Process Agility and Software Usability: Toward Lightweight Usage-Centered Design

Larry L. Constantine
Constantine & Lockwood, Ltd.
University of Technology, Sydney

Abstract. A streamlined and simplified variant of the usage-centered process that is readily integrated with lightweight methods is outlined. Extreme programming and other so-called agile or lightweight methods promise to speed and simplify applications development. However, as this paper highlights, they share with the "unified process" and other heavyweight brethren some common shortcomings in the areas of usability and user interface design. Usage-centered design is readily integrated with these lightweight methods. As in extreme programming , ordinary index cards help streamline the process of modeling and prioritizing for design and implementation in successive increments. Links to selected Web resources on extreme programming, agile modeling, and other agile processes are also provided.

Keywords: usability, user interface design, usage-centered design, extreme programming, lightweight methods, agile methods, iterative development]

Software and Web applications development today can seem like a strange game. At one end of the playing field, in scruffy jerseys of assorted color and stripe, are the unruly rabble representing the method-less chaos that all too often passes for programming in many organizations; at the other end we see the advancing phalanx of the heavyweight team, a bright letter *U* (for Unified) emblazoned on the chest of each hulking goliath. These true heavyweights of modern software development—the Unified Process and its supporting player, the Unified Modeling Language—have shrugged off scholarly scrutiny and competition alike to lead the league in goals scored [Constantine, 2000b].

Between the contrasting extremes of libertarian license and methodological muddling, the centerfield of the process playing field is currently dominated by the real crowd pleasers of

Reprinted from *Information Age,* August/September, 2002. An earlier version first appeared in The Management Forum, *Software Development, 9,* (6), June 2001; reprinted in L. Constantine, ed. *Beyond Chaos: The Expert Edge in Managing Software Development.* Boston: Addison-Wesley, 2001. The author may be contacted at Constantine & Lockwood, Ltd., 58 Kathleen Circle, Rowley, MA 01969; tel: 1 (978) 948 5012; fax: 1 (978) 948 5036; email: larry@foruse.com

modern software development: a small group of relatively young, fast-moving players—the so-called lightweight processes with names like XP, Crystal, SCRUM and FDD. To be both correct and *au courant,* however, we should follow the lead of a group of the leading coaches and promoters of these processes who met in February 2001 at Snowbird in Utah to discuss their current interests and future prospects. Wishing to shed the negative connotations of the term *lightweight,* they agreed thereafter to refer to their sundry methods as "agile" processes. Agile it is, then. After all, who would want to risk being thought of as the 98-pound weaklings of software and Web development?

Agile Players

Managers and developers who want to play today's game need to understand not only the league leaders like RUP, the Rational Unified Process, but also the agile competitors. A complete comparison of all the agile processes would be beyond the scope of this article; there are just too many and they differ in too many details. In any case, Martin Fowler has already provided a concise but detailed overview [Fowler, 2000]. In this article, the emphasize is on a management and decision-making perspective with a focus on some of the pluses and minuses common to the agile processes.

The archetype of the agile athletes of software development is XP, short for eXtreme Programming [Beck, 2000; Jeffries, et al., 2001]. Undoubtedly the best known and arguably the most successful of the agile processes—it is the subject of over a dozen recent books—in many respects XP typifies the entire diverse bunch. Like nearly every other methodology—light, heavy, or middleweight—XP and its agile teammates are really tossed salads that mix tricks and techniques with a mish-mash of methods, models, and management formulas, all dressed with tools, practices and even a bit of philosophy. Some advocates argue that, because the various pieces were all concocted to complement each other, you have to buy the whole assorted package of any given method, but many practitioners simply pick and choose the parts that seem to work for them and toss out the bits they don't like or don't believe.

The hallmarks of XP, as with most of the agile processes, are that it is more people-oriented than process-oriented and that it emphasizes flexible adaptation over full description. Along with the need for agility, the stress on effective teamwork puts a premium on good management and leadership skills. Success with most of the agile approaches requires particularly close, hands-on management aimed at smoothing the process and removing impediments.

