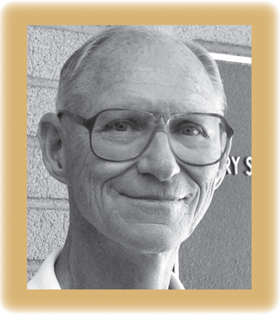
Contact Editor: Dale Strok ■ dstrok@computer.org

The Changing Nature of Software Evolution

Barry Boehm

Traditionally, software evolution took place after software development put a system in place.[1] There were even separate budgets, teams, and procedures for doing it. However, the pace of change in technology and competition has changed the nature of software evolution to a continuous process,[2] in which there's no neat boundary between development and evolution.[3–4]



Many traditional software development assumptions and practices haven't recognized this changing nature and increasingly find themselves in deep trouble as a result. The assumption that your project can be developed to a fixed-price, build-to-prespecification contract that treats "requirements creep" as something to be discouraged will increasingly result in an obsolete, brittle system that you can't evolve into something better. Dismissing your architects after the total-system design review means that nobody qualified to evolve it will still be available. Building the easiest parts first frequently makes it impossible to evolve into a scalable, safe, or secure system. Minimizing development costs by adopting numerous off-the-shelf products often leads to unaffordable evolution costs as your vendors ship new releases and stop supporting the old ones. Assuming that a single form of evolutionary development covers all situations often leads to unrealistic commitments and dead-end systems as situations change.

Evolutionary development takes several forms. Craig Larman's historical summary traces the earliest incarnations back to the early 1950s,[5] when Herbert D. Benington documented the Semi-Automatic Ground Environment (SAGE) project's "stagewise development."[6] Even all these years later, there are still times when this traditional one-step, build-to-spec approach is best. As much as one would like there to be, there's no one-size-fits-all software evolution approach that's best for all situations. For rapid-fielding, narrow-market-window situations, an easiest-first, "get something working quickly and productize it later" approach is best. But for enduring systems, an easiest-first approach is likely to produce an unscalable system whose architecture is unable to achieve high levels of performance, safety, or security. In general, software evolution now requires

- much higher sustained levels of systems engineering efforts,
- earlier and continuous integration,
- test and proactive approaches to address sources of system change, and
- greater levels of concurrent engineering and achievement reviews based on evidence of feasibility versus evidence of plans, activity, and system descriptions.[4]

Table 1 provides criteria for deciding which of the four primary classes of incremental and evolutionary development to use, in addition to the choice of single-step development.

The single-step-to-full-capability process exemplified by the traditional waterfall or sequential V-model (perhaps accompanied by early prototyping of high-risk features) is appropriate if the product's requirements are prespecifiable and have a low probability of significant change—and if there's no value or chance to deliver a partial product

Table 1

Incremental and evolutionary development decision table*

Type	Stable, prespecifiable requirements?	OK to wait for full system to be developed?	Need to wait for next-increment priorities?	Need to wait for next-increment enablers**?
Single step	Yes	Yes		
Prespecified sequential	Yes	No		
Evolutionary sequential	No	No	Yes	
Evolutionary overlapped	No	No	No	Yes
Evolutionary concurrent	No	No	No	No

** Source: Barry Boehm and Jo Ann Lane, used with permission.[10]*
*** Example enablers: technology maturity, external-system capabilities, and needed resources.*

capability. A good example would be the hardware for an Earth resources monitoring satellite whose altitude is too high for practical modification, or a decision to embody various standard mathematical software functions into a hardware chip for top performance. In general, though, software developed for human applications will undergo continuing evolution.[1]

The prespecified sequential process splits up the development to field an early initial operational capability and several preplanned product improvements (P3Is). It's best if the product's requirements are prespecifiable (again, perhaps after some early prototyping of high-risk features) and have a low probability of significant change (either because of stable subject matter or a short useful lifetime), and if waiting for development of the full system incurs a loss of important and deliverable early mission capabilities. A good example would be a well-understood and well-prioritized sequence of software upgrades that you could electronically upload for the high-altitude satellite's onboard Earth resources monitoring capabilities.

The evolutionary sequential process develops an initial operational capability and upgrades it on the basis of operational experience (as exemplified by agile methods[3]) or more plan-driven evolutionary methods.[7] It's best when there's a need to get operational feedback on an initial capability before defining and developing the next increment's content. A good example would be the software upgrades suggested by experiences with the satellite's payload, such as what kind of multispectral data collection and analysis capabilities are best for what kind of agriculture under what weather conditions.

