Odyssey and Other Code Science Success Stories

John Manzo
AgileTek L.L.C.

Code Science® is an agile software development method based on eXtreme Programming (XP). This article describes the success achieved using code science to develop a complex industrial automation application. With a brief review of XP as background, code science is described in terms of refinements made to XP in applying it to a wide variety of application domains and industries over a period of almost four years. Included are real-world insights from the developers' experience in applying this agile development method, concluding with a quantitative measure of the effectiveness of XP since its inception almost four years ago.

Using Code Science®, an agile software development methodology based on eXtreme Programming (XP), we[1] recently delivered an application (code-named *Odyssey*) consisting of 400,000-plus executable source lines of code (ESLOC) to one of the world's premier industrial automation companies. The application was written in C++ by as many as 17 developers (including some of our customer's staff) in approximately 15 months. We are geographically remote from this customer.

Odyssey was delivered a month and a half ahead of schedule with a productivity rate of 43 ESLOC per coding hour. During the project duration, approximately 2,400 defects were found and fixed, yielding a *captured* defect density of six defects per thousand lines of code (KLOC). During a thorough, more than six-week customer-conducted acceptance test, only about 200 defects were found (none severe), yielding a *delivered* defect density of 0.5/KLOC. The customer is delighted with the product and is confident of the competitive edge achieved.

The Application

The Odyssey program consists of two distinct but related scalable vector graphics applications. The first is a run-time application that issues real-time commands from a touch screen panel to devices known as programmable logic controllers, which are used in manufacturing assembly processes. The second is a panel design application that enables human-machine interface engineers to develop the graphical equivalent of a hardware panel made up of buttons, gauges, and other control and monitoring devices.

Why Agile Methods

We began experimenting with XP several years ago, and actually began our first XP project a few months before Kent Beck published his first book on the subject [1]. Before that time, we used several traditional waterfall and rapid application design-based methods. We were impressed at how quickly our first XP project was completed.

In a side-by-side comparison of XP and waterfall on the very same project, the XP team delivered their final product when the other team was less than 50 percent complete. Since then, we refined our initial XP approach to encompass successive refinements that became known as Code Science.

> *"In a side-by-side comparison of XP and waterfall on the very same project, the XP team delivered their final product when the other team was less than 50 percent complete."*

Code Science is *largely* based on the twelve tenets of XP. These are as follows:

1. Customer at the Center of the Project. The customer is treated as a full-fledged member of the development team with access to all the information that the rest of the team is privy to (e.g., defect logs, issue lists, etc.).
2. Small Releases. Simple releases are put into production early and updated frequently on a very short cycle (two to three days). New versions are released at the end of each iteration (three to five weeks).
3. Simple Design. A program built with XP should be the simplest program that meets the *current* requirements.
4. Relentless Testing. XP teams focus on validation of the software at all times. Programmers develop software by writing tests first followed by software that fulfills the requirements reflected in the tests. Customers provide acceptance tests that enable them to be certain that the features they need are provided.
5. Refactoring. The system design is improved throughout the entire development process. This is done by keeping the software clean, without duplication, as simple as possible, and yet complete – ready for any change that comes along. (Martin Fowler defines refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [2]).
6. Pair Programming. XP programmers write all production code in pairs: Two programmers work together at one machine.
7. Collective Ownership. All the code belongs to the all the programmers. This enables the team to work at full speed. When something needs changing, it can be changed without delay. It is important to note that an effective configuration management discipline is an important enabler of this practice.
8. Continuous Integration. The software system is integrated and built multiple times per day (ideally, every time a task is finished). Continual regression testing prevents functional regressions when requirements change. This also keeps the programmers on the same page and enables very rapid progress.
9. 40-Hour Workweek. Tired programmers make more mistakes. XP teams do not work excessive overtime, which keeps them fresh, healthy, and effective.
10. On-Site Customer. An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions.
11. Coding Standards. For a team to work effectively in pairs and to share ownership of all the code, programmers need to write the code in the same way with rules that ensure the code communicates clearly.
12. Metaphor. Development is guided with a

® Code Science is registered in the U.S. Patent and Trademark Office.

simple shared story of how the overall system works. XP was originally used to develop a payroll program at Chrysler Corporation [3]. The team used the metaphor of an assembly line to describe the process of *building* a payroll check.

