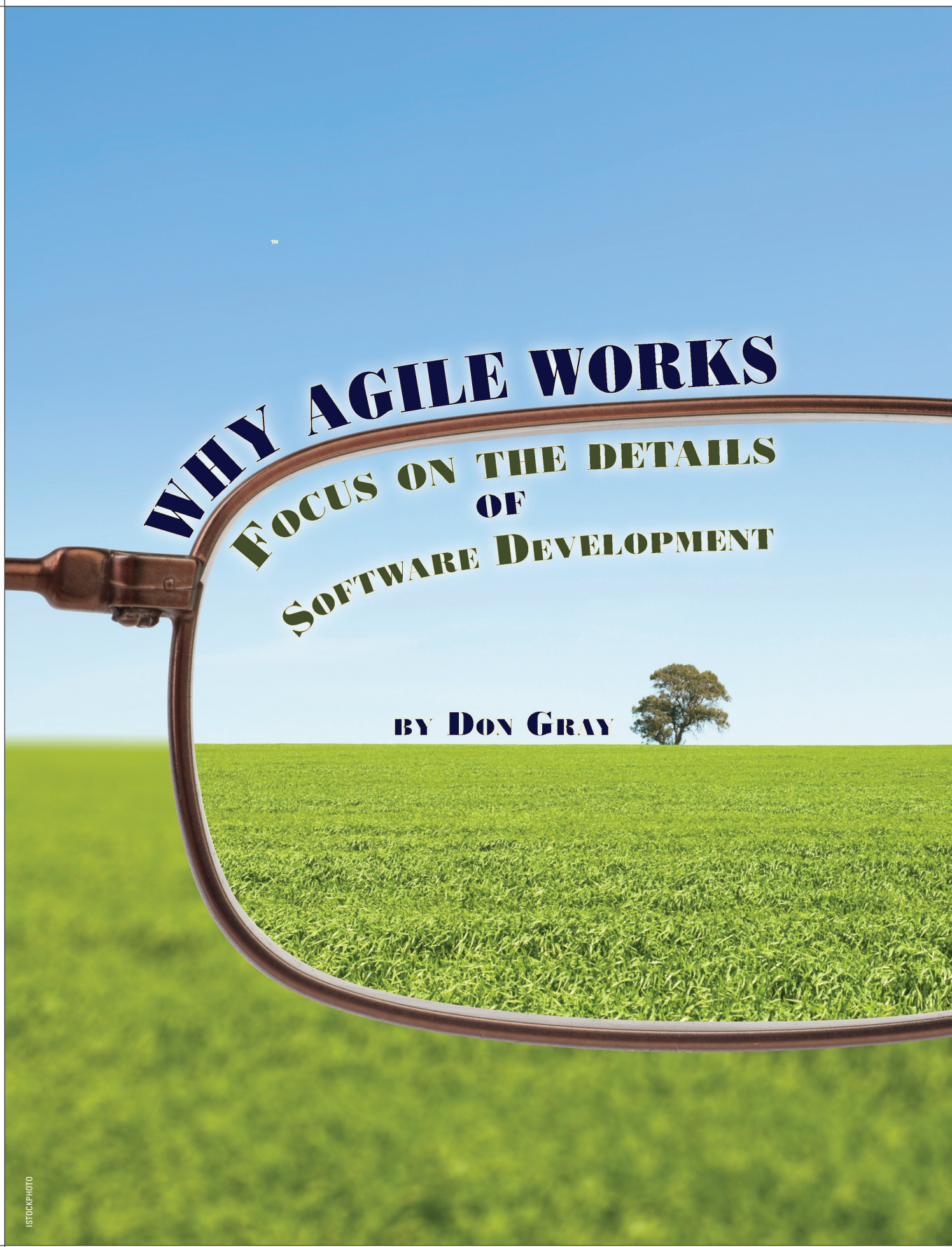


Why Agile Works

Focus on the Details of Software Development

By Don Gray

ISTOCKPHOTO

Someone once asked me, "Don, what does even the wisest person overlook?" The answer was "his nose." You can see your nose if you focus, but you generally don't bother. You assume it's there. Likewise, as I have gained experience in software development, I have learned that explicitly stating my assumptions keeps me from overlooking important aspects of what I'm working on.