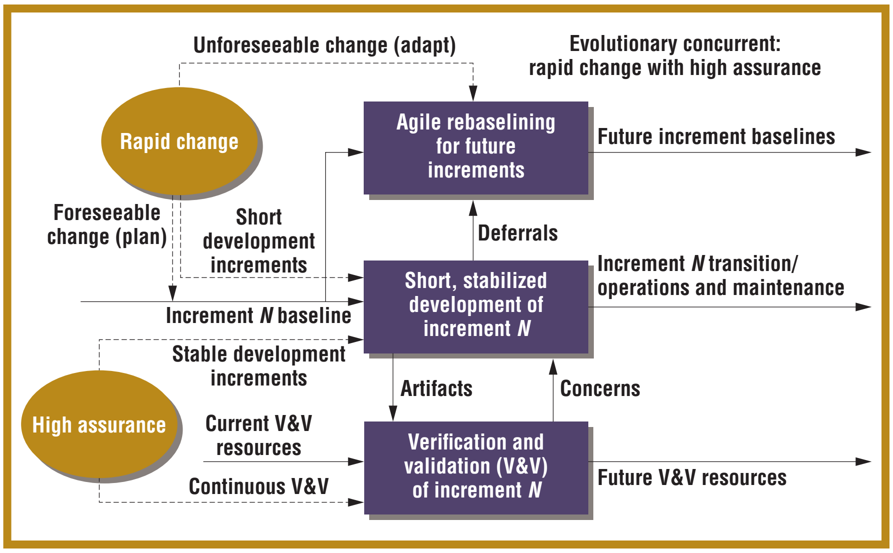

Unforeseeable change (adapt)
Evolutionary concurrent: rapid change with high assurance
Rapid change
Agile rebaselining for future increments
Future increment baselines
Foreseeable change (plan)
Short development increments
Deferrals
Increment N transition/ operations and maintenance
Short, stabilized development of increment N
Increment N baseline
Stable development increments
Artifacts
Concerns
High assurance
Current V&V resources
Continuous V&V
Verification and validation (V&V) of increment N
Future V&V resources

Figure 1. Evolutionary concurrent process. This process provides rapid change handling and high assurance. The solid lines represent inputs and outputs, and the dotted lines represent strategic implications (the foreseeable and unforeseeable changes also represent inputs). The ellipses represent goals, and the rectangles represent activities. (Source: Barry Boehm and Jo Ann Lane, used with permission.[10])

The evolutionary overlapped process defers the next increment until its needed capabilities are available and mature enough to be added. It's best when you don't need to wait for operational feedback, but might need to wait for next-increment enablers such as technology maturity, external system capabilities, or needed resources. A good example would be the need to wait for some agent-based satellite safety trend analysis and mission adaptation software to become predictably stable before incorporating it in a scheduled increment.

Figure 1 shows that the evolutionary concurrent process, as realized in the incremental commitment model, has a continuing team of systems engineers handling the change traffic and rebaselining the plans and specifications for the next increment, while keeping a development team stabilized for on-time, high-assurance delivery of the current increment and employing a concurrent verification and validation team to perform continuous defect detection to enable even higher-assurance levels.[8,9] A good example would be the satellite's ground-based mission control software's rebaselining to adapt to new COTS releases and continuing user requests for data processing upgrades. The solid lines in Figure 1 represent inputs and outputs, and the dotted lines represent strategic implications, although the foreseeable and unforeseeable changes also represent inputs. The ellipses represent goals, and the rectangles represent activities.

The satellite example shows that for the complex systems of the future, different parts of the system and its software might evolve in different ways, again indicating that there will be no one-size-fits-all process for software evolution. However, Table 1 can be quite helpful in determining which processes are the best fits for evolving each part of the system, and the three-team model in Figure 1 provides a way for projects to develop the challenging software-intensive systems of the future that will need both adaptability to rapid change and high levels of assurance.

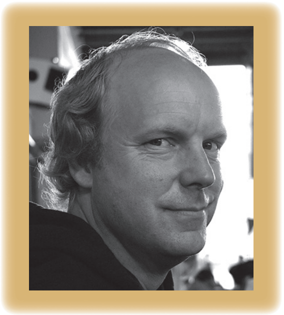
References

1. *Program Evolution—Processes of Software Change*, M. Lehman and L. Belady, eds., Academic Press, 1985.
2. M. Cusumano and D. Yoffee, *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft*, Free Press, 1998.
3. K. Beck, *Extreme Programming Explained*, Addison Wesley, 1999.
4. B. Boehm, "Some Future Trends and Implications for Systems and Software Engineering Processes," *Systems Engineering*, vol. 9, no. 1, 2006, pp. 1–19.
5. C. Larman, *Agile and Iterative Development*, Addison Wesley, 2004.
6. H.D. Benington, "Production of Large Computer Programs," *Proc. ONR Symp. Advanced Program Methods for Digital Computers*, Office of Naval Research, 1956, pp. 15–27.
7. T. Gilb, *Competitive Engineering*, Elsevier Butterworth Heinemann, 2005.
8. R.W. Pew and A.S. Mavor, *Human-System Integration in the System Development Process: A New Look*, National Academy Press, 2007.
9. B. Boehm and J. Lane, "Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering," *CrossTalk*, October 2007, pp. 4–9.
10. B. Boehm and J. Lane, *DoD Systems Engineering and Management Implications for Evolutionary Acquisition of Major Defense Systems*, tech. report, Systems Eng. Research Center, Univ. Southern California, 2010.

