a spate of articles, project management dashboards, and the like that apply health and wellness imagery to software development activities.*

So are these associations between physical fitness and software development valid and useful? Let's see for ourselves...

Basic assumptions

Like people, every project is different: different team members, different skills, and different goals. Just as some individuals can eat whatever they want and never gain an ounce, others who eat the very same food have a very different result. In short, what works for one project may not work for the next project. But no matter what type of project you're talking about, certain principles hold true. This also applies to our physical bodies.

In particular, no matter what body type you have, the key to weight loss is accelerating your metabolism. Metabolism for our purposes refers to the speed at which your body burns the food that you eat. Someone with a fast metabolism has no trouble staying thin. Someone with a slow metabolic rate can eat very little and still gain weight. Software organizations work much the same way. A software development project's metabolic rate is the speed that the organization executes on the committed features and tasks. Wait gain (as opposed to weight gain) takes the form of an increased backlog of defects, late feature deliveries, missed deadlines, and postponed or removed requirements (i.e., increased wait time for value-added assets). The faster our organizational metabolic rate, the more features we can deliver and the greater our capacity to respond to an increased activity level without creating an inventory of defects, delays, and bottlenecks.

Creating a lean and fit organization

To create a lean and fit software organization, many companies turn to the "reductions in force" method. This is much like our fad "starvation diets" -- and with similar results. These approaches might appear to work in the short term, but they have undesirable consequences downstream.

When we go on a starvation diet to reduce our calorie intake, we not only destroy muscle tissue but also actually slow our metabolic rate. Because we now burn food more slowly, we gain weight more easily when we begin to eat normally again, which is the exact opposite of our original intention.

Similarly, when our company reduces its workforce, we slow our organizational metabolic rate (i.e., our capacity to do work). This forces our work body to gain wait in the form of a large inventory of defects, undelivered features, missed deadlines, and bottlenecks. The organizational equivalent of a starvation diet is a downward spiral that destroys a team's muscle and ability to execute. Although the goal in reducing the workforce was to reduce costs and increase net worth, the team may now be unable to deliver the product that would actually increase revenue. In that case, we're not accomplishing our ultimate goal, which is "to make money."

So how can an organization reprogram its metabolism so it can commit to more and simultaneously reduce wait gain quickly, while delivering high-quality software products?

Reprogramming organizational metabolism

If you want to lose weight, ironically you have to eat. The key is to eat smaller meals more often, which speeds up your metabolism. (What you eat is important, too, of course.) Skipping a meal, or eating too much food at any one meal, will tend to slow down your metabolism. This basic approach also works when it comes to applying the iterative development lifecycle approach in a software organization.

In his book *The 6-Week Body Makeover*,[2] nutritionist Michael Thurmond describes metabolism as "a bonfire" and the food you eat as logs you put on the fire. If the bonfire is burning big and hot, it has no trouble consuming the additional wood that you put on it. Conversely, if the fire starts to die out and you put a large log on the smoldering embers, the log just sits. Better to feed the embers first with kindling and small branches until the fire is roaring again.

A software project works the same way. If your delivery rate is already very slow, you can't expect the team to deliver the big features. The work just sits there and eventually gets stored as wait (backlog). But if you continue a cycle of feature deliveries in small increments, you will continuously deliver features in an ongoing fashion -- which is the foundation of the iterative development method.

The right diet for a software project

To lose weight successfully, it's useful to understand the basic chemistry of your food. The types of food most beneficial for weight loss purposes are fresh, whole, unprocessed foods. Processed foods aren't an efficient way to eat or be satisfied, because the unnatural chemicals, salt, and preservatives they contain alter the way our bodies function. A body running on unclean, processed food doesn't run as efficiently.

Software projects work pretty much the same way. It's better for the project to consume fresh, natural, clean, simple "bug free" code.[3] But in today's world of service-oriented architectures, open source, and outsourcing, we're often dependent on pre-packaged code, applications, or resources. Therefore, we can't always prepare our own software meals to avoid the equivalent of added chemicals and preservatives; i.e., bugs. But don't pre-packaged components, like fast food, save us time? Like a busy lifestyle that is frequently fueled by fast-food burgers and fries, the rewards of using pre-packaged code might be short term, because we're not immediately aware of the downstream effects of these choices. Some pertinent considerations:

- By consuming pre-packaged components, we've influenced the testability of our own solutions.
- Pre-packaged code compromises our ability to maintain and upgrade our own product.
- Pre-packaged code inherently limits the product's quality standard to the lowest common denominator; i.e., the weakest link may reside in code that is outside our control.