Oddly enough, one of the more controversial aspects of XP teamwork is also one of the best-established best practices of software development: pair programming. Also known as the dynamic duo model, pair programming puts two programmers in front of a single screen. By providing instantaneous inspection for every line of code that gets cut, the technique radically reduces bugs, improves programming style, and cuts the overall cost of programming. Pioneered by C/C++ guru P. J. Plauger and popularized by yours truly [Constantine, 1992], pair programming is a proven technique with a solid track record.

Philosophy is also a big part of nearly all the agile methods. XP itself is built around a set of core values: simplicity, communication, testing, and fearlessness. Yes, fearlessness. To this list, Scott Ambler, a recent banner-waver for model-driven agile processes, has added the value of humility [Ambler, 2002]. Jim Highsmith's Adaptive Software Development [Highsmith, 2000], which has also been classified with the lightweight processes, is more framework and philosophy than prescribed practices, especially compared to such precisely delineated approaches as XP or Pete Coad's Feature Driven Development (FDD), for which, check the **Links** below.

The rules of the agility game are relatively simple. Work in short release cycles. Do only what is needed without embellishment. Don't waste time in analysis or design, just start cutting code. Describe the problem simply in terms of small, distinct pieces, then implement these pieces in successive iterations. Develop a reliable system by building and testing in increments with immediate feedback. Start with something small and simple that works, then elaborate on successive iterations. Maintain tight communication with clients and among programmers. Test every piece in itself and regression test continuously.

Although some programmers and their managers may think of agile methods as a license to hack, it should be clear that following its philosophy and practices takes genuine discipline. I have heard some managers refer to XP as structured hacking, and that may not be far from the truth.

The Pros

Perhaps the biggest selling point of the agile processes is their weight—or lack thereof. There is simply far less to them than to the leading heavyweights, which means potentially less to learn and master. That does not mean that agile processes are easy, any more than it means that agile process can substitute for programming skills and developmental discipline—but it does mean that there is less to explain about the processes themselves.

Reduced overhead is another strong point. Unlike the more heavy duty processes that emphasize diverse deliverables and numerous process artifacts, the agile processes concentrate on code. Design is on-the-fly and as needed. Index cards and whiteboard sketches take the place of a slew of design documents, and brief, standup confabs replace protracted meetings.

Early results are yet another point of appeal of agile processes. With short release cycles that produce a fully-functional system on every iteration, agile methods enable clients to begin using a simplified working core with limited but useful capability early in a project.

For clients, managers, and developers alike, a major potential payoff comes from the ways in which agile methods reduce the defect injection and leakage rates: the number of bugs created in the first place and the number that sneak past successive phases of development. More reliable code means more up-time, cheaper development, and less support and maintenance.

The simple philosophies of the agile processes appear to translate fairly well into practice. Despite—or because of—the streamlining, agile processes seem to have worked well for a variety of real-world projects. The success stories far outnumber sob stories (but of course!), at least for well-trained teams working on appropriately limited projects.

What goes round

What are the risks and shortcomings of the agile methods? Readers who have been around long enough or have a sufficient grounding in the history of programming may be experiencing a strong sense of déjà vu. Some thirty years ago, IBM was touting so-called chief programmer teams [Baker, 1972], small, agile groups headed by a strong technical lead who could hold the implicit architecture in mind without resorting to much explicit design. Applications were constructed with thorough inspection and testing through successive refinements. The objective was to have working code at all times, gradually growing it through successive enhancements into a full-scale working system.

Like XP and its agile teammates, chief programmer teams also seemed to enjoy early victories for awhile, but the model ultimately foundered on the shoals, caught between the Charybdis and Scylla of charisma and scale. Led by gifted technical managers and applied to problems of modest size, the approach worked well, but there were only so many Terry Bakers in the world and not all problems could be sliced and diced into the right pieces for speedy incremental refinement.

The same issues challenge the agile methods today. There are only so many Kent Becks in the world to lead the team. All of the agile methods put a premium on having premium people. They work best, if at all, with first-rate, versatile, disciplined developers who are highly-skilled and highly motivated. Not only do you need skilled and speedy developers, but you need ones of exceptional discipline willing to work hell-bent-for-leather with someone sitting beside them watching every move.