Barry Boehm is the TRW Professor in the University of Southern California's computer sciences and industrial and systems engineering departments. He's also the director of research of the DoD-Stevens-USC Systems Engineering Research Center, and the founding director emeritus of the USC Center for Systems and Software Engineering. Boehm is a fellow of the primary professional societies in computing (ACM), aerospace (AIAA), electronics (IEEE), and systems engineering (INCOSE), a member of the US National Academy of Engineering, and the 2010 recipient of the IEEE Simon Ramo Medal for exceptional achievement in systems engineering and systems science. Contact him at boehm@usc.edu.

The Inevitability of Evolution

Kent Beck



Economic stimulus money is hard at work here in southern Oregon. Crumbling freeway bridges are being replaced at an astonishing rate. While waiting for the flaggers to turn their signs, I've had plenty of chances to meditate on evolutionary design in action.

Design evolution is absolutely inevitable (see the "Recommended Reading" sidebar). Only failed projects are satisfied with their initial design. Success breeds change, change is unpredictable, and the design that seemed sufficient yesterday becomes today's bottleneck.

Somehow the story got out that if we were just good enough designers, we wouldn't have to change designs, we'd get it right the first time. My father's generation of programmers knew this was ridiculous. They were designing software to solve problems that hadn't ever been solved before. Design change was a natural part of the learning process. My daughter's generation of programmers likewise sees design evolution as a natural part of development. Businesses don't have time to wait for a perfect design, the kind you can imagine never having to change. They need feedback, and they need to respond. Design change again is a natural part of development. Perhaps the yearning for the "right" design is a generational anomaly, but it's nevertheless present in daily development.

The Challenge

What makes changing software hard? Cost, time, and risk.

Cost checks change. At the end of the life of every system, there's a point where changing it further becomes too expensive. The system is abandoned and (perhaps) another takes its place. These costs aren't linear. Small changes at the beginning of a project can have large effects on the cost of change later. Some systems, like the hardware on the satellite that Boehm mentions, can't change at all after a certain point. Balancing the desire to change the system are the costs to make that change and the effort required to enable those changes.

Timing plays with change. Some options might be easier to change but take longer to deliver. The cost of that time might exceed the value of the flexibility. Just to make the designer's job more interesting, though, sometimes the more flexible design actually reduces the time to deliver. The right foundational component can simplify the development of the rest of the system and produce a fertile resource for further changes.

The future has a way of mocking our attempts to anticipate change. You can anticipate that certain avenues are likely for future evolution. Sometimes you're right. Sometimes, though, the system proves to have unanticipated utility that invalidates those original assumptions. *This*, which we carefully made flexible, never changes while *that*, which seemed concrete, needs to morph a hundred ways.

These are the elements that designers need to understand to thoughtfully evolve software: cost, time, and risk. What makes software design and evolution such a joy (by which I mean maddening, frustrating, and confusing) is juggling these elements. There are seldom single right answers. There are, however, principles and experience that can inform the discussion (by which I mean fight) of design between designers.

Succession

One particular topic I haven't seen addressed adequately in software design was brought home to me during my enforced design meditations at freeway bridge construction sites. Upgrading the design of a bridge is a story. First, traffic is diverted to one side, and then the other bridge is demolished and replaced. Traffic is diverted to the new bridge, the other side is demolished and replaced, and traffic is restored. This is the simplest scenario. Some changes require much more elaborate sequences of changes to accomplish.

The engineers clearly spend substantial time and effort planning the succession of changes required to get to the design. It's not enough to imagine what kind of bridge you want to end up with, you also need to be able to get from here to there. Some potentially acceptable designs need to be discarded not because of any technical design fault, but simply because they violate the need for succession. You can't get there from here. Design needs to be informed as much by the disruption of the change process as it is by the value of the end result.

We're just beginning to understand software evolution. Just how frequently can I release design changes? Annually? Weekly? Hourly? How can I use automation and better technique to lower the cost and risk of changes? How can I maintain focus as I make large changes in small, safe steps over a long period of time, interleaved with feature development? My goal is to make microscale incremental change, what I call responsive design, a safe and inexpensive way to evolve designs. My Responsive Design Project includes a systematic study of succession (named after the permaculture principle).

Knowing both where to go with design evolution and how to get there in safe steps will make software development more effective.

Recommended Reading

- C. Alexander, *The Timeless Way of Building*, Oxford Univ. Press, 1979.
- E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice Hall, 1979.
- D. Thompson, *On Growth and Form*, Dover Publications, 1992.
- P. Whitefield, *Permaculture in a Nutshell*, Green Books, 1993.

Kent Beck is the founder and director of Three Rivers Institute. His contributions to software development include patterns for software, the rediscovery of test-first programming, the xUnit family of developer testing tools, and Extreme Programming. He currently divides his time between writing, programming, and coaching. Beck is the author/coauthor of *Implementation Patterns and Extreme Programming Explained: Embrace Change* (Addison-Wesley, 2005). Contact him at kent@threeriversinstitute.org.