So how can we create a lean, quality-focused software development environment while working with pre-packaged code? As mentioned above, different organizations respond to these types of inputs differently. Organizations that don't need to integrate tightly with third-party components may have no problem absorbing and tolerating them. For teams that do need to work closely with pre-packaged modules, here are some techniques to increase success:

- Understand the ingredients. Just like reading food labels for sugar, fat, and sodium content, educate yourself on the code that you're about to consume. If the packaged product doesn't already come with acceptance criteria labels, define your acceptance criteria based upon static analysis results, code inspections, defect reports, and other criteria that conform to a known quality standard.
- Ask for "lite" versions, stripped of features your project doesn't need. Just as fried foods or foods cooked in oils and sauces slow down your body's metabolism, complex components that contain functions or features that you don't need can slow down your production rate and may even reverse it. Don't assume that you have to accept the entire package.

Just like many restaurants are willing to accommodate special orders, ask to have your version prepared with your smart choices.

- Use agreed-upon acceptance tests to increase your confidence prior to hand-off. If possible, automate the acceptance tests that specifically check for potential problems you've anticipated. For instance, while it's common to create tests to make sure your product works "as expected," also include tests that catch behaviors in the external component that you know will cause problems for other modules. Continually run those to flag any unexpected changes that will affect team productivity or the product's maintainability.
- Recognize that it may take extra time to adopt changes to external components -- particularly if they are made frequently -- and modify your project schedule accordingly.
- Create stubs or temporary (substitute) routines to work around a dependent-code area. This is analogous to packing a healthy snack in the car as a backup, to ensure you have something appropriate to consume. If the external component is delivered late or does not work as expected, you have an acceptable backup to support testing. By reducing your dependency, you make it easier to maneuver past third-party problems.
- Bring your own code to the open source library. This is like bringing a dish of your own to a pot luck party, in order to make sure you have something appropriate to enjoy.
- Create clear, precise, and unambiguous "contracts" to improve communications with both internal and outsourced teams -- and work according to them. This creates a transparent and predictable environment that enables constant feedback.

Fat-burning exercises for software projects

To accelerate physical fat loss into a higher gear, we can incorporate an exercise program as well as healthy eating. So, what is a proper exercise routine for a software development lifecycle?

Although every organization is different, there are some typical problem areas and activities that generate the most waste. Waste in terms of a software development project generally falls into three categories: (1) wait time, (2) rework, and (3) defect generation.

In the sections below, we'll customize a metaphorical "exercise program" for software projects to help tone and strengthen these typical problem areas by working with their causes.

Reducing wait gain from decision delays

One of the largest time-wasters and wait-gain areas on software projects is decision delays. Often the reason for a delayed decision is the fear of making the wrong one. But, ironically, choosing not to make a decision is itself a decision to delay action, which doesn't bring you any closer to the right answer. Even the wrong decision brings you closer to a workable solution, because you can immediately deal with the consequences of a "made" decision. The quicker you make the decision, the quicker you can move forward and take action on both the positive and negative results.

I'm not suggesting reckless or snap decision-making. I recommend moving fast on the reversible choices and more slowly on non-reversible, high-risk items. If the projected consequences of the wrong choice are minor, we should make the best decision based on what we know now and move on. In our iterative and constantly changing technical environments, we cannot know with 100% certainty that any decision is correct, because it is implemented in the future. So make it and don't worry about it. Once the decision is made, stop discussing it. Execute, learn from any ensuing mistake, and just move on.

Of the items you cannot decide today, define the specific action items that will close the gap between where you are and where you need to be to make that decision. Make sure you have explicit owners and deadlines for each action. In my product group, we often attend meetings to exhaustively discuss a problem. After much struggle, we are dismissed... often because we need to attend another meeting concerning another issue. Therefore, not only have we delayed the decision, we haven't put in place a chain of events to get us any closer.

An effective exercise for reducing the wait time on decisions is to have a Recovery Protocol (see Figure 1).