The key tenet in XP is iterative development and the unforgiving honesty of working code. The concept of iterative development has been around for a long time. However, XP does have some limitations such as scaling – the ability to add large numbers of developers to a project that requires them. (Most XP practitioners consider six to 12 developers to be the practical limit.) It was necessary to modify XP to develop a methodology that would work on large projects, across multiple application domains, and for clients with diverse and sometimes very specialized needs, for example, regulatory environments such as the Food and Drug Administration (FDA), where there is a strong need for extensive documentation.

The Code Science Difference

A way to quickly understand Code Science is to think of it as XP with a delta (a set of differences). Some of the differences are additive (+), some are subtractive (-) and some are simply modifications or refinements (▲). Following are a set of differences defined.

+ Business Process Analysis

In employing XP there is an implicit assumption that the client basically knows what it wants and, therefore, the requirements gathering process can begin with user stories. Although this is often the case, many of our clients need to focus and solidify their ideas and, most importantly, determine with clarity what they *need* rather than what they *want*. To accomplish this, we developed a process that helps bring focus and understanding to the client's business needs, prioritizing features and functions in terms of the business value they represent. This first step, which is formally absent from XP, is a step we can take when *necessary* to ensure that the story-gathering effort produces stories based on a real vs. perceived need.

+ Delphi Estimation

The Delphi method of estimating involves three or more participants who discuss the work and provide anonymous estimates of the time for completion (usually in units of *perfect programmer hours* – i.e., an ideal, no interruption, period of time). These estimates are tallied and a mean and standard deviation is made known to the participants. Discussion ensues among the participants as to the differences in the estimates (which remain anonymous). This continues for successive rounds (usually three) until the standard deviation (a measure of uncertainty) is made sufficiently narrow. Once the number of perfect programmer hours is known, a loading factor is applied to convert this estimate to *real* programmer hours.

+ Componentized Architecture

For complex systems, it is especially important to assure conceptual integrity in the final product. Also, because complex systems can be large, it is also important to enable the system to be developed in an environment of distributed ownership. Among the least understood areas of XP is the notion of *design-as-you-go* through refactoring. To some, especially those who equate design and architecture, this means no up-front architecture, and, by implication, any architecture that the delivered system may have is a de-facto one at best.

> *"The best architectures are isomorphic (one-on-one) mappings between problem and program space."*

Architecture of a system simply means identifying the constituent components of the system and defining the interrelationship(s) between them. The best architectures are isomorphic (one-to-one) mappings between problem and program space. This ensures that a system's underlying structure and components mirror the problem being solved. This means that for the *program* to change requires that the *problem* changes and, therefore, you are *change-proofing* your program. While there may be more efficient ways to solve a problem (e.g., creating one module to perform similar functions by invoking it in a context sensitive way), this efficiency will almost always come at the expense of time spent debugging and later modifying the program if one or more of the functions change.

However, it also means something more. By defining the relationships between the various components, one has gone most of the way toward establishing agreements for the interfaces. The power of interface agreements is that they serve as restrictive liberators. In other words, the individuals working on various system components are free to design the internals of those components without regard for potential untoward effects on the rest of the system – so long as the interface agreements are honored.

By spending a relatively small amount of time up front, one can ensure both a product with conceptual integrity and a project that can scale.

+ Automated Contract and Regression Testing

Given that XP is premised upon the need to embrace change, making it easy to perform regression testing is an important part of any XP project. We have taken this to the next level by implementing the capability to perform contract testing, which checks for the existence of predefined pre-conditions, post-conditions and class invariants. (As an example, an overdrawn flag in your checking account is invalid if there is a positive balance remaining after the last transaction.)

+ Story Actors

We have added to the notion of *stories* the concept of story *actors*. Actors are personifications of the various categories of users the system will encounter. Thinking of the requirements in terms of actors brings the requirements to life as well as unmasks nuances that would otherwise remain invisible to both the developers and the customer.

+ Wall Gantts

Frequently used in project management, a Gantt chart provides a graphical illustration of a schedule that helps to plan, coordinate, and track specific tasks in a project. We have taken the concept one step further and adapted it to agile methods by creating a physical construct using twine, pushpins, and index cards. The twine is used to create a line on a wall. Tasks, written on cards, are folded in half and hung on the line (one line for each project participant). Index cards with dates (one for each day of an iteration, which usually lasts three to five weeks) are pinned across the top of the chart.

Physically constructing the Gantt chart makes it very easy to move tasks around, drive out dependencies, and load balance. Because the chart is wall size, it is easy for the team to stand around the chart to discuss the state of the project in near real-time (each day starts with a stand-up meeting). The wall Gantt also provides clear ownership for development efforts, encourages accountability, and serves as the team's *war room* and center of the project universe.