The Sliding Scale of Software Development Complexity

Software systems run the gamut, from closed systems with low dimensions and linear behavior to open systems with high dimensions and non-linear behavior (see figure 1). Closed systems are unchanging, while open systems change based on interactions with their environments. A system with low dimensions has few interactions between its parts, while a system with high dimensions has many interactions. In a system with linear behavior, the output is the sum of the parts, while in a system with non-linear behavior, the output is a product of the system's interactions. Our assumptions about where software development as a system lies on this continuum determine the processes we choose to use for a project.

Early in my career, I worked on automating the production area in a new plant, "Project S." We had about 10,000 input/output points and twenty-five operator terminals scattered through five production areas. This project exhibited detailed complexity with a lot of parts to keep track of, but the project's physical nature limited the dimensions, established a linear process, and kept the scope from expanding.

I assume that some non-physical software projects, such as accounting software, have occurred often enough that they, too, fall in the "Close To" corner. But, how valuable can another accounting package be? That market seems to have enough. If we want to create value, we must leave the "Close To" corner and start sliding toward the "Far From" corner. We might start using unfamiliar technology (have you noticed the recent explosion in languages and frameworks?) and not be completely sure of what needs to be done.

Notice, in figure 2, as we move toward "Far From," that a project's nature shifts from simple through complicated, complex, and finally becomes chaotic. In each area, the methods and processes for delivering the project change. Fortunately, methods and processes used for complex projects can also be used for complicated and simple projects. Unfortunately, the single greatest problem I see when working with software development organizations is that they apply methods and processes for simple projects to complex projects.

What Is a Simple Project?

The primary characteristic of a simple project is the ability to analyze the project. Analysis means breaking a larger project into smaller parts to understand it. If we can understand the smaller parts and successfully aggregate them into

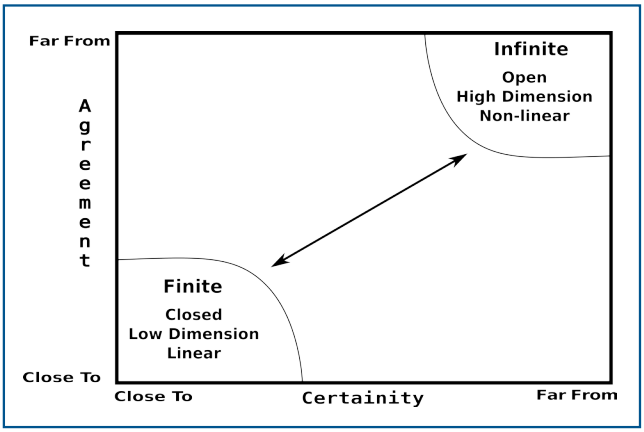


Figure 1: The sliding scale of software development complexity

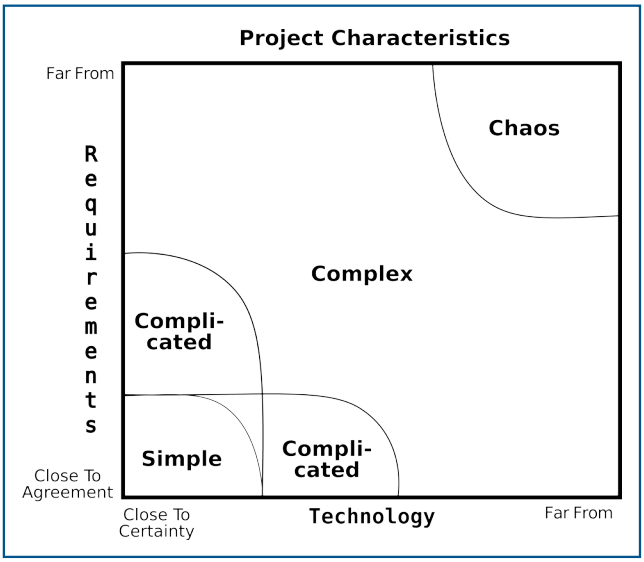


Figure 2: Project types in Stacey matrix format

"To deliver complex projects, we need a clear goal, reduced delay, feedback, aligned effort, and continuous improvement."

the larger project, then we have a simple project. This reductionist thinking started with the Greeks' trying to understand their world. They hypothesized atoms, which weren't verified until millennia later.