Attributes	**Add (Most Flex)**	**Optimize (Less Flex)**	**Comprimises (Less Flex)**	**Accept (Least Flex)**
Resource	X			
Scope		X		
Schedule			X	
Quality Criteria				X

Figure 1: Sample Recovery Protocol

The Recovery Protocol is a good tool to use at the beginning of a project. A team can use it to agree on and specify what approach will be taken when a problem arises; for instance, what will we do if the project starts slipping? Figure 1 illustrates a Recovery Protocol that prioritizes several critical project attributes. This example indicates that the team has agreed -- at the outset -- that resources are the most flexible value in their value system. Therefore, if the project starts to slip, they will continue to add resources until unable to do so. This would include taking people from other projects (possibly delaying those projects) and/or hiring additional contractors, etc. If they have utilized all available resources and are still behind schedule, they will turn to reducing the feature list of the project while still optimizing value to the customer. The project team would continue in this manner until they were back on schedule. Only as they run out of features to eliminate will they slip their schedule. Product quality will be the last element compromised under this particular Recovery Protocol.

By putting a Recovery Protocol in place before any problems actually arise (and we're cool and not under pressure), decision delays are reduced or eliminated when problems occur because we've already agreed on how to proceed. This wait-burning exercise gives the organization the flexibility and stamina to respond to change in a consistent and value-driven way.

Of course, the above chart is merely an example. Teams may decide on different protocol attributes or a different presentation scheme. The objective is to decide how you plan to handle situations according to your agreed-upon values.

Dealing with an ineffective build process

Daily builds, like a daily exercise regimen, is typically accepted as a best practice. But doing either practice incorrectly often slows our progress and negatively affects our productivity. An ineffective build process not only adds time to market, but also negatively affects the morale of the entire project team. This extra wait is carried by everyone. By the same token, improving this area increases the agility of the entire software development team.

It's generally accepted that the frequency with which builds are generated, bound, tested, and released to the test team reflects on the maturity and quality of a development process. Automated test cases are thus continually added, bulk batched, and blindly executed daily because it means we're doing our jobs better.

For example, I've encountered numerous "health reports" that monitor and track pass/fail statistics for daily builds based upon these kinds of automated unit and component tests. But even though an ever-growing suite of automated test cases is run daily, the failures they turn up are rarely reviewed, corrected, and fixed with the same urgency. Since the fix rate of the unit test failures that drive the pass/fail rate of the builds aren't on the same daily cycle, the number of daily failed builds inevitably accumulates.

One line of thinking is that, since the build verification tests are all automated, what's the harm in running all of them every night -- it's just machine time, right? It's true that executing the tests doesn't take a lot of any person's time. But failure analysis does. By deciding to run all the tests, you've implicitly agreed to review each failed test case within one work day, in time for the next daily build run. For example, if a build suite of 8,000 automated test cases generates 600 failures, are you realistically able to handle this? If you're not reviewing the failures, the tests become meaningless.

Naturally, as software quality practitioners, we don't expect or want every build to pass. In the course of adopting outside components, for instance, certain tests will continue to fail until we've fully integrated the acquired package. However, when we mechanically overlook failed builds because we expect certain tests to fail, we might miss the fact that tests that should be passing aren't.

The solution is cut the excess from your build exercise routine so that you only run the tests that are supposed to pass -- and that you can commit to reviewing and resolving if they fail. This way when the build fails, it's truly failing. This approach is also more manageable, because it makes it possible for you to immediately analyze, fix, and verify those failed test cases in time for the next scheduled build.

Even when we're focused on fixing just those test cases that should be passing, it may take more than one day to diagnose, fix, and verify defects. Therefore, the daily builds will continually fail as expected until the appropriate test cases are passing. In this case, are those builds really failing? Aren't they performing exactly as expected (i.e., loosely translated, aren't they "passing")? Since these are expected occurrences, the daily pass/fail graphing (although popular) isn't a useful indication of the actual health of the product.

Here's an example: Tuesday's build contains only the test cases that are supposed to pass on this build. Tuesday's build fails because of defects associated with TestCaseA, TestCaseB, and TestCaseC. John is able to fix the TestCaseA- and TestCaseB-related defects on Wednesday. And he fixes TestCaseC on Thursday. Since Friday is the first time the build should actually pass, Wednesday's and Thursday's failed builds are not reported as build failures. Thus, in a typical daily build report, we would be reporting a 40% pass rate for that week. But if we only consider the builds that were expected to pass, we actually have an 80% pass rate (see Table 1).

Table 1: Builds expected and blocking counts

Build	Tests Pass	Results Match Expected	Test Days Blocked
Monday	1	1	0
Tuesday	0	0	1
Wednesday	0	1	1
Thursday	0	1	1
Friday	1	1	0
Health	40%	80%	20%

Therefore, a more accurate graph would be one that tracks the builds we anticipate will pass.