Scale is another problem. When I surveyed colleagues who are leading lights in the world of light methods, they agreed that the agile processes do not readily scale up beyond a certain point. While a few of XP efforts with teams of 30 developers have been launched, the results have been mixed. Alistair Cockburn claims that his family of methods known as Crystal [Cockburn, 2001; 2002] (and see **Links** below) has been used for larger projects, but the general consensus seems to be that 12 to 15 developers is a workable upper limit for most agile processes. The tightly coordinated teamwork needed for these methods to succeed becomes increasingly difficult beyond 15 or 20 developers.

All the agile methods are based on incremental refinement or iterative development in one form or another. The basic premise is that it is easier to add capability to a compact base of well-written existing code than to build a complete application in one fell swoop. Within the prescribed short release iterations of agile processes, a small team can only do so much. Systems that total some 250 thousand lines of code may be achievable over many such iterations, but a million lines is probably out of reach. Moreover, some projects and applications cannot, for all practical purposes, be broken into nice and neat 30-, 60-, or 90-day increments.

What's the use?

Informants in the agile process community have confirmed what numerous colleagues and clients have reported to me. XP and the other light methods are light on the user side of software. They seem to be at their best in applications that are not GUI-intensive. As Alistair Cockburn expressed it in email to me, this "is not a weak point, it is an absence." User-interface design and usability are largely overlooked by the agile processes. With the possible exception of DSDM and FDD (see **Links** below), users and user interfaces are all but ignored. To be completely fair, neglect of users and user interface design is a failing shared as well by the bulk of the big-brother behemoths in the heavyweight arena. Nevertheless, it is significant that no representatives from the human-factors, interaction design, or usability communities were invited to participate in the formation of the Agile Alliance (see **Links** below).

Some agile methods, like XP, do explicitly provide for user or client participation in pinning down requirements and setting scope through jointly developed scenarios, known as *user stories.* It is noteworthy that user stories are typically written by customers or customer representatives, not necessarily by genuine direct end-users. Anyone familiar with usage-centered design [Constantine and Lockwood, 1999; 2002] understands the importance of this distinction. Users outnumber customers and are the critical source of information for user interface design. Clients legitimately make decisions regarding scope and capabilities, but as often as not do not really understand genuine user needs. Moreover, user stories are concrete scenarios, detailed and believable vignettes that, like conventional concrete use cases, tend to assume many aspects of the yet-to-be-designed user interface.

Despite the sometime use of user stories, when it comes to user interface design the agile processes maintain their emphasis on minimalist design process and rely on repeated refinement to pound the user interface into shape, favoring somewhat simplistic forms of iterative paper prototyping rather than model-driven design or thorough forethought. Particularly with XP, which relies so heavily on testing, GUI-intensive projects pose particular problems. Testing of user interfaces is labor intensive and time consuming; automated user-interface testing is difficult if not impossible except at the most elementary level. True usability testing requires repeated testing with numbers of users under controlled settings. User or client reactions to paper prototypes are no substitute and can even be completely misleading. What people will say they like or claim is workable when they see a paper sketch will often prove unworkable in practice.

The most critical shortcoming of nearly all techniques that are based on iterative expansion and refinement in small increments is the absence of any comprehensive overview of the entire architecture. For internal elements of the software, this shortcoming is not fatal, because the architecture can be refined and restructured at a later time. Refactoring [Fowler, 1999] can, in many cases, make up for the absence of a complete design in advance. User interfaces are a different story.

For user interfaces, the architecture—the overall organization, the navigation, and the look-and-feel—must be designed to fit the full panoply of tasks to be covered. When it comes to the user interface, later refinement of the architecture is not as acceptable because it means changing the system for users who have already learned or mastered an earlier interface. Even small adjustments in the placement or form of features can be problematic for users. Whereas refactoring of internal component structure need not necessarily have any manifestations on the user interface, redesigning the user interface architecture is unavoidably disruptive for users. Maintaining a consistent and comprehensible look-and-feel as new features and facilities are added to the user interface becomes increasingly difficult as the user interface evolves through successive releases, as the history of many large commercial products attests. Periodic rework of the user interface as a whole to overcome the entropy of inconsistent expansion imposes a huge burden on users. For these reasons, the user interface is one aspect of the system that absolutely must be designed, designed completely, and designed right before its code is cut.