+ Automatic Document Generation

Through a tool we have built called Doc-It (similar to JavaDoc), we are able to reduce the burden and streamline the process of generating documentation that describes the inner workings of the code. Experience shows that it is a poor practice to separate

documentation from the code that it describes. Updating source code documentation is difficult enough, but once the documentation is separated from the code, it is "out of sight, out of mind." To deal with this, a programmer simply needs to tag a comment in the source code and Doc-It creates automatic HTML Application Program Interface documentation with every build.

Doc-It traverses source code directories, creating a navigable hierarchy (directory, class, method) and creates a Web page for each source file. This makes the documentation easily accessible to new and existing team members. It also makes the documentation easily accessible to clients during co-development or during knowledge transfer phases.

▲ *Pair Programming*

Although our experience proves pair programming to be extremely effective, for many routine programming tasks, pair programming has not shown itself to be cost effective. On the other hand, for anything either algorithmically or logically complex, pair programming is a must. The default is to program in pairs, but the team gets to decide which modules will be coded solo.

- 40-Hour Workweek

While we strive to provide the highest quality of life for all our staff members, it is unrealistic to expect that our client's time-critical requirements will not sometimes necessitate sustained periods of activity. Treating a 40-hour workweek as a hard requirement is often impractical.

- Metaphor

Metaphor is not included in Code Science. While we concede that it has benefits, so far we have not found a need to incorporate the use of metaphor in our methodology.

+ Flexibility to Meet Client's Special Needs

Some of our clients have special needs that are not accounted for by pure XP (e.g., in highly regulated environments such as biomedicine, the FDA requires specialized documentation and traceability for certain types of software). Code Science eliminates this XP limitation by incorporating a *special needs* provision in our methodology.

Application to Odyssey

Code Science is used on all Geneer software development projects. The Odyssey project was no exception. However, no two projects are the same. Each emphasizes certain of the specific tenets described above to differing degrees. In the interest of brevity, we describe some of our developers' more salient experiences and insights in applying these tenets.

The Customer Is at the Center

XP talks about having the customer on-site. While this is ideal, our experience in using XP/Code Science over the last four years is that it is seldom practical unless your customer is internal. In the case of Odyssey, the customer was located hundreds of miles away.

More important than physical location, however, is putting the customer at the center of your project as described earlier. In an XP/Code Science project, there is no attempt to hide information from the customer. While we did not insist the customer be physically in our facility, we did request that they be present during iteration planning, periods of critical knowledge transfer, or to approve test plans and validate their results – usually at the beginning/end of iteration. When this was impractical, or for routine communications, we used e-mail, conference calls, WebEx sessions, or video-conference. With active customer participation, the resulting product can be everything that the customer expects it to be.

Refactoring

Refactoring does not mean re-working. Do not partially write a feature with the *intent* of refactoring to get it complete later. Keep the changes simple, but keep them atomically complete.

Pair Programming

Pair programming was especially useful in ramping up a new staff member. It was also quite useful for chasing down complex defects. For simple modules, the team found it more expedient to use the white board in pairs for 15 minutes, then program solo.

Continuous Integration

The team performed builds at least daily, more often, two or three times a day. With a good automated build program, you cannot build too often. Our builds are generated with a custom, home-grown application that creates builds at 4 a.m. and again at 3 p.m. This gives the team a fresh build every morning and also one to work on in the afternoon. Besides performing the physical build, we are also informed if the build is broken (e.g., cannot compile because of a syntax problem, or a configuration management issue – checked in one file and not another, etc.).

40-Hour Week

During a long period of peak activity, the team found it helpful to make their work environment homier. By making their workplace a more dorm-like environment, they significantly eased the stress of the long, often intense, workdays and nights.

Componetized Architecture

A high-level architecture was defined at the beginning of the first iteration, and as more information became available, more detail was added to successive iterations. Team leads would spend perhaps two days with their teams using white boards for a four to six-week iteration. We found that the biggest mistake one can make here is to attempt to get too detailed about something for which there is insufficient information.

Story Actors

Because most of the team never worked on an industrial automation application, actors helped the team get familiar with the client's domain. When the team took a field trip, they could identify the user types by their actor names (representative of their role-play, more than their job title). It helped the team understand the business and how the product would be used in stories. The requirements were written in terms of how the system would be used, vs. desired functions. By associating who is doing what, it helps conceptualize and compartmentalize the functions.

Wall Gantts

Wall Gantts make load balancing easy and kept the project on track. The whole team sees the actual size of the function based on the task cards and there is great satisfaction in putting completion stickers on each card. An extremely useful management tool, Wall Gantts also helped to reveal issues and expose risks.