A better indicator of process maturity than the accumulated number of failed daily builds is how quickly one can recover from the failure, create a testable build, and move on to the next development phase. In the above, it took three days (Tuesday, Wednesday, Thursday), to "pass" the build. Testing on this project was delayed three days. Would not a causal analysis on how to reduce the fix-time rate be more beneficial to the team's overall efficiency, in terms of reducing wait time, than tallying how often a build fails?

Another way to make daily builds more useful for reducing wait time is to prioritize what failures to fix first. What's important isn't whether a build "passes" or "fails" -- it's what useful information we can gather from the build statistics to prioritize our activities. Yet many software development processes use the build status to blindly and artificially gate progress downstream. Artificially gating progress creates delays and wait. Even when we streamline our test cases to only those that "should pass," we can fall into this trap.

Let's replay the scenario just described to illustrate a wait reduction approach. We know the Tuesday build contains only the test cases that are expected to pass at that point. Tuesday's build fails because of defects associated with TestCaseA, TestCaseB, and TestCaseC. TestCaseA and TestcaseC are critical to the proper startup and installation of our product. Unless those test cases pass, the team is blocked from any significant testing. TestCaseB is also an important test case, but it lies downstream in the test cycle. Therefore, we won't execute TestCaseB for the first few days. Meanwhile, John focuses his efforts on TestCaseA and TestCaseC. John is able to fix TestCaseA and TestCaseC on Wednesday. We remove TestCaseB from the "expected pass testcase" list and rerun the build. The Wednesday build passes. Test teams are unblocked and moving forward. John fixes TestCaseB on Thursday, updates the "expected pass testcase" list, and that build passes. The test team receives an updated build with TestCaseB fix in time for that area testing, without slowing progress on their test schedule.

Table 2: Prioritization of test cases to reduce wait

Build	Tests Pass	Results Match Expected	Test Days Blocked
Monday	1	1	0
Tuesday	0	0	1
Wednesday	0	1	1
Thursday	0	1	0
Friday	1	1	0
Health	40%	80%	20%

So, even though the "health report" on the build rates is exactly the same in Table 1 and Table 2, the first example delayed the program three days. In the second instance, the team experienced only one day of wait time.

Therefore, we need a report that prioritizes failures and tells us about actual downtime, versus a potentially misleading list of daily build results. Health isn't really determined by the number of adjustments needed. The proper sign of an organization's health, agility, and fitness is how quickly the organization can recover. When recovery is smooth and immediate, the problem that required adjustment has little impact on the general outcome of the software development project. It's like an agile athlete balancing on one foot: They constantly adjust and realign themselves, but these adjustments are virtually unnoticeable in the context of the performance.

To sum up, we can improve build efficiency and effectiveness through what might be called, in fitness terminology, a precision sculpting strategy. Instead of running all your automated unit tests every day, only run the ones that need to pass for this particular delivery or iteration. To further optimize the effectiveness of your daily build verification strategy, prioritize every failure with a focus on minimizing wait time.

Toning flabby requirements

Most software projects can be considered at least partial failures, in the sense that few projects meet all their cost, schedule, quality, and requirements objectives. One primary cause of failures is incomplete, vague, or ambiguous requirements. Even the best requirement tools do little to help us verify the quality or completeness of the requirements.

Requirement inspections are an effective toning exercise to assure you get the product you really want while reducing time to market.

In his book *Software Testing*,[5] Ron Patton states that a well-considered product specification has eight important attributes. By critically reviewing your requirements with these attributes in mind, you'll jump-start your inspection process. According to Patton, our requirements ought to be:

- **Complete**. Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
- **Accurate**. Is the proposed solution correct? Does it properly define the goal? Are there any errors?
- **Precise, unambiguous, and clear**. Is the description exact and not vague? Is there only one interpretation? Is it easy to read and understand?
- **Consistent**. Is the description of the feature written so that it doesn't conflict with other items in the specification?
- **Relevant**. Is the statement necessary to the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?
- **Feasible**. Can the feature be implemented with the available staff, tools, and resources within the specified budget and schedule?
- **Code-free**. Does the specification stick with defining the product and not the underlying software design, architecture, and code?
- **Testable**. Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

Patton further suggests we comb our specs for "problem language," whose presence often spells trouble. Examples include:

- **Always, Every, All, None, Never**. If you see words such as these that denote something as certain and absolute, make sure that they are indeed, certain. Think of cases that violate them when reviewing the spec.
- **Certainly, Therefore, Clearly, Obviously, Ordinarily, Customarily, Most, Mostly**. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- **Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly**. There words are too vague. It's impossible to test a feature that operates "sometimes."
- **Etc., And So Forth, And So On, Such As**. Lists that finish with these words aren't testable. There should be no confusion as to how the series is generated and what appears next in the list.
- **Good, Fast, Cheap, Efficient, Small, Stable**. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- **Handled, Processed, Rejected, Skipped, Eliminated**. These terms can hide large amounts of functionality and need to be specified.
- **If... Then (but missing Else)**. Look for statements that have "if...then" clauses but don't have a matching "else." Ask yourself what will happen if the "if" doesn't happen.