Iterative prototyping is an acceptable substitute for thorough user interface design only when the problems are not too complicated, when there are not too many screens with too many subtleties, and where a rather pedestrian and uninspired solution will suffice. Software with intricate user interface problems or for which usability will be a major factor in product success demand a more sophisticated, model-driven approach to user interface design. This is where usage-centered design enters the picture [Constantine & Lockwood, 1999].

Agile Usability Processes

When Lucy Lockwood and I began developing usage-centered design in the early 1990s, we did not set out to create a lightweight process. Our philosophy has always been simply to use whatever tools and techniques helped us design more usable systems in less time. We draw diagrams only when we have to or when drawing them is faster than not drawing them. We should not be surprised, then, to learn that usage-centered design is increasingly mentioned in company with the other agile processes.

Where usage-centered design parts company from most of the agile processes is in the emphasis we place on modeling and on driving the user interface design and development by models. Scott Ambler's Agile Modeling *nee* Extreme Modeling [Ambler, 2002] (see **Links** below) is an exception to widespread model-phobia among the lightweight approaches. In usage-centered design, the profiles we build of roles users play in relation to the system directly inform the task model we construct to capture the nature of the tasks to be supported by the system. The task model—collections of interrelated use cases in simplified, abstract form—directly drives how information and capability are collected and arrayed on the user interface.

In its most agile incarnation, usage-centered design works some of the same card magic that powers XP [Jeffries, 2000]. We model task cases on index cards, thus keeping them short and sweet and facilitating easy collection and distribution. We shuffle the cards to prioritize them in terms of business and end-user importance. We sort them into related groups that help us construct complete usage scenarios and that guide the collection of user interface components into screens, pages, or dialog boxes.

The same task cases that steer the user interface design are grist for the programming mill that grinds out the objects and methods that power the software. In detailing task cases, we focus on user intentions and system responsibilities in order to help distinguish genuine needs of users from mere wants and wishes. In short, usage-centered design probably qualifies as an agile process, even if it wasn't contrived with that in mind. Moreover, it supplies a critical piece missing from other methods—an effective and efficient scheme for designing highly usable user interfaces. However, usage-centered design makes no claim to being an end-to-end process for cranking out software. Rather, it is an adjunct to an effective agile process, the missing link back to users that can turn an incomplete process into one that can reliably deliver good solutions for users, not just good code.

For success as an agile process, usage-centered design requires close collaboration and communication with users, domain experts, and clients, but especially with genuine users and qualified experts in the application domain. If users and domain experts are not actual participants in the design process, then they need to be readily available for quick review and validation of the design team's work. The JITR (Just-in-Time-Requirements) technique [Constantine, 2000a], in which users and/or domain experts are immediately available to answer questions and clarify issues as they arise, is another approach that works for accelerated, agile design.

In outline, the most "extreme" or agile incarnation of usage-centered design follows a simple process of card-based modeling and decision-making. The use of index cards not only speeds and simplifies the process, but the limited visual "real-estate" of index cards forces parsimonious modeling. The chief disadvantage of card-based modeling—that models are not software compatible without transcription—is not a major problem in agile development where direct communication and face-to-face discussion is substituted for documentation and diagrams. While the use of cards to carry information does not preclude drawing a picture for clarification, it does not require it either.

We have long argued that collaboration, as in Joint Essential Modeling [Constantine and Lockwood, 1999], yields superior results more quickly. As with other more elaborate and detailed variations on the usage-centered theme, the agile activities outlined below are best carried out within a team that includes both designers and developers plus at least one user or user surrogate (such as a domain expert).

1. **Inventorying roles.** Construct an inventory of the roles users can play in relation to the planned system by brainstorming directly onto index cards.