Large Team Experience

Although the overall team was divided into subteams, stand-up meetings were typically with the entire team. Team leads summarize and add detail with other team members as needed. As tasks are completed, people can move from one team to another.

Conclusion

During a period of almost four years, XP/Code Science has been employed on 14 projects across a wide variety of application domains and industries such as aerospace, telecommunications, banking and finance, pharmaceuticals, consumer goods, and even pari-mutuels. These projects ranged in size from 10 KLOCs for a Personal Data Assistant client, to more than 400 KLOCs

CONTINUED ON PAGE 30

Should You Be More Agile?

Rich McCabe and Michael Pollen
Software Productivity Consortium

Agile software development techniques are an effective response to many problems still plaguing development projects. Although there are a number of issues to consider, almost any project can become more agile to its benefit. What exactly does it mean to be more agile? Words like predictable, cost-effective, and mature are more often used to characterize desirable software development processes. Agile development has come into focus recently due to the popularity of its most widely known interpretation, eXtreme Programming, but some of its foundations go back as far as 20 years. This article addresses some of the questions about agile: What is agile? Who needs to be agile? How can any project not creating small business applications seriously consider agile development? Is agile development an "all or nothing" proposition?

Agile Development: Weed or Wildflower?[1]

David Kane
SRA International

Steve Ornburn
GBC Group, Inc.

The Software Engineering Institute's Capability Maturity Model® (CMM®) has been a major force for software process and acquisition improvement in the federal government's civil and defense communities for the past decade. Major investments have been made by the government, their contractors, and many other organizations to make software development more consistent and reliable. The CMM provided an alternative to the cowboy programmer archetype. Amid this backdrop of progress, a new trend in software development has emerged – agile development, which aims to build software faster and more flexibly than traditional approaches. Agile values "individuals and interactions over processes and tools" [1]. *For organizations that have invested in a CMM, do agile methods represent the rebirth of the cowboy – a weed to be stamped out? Or, are agile methods a reasonable way to build software in a world in which needs are changing at an ever-increasing pace – a wildflower to be nurtured? This article looks at whether there is a home for agile methods in communities that have have embraced the CMM.*

Editor's Note: Due to space constraints, CROSSTALK was not able to publish these articles in their entirety. However, they can be viewed in this month's issue on our Web site at <www.stsc.hill.af.mil/crosstalk> along with back issues of CROSSTALK.

CONTINUED FROM PAGE 21

for the industrial automation project described herein. The languages were mostly C++ but also included C, HTML, VB, and SQL. Productivity ranged from a low of 21 to a high of 48 lines of code per coding hour, averaging 35 lines of code per coding hour.

Compared to projects conducted before adopting this intensely practical and agile software development discipline, our cost per line of code and defect rates were drastically reduced while our development velocity was significantly increased. Our most recent audit revealed an *overall* average productivity index of 22 [4]. This index is a management scale corresponding to the overall process productivity achieved by an organization during the main software build. An index of 25 is considered among the highest ever recorded.◆

References

1. Beck, Kent. eXtreme Programming Explained: Embrace Change. Boston: Addison-Wesley, 1999.
2. Fowler, Martin, et. al. Refactoring: Improving the Design Of Existing Code. Boston: Addison-Wesley, 1999.
3. C3 Team. "Chrysler Goes to Extremes." Distributed Computing. Oct. 1998 <www.xprogramming.com/publications/distributed-computing .html>.
4. Putnam, Lawrence H., and Ware Myers. Measures of Excellence: Reliable Software on Time, Within Budget. Upper Saddle River: Prentice Hall/Yourdon Press, 1992.

Note

1. "We" as mentioned throughout this article refers to the Geneer company.

About the Author

John Manzo has spent more than three decades of his career in software engineering, and has contributed to and made significant accomplishments in the development of software, computer, and telecommunications solutions. Manzo comes to AgileTek from Geneer where he was chief technology officer, and brings with him a legacy of broad and deep experience in agile development methods. Earlier in his career, Manzo was recognized for his development of the Fire Control software for the Navy's highly successful AEGIS system – one of the largest and most complex software developments ever delivered to the Department of Defense. He has served as a representative to the President's National Science Advisory Board, and served as an adjunct faculty member of Harvard University where he developed, and for several years taught, "The Management of Software Engineering."

AgileTek
934 South Golf Cul de Sac
Des Plaines, IL 60016
Phone: (847) 840-3765
Fax: (847) 376-8308
E-mail: jmanzo@agiletek.com