How valuable is this exercise to creating a firm n' fit development process? Since we know that even back in 1996 it cost an average of $25 to fix one defect in the design phase versus $16,000 after product deployment (see Figure 2), keeping metrics on the number of important defects you find during your requirements inspection process can be directly converted to cost savings. Using these metrics illustrates your return on investment and revenue impact.

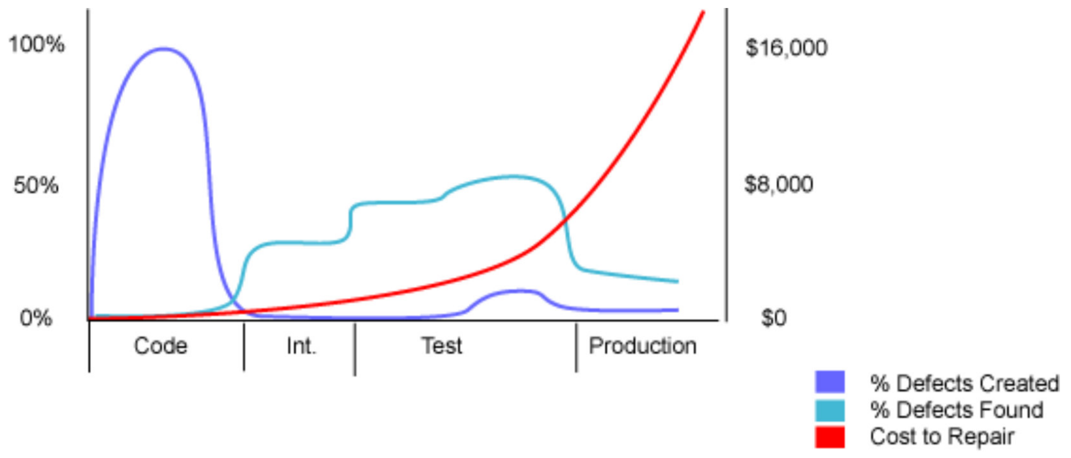


Figure 2: Cost to repair defects at different points in the software lifecycle

Relaxing the tension between teams

Another source of delays is miscommunication. This problem is prevalent whether you're dealing with outsourcing teams, geographically distributed teams, or internal teams. Miscommunication exists wherever there's a division of roles, goals, and labor. For the purposes of this example, we'll focus on the relationship between developers and testers, although the same techniques work in any relationship.

"Professional product relationships are like contracts."[6] In order to establish and maintain such contracts, we need to understand each other's roles. For instance, if you need something from me in order to do your job, you will want me to have a clear understanding of your expectations.

On many software development projects, the relationship between developers and testers (as well as other teams) is clouded by tension in the form of misunderstanding, disappointment, and frustration. The difference in the way developers and testers approach problems is the yin and yang of this relationship. Table 3 lists some stereotypical attitudes that developers and testers may hold.[7]

Table 3: Attitudes that contribute to developer/tester tension

Sample Issues -- Test	Sample Issues -- Development
The development team is always in a rush and telling us to go faster. Unlike them, we cannot afford to make mistakes.	They don't work as fast as we do.
They forget to tell us things that change. They make the mistakes and then they want us to work late to make up for it.	They don't work as hard as we do.
It's hard to get accurate information. We ask questions and often hear nothing back.	There are so many questions; I cannot get my own work done anymore.
They are always adding things to the project even at the end. They are so disorganized.	They don't always tell the truth about what is going on. They tend to exaggerate the symptoms.
This software looks like it was never tested by the developer.	Why do I need to test my code? We have an entire department to do that!

Developer/tester tension exists not only during the project, but also after the software application is deployed. One underlying issue is that most project members don't have a clear grasp of the tester's role in their project. Developers, product managers, project managers, and even the testers have unrealistically high expectations about the level of coverage that can be achieved, the types of problems that can be routinely uncovered, and the time it takes to consistently duplicate and diagnose the symptom. Ironically, organizations that depend on separate test organizations to promote or highlight quality issues often accomplish the opposite effect. Separatism de-emphasizes the developer's responsibility for software quality. Since software quality doesn't start with tests, asking testers to shoulder the full burden of ensuring quality is unrealistic.