This thinking leads project managers to ask for a project description. Then, they (or a systems or business analyst) subdivide the project into smaller parts until they can generate a Gantt chart showing modules or tasks, dependencies, and an eventual delivery date.

Many project tools exist (perhaps most famously Microsoft Project) that deal with this detailed complexity. Properly created and constantly updated, these tools can project a delivery date. But, what assumptions do project managers make while creating these schedules? In my experience, common assumptions include:

1. What the customer asks for won't change (i.e., fixed scope).
2. The people I expect to work on the project *will* work on the project.
3. The task breakdown is accurate.
4. The time estimates for tasks are close enough.

Experienced project managers will include tolerances to allow for some variation because these assumptions will exhibit some variation. Nonetheless, the major underlying assumption remains: The project's nature fits the definition of a simple project (closed, low dimension, and linear behavior).

Not All Projects Are Created Equal

Several years after working on Project S, in a hot August in Chattanooga, Tennessee, I stood quietly in a pool of sweat as the project manager berated us for being behind schedule on "Project C." The chemical plant we were automating had a fast-track schedule. Pipe fitters made piping runs before the elevation drawings were complete and reviewed. Piping and instrumentation diagrams updated daily. We constantly rewrote the control software and operator displays based on the information *du jour*.

After three months, the plant owners stopped the project and sent everyone home. A week later, they invited us back to finish the project. We developed the software in the operator control room, not the engineering office. This gave us instant feedback from the users on the displays. The plant engineering VP's office was two doors away. Every morning, he updated us on necessary control system changes.

I don't know what happened to the yelling project manager. We never saw him again.

In some ways, this project was simpler than Project S. The plant was much smaller, with fewer input/output points and operator terminals. So, what created the problems?

1. Project details changed almost daily.
2. Everyone—engineers, developers, electricians, and pipe-fitters—constantly interacted to deal with the newest information.

Due to these dynamics, this project exhibited the characteristics of a complex project. The original engineering and project management company had demonstrated experience with simple projects, but their simple project methods and tools couldn't deal with this project's dynamic complexity. The product of the constant change, the project's interactions, and time constraint overwhelmed the plan.

Working with Complex Projects

Complex projects exist in the region where we need different tools and methods to solve the problems. Decision making shifts from technically rational thinking to brainstorming, dialectical inquiry, agenda building, intuition, searching for error, and occasionally muddling through. Rather than a process of "sense, categorize, and respond," the domain requires "probe, sense, and respond." We try something, evaluate the situation, respond appropriately, and the solution emerges.

Solving complex systems using reduction shifts the goal from providing customer value to building pieces of the overall solution. This leads developers to focus on technical tasks, not delivering small pieces of user value. The hope is that when all the parts get assembled into the system, it will exhibit the characteristics we're looking for. This works for simple projects. But, more often than not, it doesn't work, indicating that the project has a complex nature and we have used the wrong method to work it.

To deliver complex projects, we need a clear goal, reduced delay, feedback, aligned effort, and continuous improvement.

Clear Goals

> *Agile Manifesto Principle: "Business people and developers must work together daily throughout the project."* [1]

In my article, "Goal, Goal, Who's Got the Goal?" [2] I shared the three goals every project needs: the project goal, sprint goal, and daily goal.

The project goal needs to align with your company's strategic goals. It should focus on customer value. People don't purchase software because they want another program but because they will benefit from using the software. This provides them the value that convinces them to spend the money.

The sprint goal defines what the development team will deliver in the next sprint or iteration. While the team works to deliver customer value, it does so by completing technical work. The sprint goal keeps the team focused on the immediate work.

The daily goal keeps the team synchronized. It highlights completed work (I've worked with a team that applauds every task completed) and allows team members to share information and ask for help removing impediments.

Having clear and aligned goals creates a point attractor. This attractor allows the complex solution to develop over time. It creates a focus that pulls the work toward the final solution, one piece of user value at a time.

During Project C, sitting in the control room allowed us to ask the plant operators for ideas on what they needed to see and be able to do with the control software. Having the VP of engineering down the hall enabled us to deliver value with minimal delay.