2. **Refining roles.** Review and refine the inventory on cards, then briefly describe salient aspects of each role on its index card. Keep track of questions and ambiguities about user roles that arise along the way. If you are not working directly with users, these will need to be clarified through discussions, observations, or interviews.

3. **Prioritizing roles.** Sort the user role cards to rank them in order of priority for project success. Consider deferring support for low priority but challenging user roles.

4. **Inventorying tasks.** Construct an inventory of task cases (use cases in essential) to support the identified user roles by brainstorming directly onto index cards, beginning with the highest priority roles. Keep track of unanswered questions and unresolved ambiguities about user tasks to be clarified through discussions, observations, or interviews with users.

5. **Prioritizing tasks.** Sort the task case cards to rank them, first by anticipated frequency or commonality, then by overall priority for project success. Next, sort the task case cards into three heaps: required (do first on this release), desired (do if time on this release), and deferred (for later release cycles). Mark the cards by their assigned heap.

6. **Describe tasks.** For required task cases—along with any desired ones that appear to be critical, complex, unclear, or interesting—write out the narrative body of the task on the index card. The narrative should cover the "success case" (normal flow of events) in essential form (abstract, simplified, technology-free) using the standard two-column format of user intentions and system responsibilities. Extensions, alternatives, or "failure" cases of the narrative can be added on the back of each card later. If not working collaboratively with users or user surrogates, clear up remaining questions and issues through discussions, observations, or interviews with users if needed.

7. **Organize tasks.** Cluster task case cards (all of them) into co-operating groups based on the likelihood that they will be enacted together within a common scenario or sequence or timeframe. Create duplicate cards for tasks that fall in more than one cluster, but clearly identify them as "copy cards."

8. **Paper prototype.** Treat each cluster as the tentative set of tasks to be supported by an interaction context (screen, page, dialog, or the like) in the user interface, and then sketch an initial paper prototype for that part of the user interface, concentrating on the required task cases, but considering others. (All task cases are considered to ensure that the overall user interface architecture is sound.)

9. **Refine prototype.** Inspect the prototype with users and clients using scenarios derived from the task cases. Revise and refine the paper prototype based on inspection results.

10. **Interface construction.** Begin programming of the user interface or presentation layer based on the paper prototype and associated task cases.

As in nearly all agile methods, the assumption here is that development proceeds through successive release cycles. Successive iterations will pick up additional roles and task cases to guide refinement and expansion of the user interface. On each successive iteration or release cycle, then, the user roles, tasks, and co-operating clusters are reviewed and refined as needed before the next set of paper prototypes is sketched, inspected, and refined.

Not all programming has to await design of the user interface. In an accelerated development process based on agile usage-centered design, design and programming of internal and back-end components that are not directly dependent on the form and details of the user interface design can proceed in parallel with the above process. Furthermore, once task cases are identified, these can be harvested for possible programming implications independent of the user interface design. When the revised paper prototype is done, each interaction context within it and its associated task cases become grist for the programming mill, whether extreme or "normal."

Architecture? What Architecture?

Except for the grouping of tasks into co-operating clusters, the abbreviated outline above still omits any systematic consideration of the overall architecture of the user interface. The denizens of XP dismiss all such architectural design as BDUF, or Big Design Up Front. But some minimum up-front design is needed for the user interface to be well-organized and to present users with a consistent and comprehensible interface. Experience in a number of short-cycle projects, particularly on the Web [Constantine and Lockwood, 2002] has clarified what is the minimum architectural consideration for agile usage-centered design of user interfaces.

The minimal up-front design for user interfaces is not very big. You need to establish three things:

1. an overall organization for all the parts of the user interface that fits with the structure of user tasks

2. a versatile common scheme for navigation among all the parts;

3. a visual and interaction scheme that provides a consistent look-and-feel to support user tasks.