As testers, we can commit to two important strengthening exercises that can help mitigate tension with developers and other team members. First, we can create a clear, precise, and unambiguous "contract" between the test team and the rest of the organization. Second, we can fulfill the testing end of that contract.

Enforcing the success criteria for the project and clearly describing the various roles, the expectations, and how each member intends to achieve the goals is essential to creating a workable professional relationship contract -- and to reducing wait caused by unnecessary tension.

Body sculpting to build strong teams

So let's say you've implemented some of the above strategies and your projects are starting to lose wait. Your organization's metabolism has noticeably increased and your teams have renewed energies. But how can you maintain this progress on a continuous basis?

This is the time to sculpt your organization to create the exact "body" that you want. Precision body sculpting refers to the ability to shape and define each and every area of the body so that it looks precisely the way you want it to look. Muscles burn fat, and the people in your organization are its muscles. Therefore, toning and building the right team members in the right proportions will keep your organization running lean and smooth. So how would we reshape our software development organization in this way?

The first step to reshaping any body is to design a Body Blueprint or individual development plan. Following is a general approach with specific examples for a software development organization.

The general steps in a Body Blueprint are:

1. Do a self-assessment on where you (or your employees or your organization) are today.
2. Identify where you want to be.
3. Highlight problem areas or gaps between the "here" and "there."
4. Start a precision sculpting routine for targeted, high-profile areas first.
5. Outline short-term goals to make sure you reach your destination on the timetable you desire.
6. Create rewards to go with the short-term goals.
7. Once significant changes have occurred, re-evaluate and redesign your Body Blueprint for the next level of improvement.
8. Repeat...[8]

These steps might sound simple. But in applying them, you have not only customized a program to your individual needs (whether associated with your physique, product, or organization), but have also taken ownership of the process, execution, and results. Taking these steps focuses your energy and intention and will catalyze significant change.

In a software development organization, you can use many things to assess and define your product. One well-known method is the entrance and exit criteria of each iteration or milestone in your development lifecycle. Many software projects do Steps 1 and 2 above as a matter of course. They identify their goals and track their progress against those goals. We even highlight the gaps between the two in weekly and executive summaries.

What we don't execute well is Step 4 and beyond. Development organizations are notoriously optimistic. If we feel we are losing steam or falling behind, we're confident that we can catch up. No additional training or skill assessment is done to assess our actual capability or capacity to effectively achieve the assignment. The only course change that is required is to "work harder." Unfortunately, this habitual thinking assumes that we weren't already giving 100% of ourselves when we started to slip. It also assumes that we were capable of accomplishing the goal in the first place.

Another common mistake is our failure to develop a clear picture of where we should be today. We know we're supposed to be done on Day X, and maybe we think we know where we stand today. But if we don't have short-term goals and milestones already plotted along the way, we're not really sure if we're behind or ahead of schedule.

Consider this hypothetical exchange:

Joe: "The team is falling behind in our deliveries. What are your plans for getting back on schedule?"

Mark: "We've realized that we don't have anyone on staff that understands the domain well enough. We're currently interviewing a few contractors to help out in the short term."

Joe: "For how long and at what cost?"

Mark: "Final delivery is scheduled in three months, so the contracts will last three months. We'll need three additional people to complete the task within that time."

Joe: "Any additional costs? Training? Equipment needs?"

Mark: "Well... we plan to assign a developer to each contractor as a buddy/mentor to reduce the learning curve on our product and coding standards. We do need to find office space and equipment for each contractor. We're also considering remote locations."

Joe: "So, in summary, you're spending time interviewing and hiring three people, need to locate space and equipment, will be interrupting/slowing at least one developer to mentor/train the new people, and your team will have to absorb increased administration and collaboration functions associated with a larger team. If the contractors work remotely, there will be additional mentoring, administration, and management complications. So we're adding a half-dozen tasks within the same timeframe in which we couldn't accomplish the original number of tasks. How will this work?"

Mark: "We'll just work harder. We'll get it done."

An exaggeration? Maybe, but it illustrates that when we're not looking downstream in regards to our skills and capability to deliver, it will eventually catch up to us.

Reasons for those occasional slips

While producing an application with zero defects and zero stop-production issues is an admirable goal that we should continue to aspire to, it's more realistic to acknowledge that we will encounter slips and stumbles from time to time. As in our personal weight loss programs, our software development projects can slip up due to reasons like those discussed below.