Reduce Delay

Agile Manifesto Principle: "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

Delays in software development come in many forms. In large, siloed companies, finding the person with the needed information and skills isn't always easy.

I worked with one client moving from silos to Scrum who routinely took more than a day to identify the silo needed to solve a problem and the correct person to do the work. Most people in that company were also assigned to multiple teams. One person was assigned to six different teams. Another was so specialized that it took more than three weeks before that person could even look at the problem.

While this might appear efficient from a "resource utilization" viewpoint, it wreaks havoc with delivering user value. In one class I taught, 75 percent of attendees had been on a project that lasted more than a year, and one person had been on a project that went more than two years. Adding offshore development teams compounds this delay. Almost every sprint in this company had at least one task that slipped by each day as we worked with the offshore team members to clarify and answer questions.

Many people think of project work as the actual effort to write the code. Using project management tools to subdivide the work, assign dependencies, and calculate the critical path, we plot when the project will complete. This hides the reality that waiting for everything delays realizing income (that "ROI" thing managers love so much) until the end of the project. Lengthening the time until the end date also puts the project in jeopardy, due to organization priorities, changing requirements, shifting markets, and, in some cases, new legal requirements.

By focusing on incrementally delivering value, managers shift their thinking from "How can I make sure everyone is always working?" to "What causes delay in our process, and how can we restructure to remove those delays?" One client I'm working with has started "on-shoring"—hiring locally so team members can be collocated. When teams learn to think in small units of value, the ability to continuously deliver value increases. Not every project can ship at the end of every sprint, but it's cool to watch the results when it does happen.

By reducing delays, we can get better and faster feedback.

Feedback

Agile Manifesto Principle: "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

Project S and Project C employed different development structures. For Project S, someone took all they tasks he knew about and created a Gantt chart. Adding the task times on the critical path allowed the managers to compute an expected completion date. We followed the (at the time) standard process of analysis, design, coding, testing, and implementation. Managers monitored progress by comparing a task's scheduled time and actual time. As the schedule slipped, another developer would be added to the team to help make up the difference. Somehow, this never really helped, and the project's end date didn't change. This meant that the final phase,

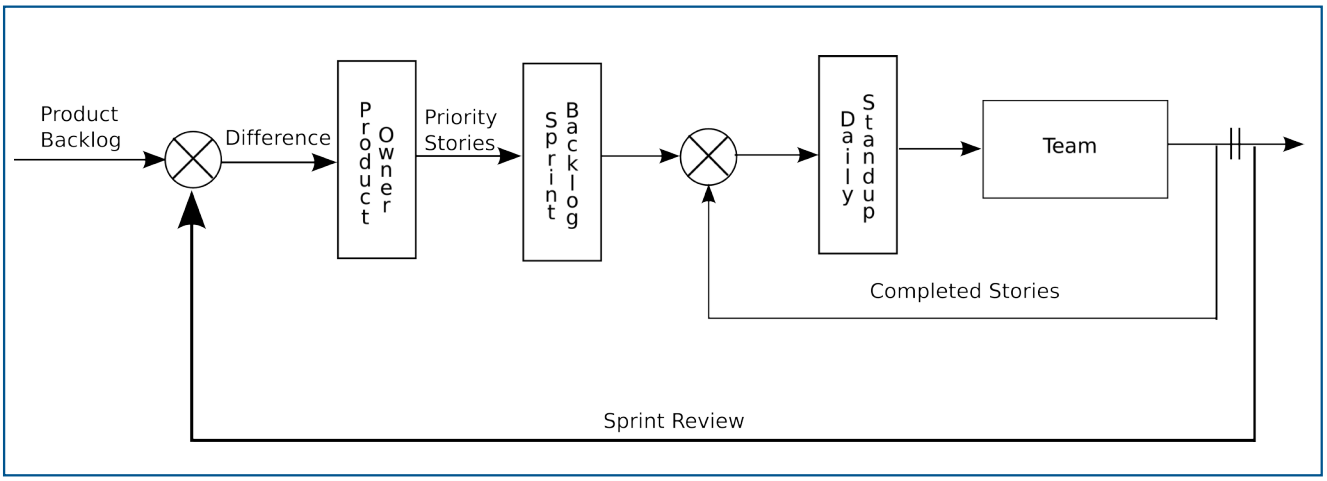


Figure 3: Two feedback loops

testing, was shortened, enabling us to meet the startup date. That didn't work well either.