The overall organization is already covered, although barely, by sorting task cases into co-operating clusters. The navigation scheme requires thinking through how best to enable users to move among different views or interaction contexts and how these will be organized into collections and presented visually to users. The third minimal requirement means constructing—and refining through successive iterations or release cycles—a kind of abstract style guide that describes the visual and interaction techniques to be used throughout the application. For examples and more discussion of navigation architecture and visual and interaction schemes, see [Constantine and Lockwood, 2002].

Mid-field Scrimmage

Between the world of hack-and-slash programming and that of obsessive-compulsive docu-mania is a middle ground offering hope for beleaguered managers, over-burdened programmers, and under-served users alike. With this in mind, I left readers of my final column for the *Software Development* Management Forum with one piece of simple advice. If you are ready to leave behind both ochlocracy and bureaucracy in software development, then best join the agile few and heed the words of the venerable William of Okham. “It is vain to do with more what can be done with less.”

References

Ambler, S. 2002 *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process.* New York: Wiley.

Baker, F. T. 1972. “Chief programmer team management of production programming,” *IBM Systems Journal, 11* (1).

Beck, K. 2000. *Extreme Programming Explained,* Reading, MA: Addison-Wesley.

Cockburn, A. 2001 *Writing Effective Use Cases.* Boston: Addison-Wesley.

Cockburn, A. 2002 *Agile Software Development.* Boston: Addison-Wesley.

Constantine, L. L. 1992. “The benefits of visibility,” *Computer Language Magazine,* September. Reprinted in *The Peopleware Papers,* Upper Saddle River, NJ: Prentice Hall, 2001.

Constantine, L. L. 2000a. “Cutting Corners: Shortcuts in Model-Driven Web Development,” *Software Development, 8,* (2), February. Reprinted in L. Constantine, ed. *Beyond Chaos: The Expert Edge in Managing Software Development.* Boston: Addison-Wesley, 2001.

Constantine, L. L. 2000b. “Unified Hegemony,” *Software Development, 8,* (11), November. Reprinted in L. Constantine, ed. *Beyond Chaos: The Expert Edge in Managing Software Development.* Boston: Addison-Wesley, 2001.

Constantine, L. L. and Lockwood, L. A. D. L. 1999. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design,* Reading, MA: Addison-Wesley.

Constantine, L. L., and Lockwood, L. A. D. 2002 “Usage-Centered Engineering for Web Applications,” *IEEE Software, 19* (2): 42-50, March/April.

Fowler, M. 1999 *Refactoring: Improving the Design of Existing Code.* Reading, MA: Addison-Wesley.

Fowler, M. 2000. “Put Your Process on a Diet,” *Software Development, v. 8,* 12, December. Expanded version on the Web, http://www.martinfowler.com/articles/newMethodology.html

Highsmith, J. 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* New York: Dorset House, 2000.

Jeffries, R., Anderson, A. Hendrickson, C. 2001 *Extreme Programming Installed.* Boston: Addison-Wesley.

Jeffries, R. 2001. “Card magic for Managers,” *Software Development, 8,* (12), December. Reprinted in L. Constantine, ed. *Beyond Chaos: The Expert Edge in Managing Software Development.* Boston: Addison-Wesley, 2001.

Links

Agile Alliance
http://www.agilealliance.org/

Crystal
Alistair Cockburn: http://crystalmethodologies.org/

DSDM
http://www.dsdm.org/default1.asp

dX
Robert Martin: http://www.objectmentor.com/publications/RUPvsXP.pdf

Extreme Modeling
Scott Ambler, Modeling: http://www.extreme-modeling.com/

FDD
Peter Coad: http://www.togethercommunity.com/coad-letter/Coad-Letter-0070.html

Fowler, Agile methods overview: http://www.martinfowler.com/articles/newMethodology.html

SCRUM
Ken Schwaber: http://www.controlchaos.com/
Jeff Sutherland: http://jeffsutherland.com/scrum/

Usage-Centered Design

The usage-centered design resource on the Web: http://foruse.com/.

XP

Jim Highsmith, J. "Extreme Programming." http://www.cutter.com/ead/ead0002.html

Ron Jeffries. http://www.xprogramming.com/

Don Wells: http://www.extremeprogramming.org/