Knowing these slips will occur from time to time, and preparing for them, allows us to respond with control instead of reacting without thinking. Let's take a look at some proper ways to respond to some common slip-ups.

Misplaced attention

From a software development point of view, our slip-ups are reflected in the number of generated defects. In the physical realm, "A sure indication of whether or not you are in the here and now -- is the number of bruises and scrapes on your extremities. When you do not pay attention, you bump into tables, chairs, doorways, you name it."[9]

Agility, flexibility, and balance all help us avoid bumping into things. But if we're not paying attention, not showing up in the moment, not being mindful, we're going to bump into things no matter how fit we are. Not paying attention to the here and now just means we're placing our attention somewhere else. So how does this translate to the software organization?

The collective goal of the entire development team is to produce something that customers will value. Getting truly engaged in creating quality outputs for each iteration and phase is critical. This means that during the requirements review activities, for example, the developer's primary task is "requirements review." During the design activities, the developer's primary task is creating and reviewing the design documents. During the coding activities, the developer's primary task is generating bug-free and customer-relevant code. During the documentation review activities, the developer's primary task is making sure the user assistance materials and error messages serve to flatten the customer's learning curve. During installation and setup, the developer's primary task is to make sure customers can set up and configure your product easily so that they can get their "real" job done as efficiently as possible.

If you're writing e-mails or coding during a requirements review meeting or teleconference walk-through, your attention is misplaced. If you start a manual test, then walk off to talk to someone or to finish another task, you're not being mindful of any unexpected results or timing problems that test uncovers. Even if you're consumed with a deadline to submit your code on a particular date, you're distracted by the schedule and are not fully engaged in the actual pair programming and testing that's going on *now*.

So what's available to help us combat distraction and stay task-centered? We can "stay in the moment" by setting smaller, short-term goals and rewarding ourselves when we meet them. Paying attention to others is another way to stay connected. We can do this by rewarding and thanking others when they've helped us meet our team's short-term goals. Focusing on the team goals of passing an iteration's milestone versus isolating on your to-do items for the long-term deliverables also helps.

Being mindful of the moment (paying attention) requires more than being present. It's something we have to actively and continually pursue. It requires being available to be transformed by the event. A good test to see if we actually did pay attention is that, if we feel exactly the same coming out of an event as going in, we didn't truly and fully engage in that activity.

Friendly and self sabotage

As I touched on above, many test teams are justified in complaining about their inability to continue their work due to build failures. As testers, we feel that we've done our job by highlighting the problem -- now it's someone else's job to correct it. While we're "putting on wait," we stay busy, perhaps by working in a different area like a nice-to-have feature that isn't scheduled for this iteration. Unfortunately, none of our "nice-to-have" features do anything to reduce the current wait-time. Everyone is working hard. But the project is falling further and further behind. Therefore, we are misusing time.

If we were fully participating in the moment and sharing in solving the problem, we probably could have found a way to assist within our talents and skills set. Time could have been better spent assisting the build team to investigate the errors, or streamlining the build verification test cases, or prioritizing the minimum fixes required to make forward progress, or helping test some of the release-defining components of this iteration. Although we were staying busy, we weren't actually participating in the issue at hand. We weren't assisting in any way to reduce the wait.

This is so-called friendly sabotage. Situations like the one just described allow us to feel we have more time without being the cause of the delay. As long as "they" are stuck, "we" can work on the things that we want to work on.

Self-sabotage looks pretty much the same, except "we" are the build team. We don't ask or allow any other team to assist. We don't experiment or investigate alternatives to how we've done it in the past.

The trap in both instances is the "my job/your job" mind-set. Everyone has different skills and talents; true enough. Everyone has their primary duties and tasks; very true. But none of this matters if the product never ships. Paying attention to the real goal -- delivering the agreed-upon, value-added release at the scheduled iteration -- helps us avoid these stumbles.

Social obligations

In a personal weight loss scenario, social obligations are social functions, parties, holidays, places where the center of the entertainment is food. On the software development scene, social obligations occur when you don't have full control over your development environments. The following are good exercises to increase your control when you feel you have none.

- Focus your energy on the areas you do control.
- Outline your overall goals, vision, and success criteria.
- Leverage standards and acceptance criteria/expectations that support those goals, visions, and success criteria.
- Get agreement (similar to a professional contract) on those expectations, roles, and timetables.
- Execute your side and assist others to execute theirs.