Project C, on the other hand, used a development process more like figure 3. Two feedback loops tracked the project's progress. The inner loop—daily standup, team, and completed stories—allowed us to monitor our daily progress and answered the question "How are we doing right now?" The outer loop (which included everything) answered the question "How are we progressing compared to all the project work?"

The feedback loops inherent in this development method demonstrated to the business (stakeholders and product owners) how the software looked and operated. If needed, corrections in the software could be made while the software was still fresh in the developer's mind and prior to adding additional functionality.

Combining this method to measure progress and reducing delay by collocating with the users demonstrated actual progress. We didn't have project phases like in Project S.

> *Agile Manifesto Principle: "Working software is the primary measure of progress."*

During Project S, our status meetings included "percent complete" updates for the various tasks. Eventually, I came to understand Tom Cargill's Ninety-ninety Law:

> The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time. [3]

Dividing software development into phases and functional layers inhibits information flow, hides delays, and defers complex and difficult tasks until it's too late to deliver on schedule. Having clear goals, focusing on business value, reducing delays, and providing feedback on working software allow team members to align their efforts to complete the project.

Aligning Effort

> *Agile Manifesto Principle: "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."*

As I worked my way through college, my employer purchased a Data General business computer. It came with a program that allowed us to define what we wanted. The program then churned for hours, creating a business system. It never worked quite right—probably because we didn't know exactly what we wanted or how to use the program's output.

What have I learned since then?

People—users, developers, and testers—work together to create software from needs and ideas. People vary, tend to inconsistency, exhibit good citizenship, and are good at looking around. [4] In other words, people also fall into the complex space. To align effort, teams need a *compelling work goal*, five to nine members, stable membership, shared history, and interdependent work. [5]

The compelling work goal changes our jobs into careers. In *Drive*, Daniel Pink calls this "purpose." Having clear goals for the project helps the team connect their effort to results, and this helps create the compelling work goal.

Continuous Improvement

> *Agile Manifesto Principle: "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."*

If people are complex systems, then interdependent teams must be complex or even chaotic systems. When teams exhibit the characteristics of chaotic systems (open, non-linear, high dimensions), they have a difficult time accomplishing work. They spend time and energy making and remaking decisions, working on tasks that don't contribute to product, and dealing with interpersonal issues.

Periodically reflecting on becoming more effective focuses the team on how best to meet their goals. They may change how they develop software ("Let's try pair programming and notice the impact on code quality and number of defects"), technical practices (TDD, CI, etc.), and how they interact with each other ("Here's how we can improve our daily check in").

These decisions provide a framework in which the team may interact and do work. Their agreements move them from chaos to complexity. By norming, they can start performing.

One for All

We tend to assume that our current software project has similar characteristics to our other software projects. But, often, the project characteristics vary significantly. Trying to deliver a complex project (open, high dimension, non-linear) using methods designed to solve simple projects (closed, low dimension, linear) doesn't work well.

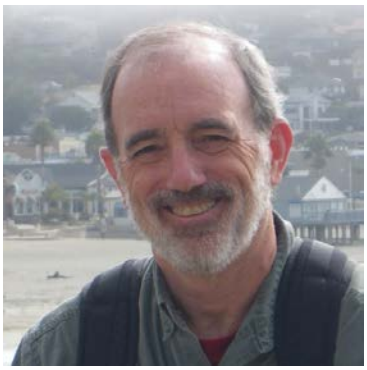
Fortunately, we can use methods designed to deliver complex projects to deliver simple projects. If you're going to assume anything, assume that you have a complex project and use agile principles. That way, your methods will solve the problem. **{end}**

don@donaldgray.com

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

- References

You can find a number of my articles on my website www.donaldegray.com or Better Software magazine, StickyMinds.com, AgileConnection.com and techwell.com.



Don Gray

DELIVERING CUSTOMER VALUE

MENTORING & TRAINING

for EXECUTIVES, MANAGERS & TEAMS

don@donaldegray.com

336.414.4645

www.donaldegray.com