This approach increases your influence over your work environment. Just like the bathroom scale, these controls help you stay focused on your goals and track your success.

On the other hand, once you've put those controls in place, don't blindly be a slave to them. Remember the controls are in place to support a larger vision (your principles and values). Like the scale, the controls are only a measuring tool. The plan is about losing wait, and building a fit and agile development organization. We should be much more interested in decreasing our time to market while increasing customer satisfaction. Therefore, don't allow the controls to create wait gain.

For example, say one of your overall goals is to deliver a product with zero known defects. If you're meeting the customer's expectations, the workarounds for those defects are acceptable, and your self-correcting utilities are allowing customers to get their work done, then the fact that you have a handful of defects need not delay the release (even though your release criteria state otherwise).

We know that we often turn to comfort food (chocolate, sweets, etc.) when we encounter stress or depression in social situations. Even on a software project, during times of stress and panic, we have a tendency to turn to our comfort zones. Unfortunately, sometimes that means we fall back to our bad habits. By using a Recovery Protocol, we can stay in synch with our overall vision, values, and goals during the rough times.

As with a fitness program, celebrating your short-term successes will calm fears and reinforce your new habits. Reviewing the team goals will avoid any negative sabotage temptations during those hard times. Placing contracts with acceptance and success criteria will boost your influence on external components. By applying these ideas to your particular situation, and sharing them with others, you can increase not only your own efficiency, but also that of the team or organization overall.

Conclusion

Just like the pounds that gradually accumulate over the years, the wait gains in our software development cycle often go largely unnoticed. Missed deadlines, postponed features, or a rising backlog of defects become accepted by-products of our development lifecycles -- business as usual. But believing that "that's the way things are done" is merely a habit that we can change.

Throughout this article, I've talked about the wait gain in our development lifecycles in terms of backlog of defects, undelivered features, time required to deliver a fix, the number of cycles it takes to reach your quality criteria, and so on. I also talked about techniques to reduce the wait time and how to stay focused on overall goals and vision. Perhaps most importantly, I suggested that we redefine the chief criterion by which we judge the health of our organizations not by the actual number of slips we make, but by how quickly and effortlessly we recover from them.

Including a "wait loss" plan in your project strategy will increase your software organization's flexibility, agility, and productivity. Armed with a new awareness of the issue of "wait gain and loss," I hope you can identify some wait loss opportunities on your own project.

Notes

[1] Anthony Akins, "Are You Fit?" *IEEE Software*, May/June 2006, Vol. 23, No. 3: 100-102.

[2] Michael Thurmond, *6-Day Body Makeover: Drop One Whole Dress or Pant Size in Just 6 Days -- and Keep It Off*. Warner Books, New York, 2005.

[3] Tod Golding, "Fighting Temptation." *Better Software*, June 2006.

[4] If your team agrees up-front to delay the next test phase until your build acceptance tests (BATs) pass, and the BATs include tests that "aren't expected to pass," then you have artificially delayed your progress.

[5] Ron Patton, *Software Testing*. Sams Publishing, San Jose, CA, 2006.

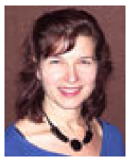
[6] Nathan Petschenik, *System Testing with an Attitude: An Approach That Nurtures Front-Loaded Software Quality*. Dorset House Publishing, New York, 2005.

[7] These sample issues reflect my synthesis of ideas from *System Testing* as well as Michael Hackett's "Thorough Training Breeds Success in Global Test Teams" in *Software Test & Performance Magazine*, May 2006.

[8] Stephen Covey, *The 7 Habits of Highly Effective People*. Simon and Schuster, New York, 1989. I liken the Body Blueprint concept to Covey's "Sharpening the Saw" and continuous renewal through the "learn, commit, and do" upward spiral.

[9] Pamela Meyer, *Quantum Creativity: Nine Principles to Transform the Way You Work*. Contemporary Books, Chicago, IL, 1997.

About the author



Laura Rose is a quality assurance expert and the product manager responsible for automated performance test tools at IBM Rational. In addition to leading projects in both software programming and testing environments, she has thirteen years of programming experience and ten in test management. She has been a member of the American Society for Quality, the Triangle Quality Council, and the Triangle Information Systems Quality Association. She is published and has presented at various test and quality conferences including IBM Test Symposium West, Practical Software Quality Conference (East and West), the American Quality Society Conference, Better Software Conference & EXPO and StarWest. You can reach her at llrose@us.ibm.com.

Share this....

Digg this story del.icio.us Slashdot it!