There are a large number of life cycle process models. As discussed in the System Life Cycle Process Drivers and Choices article, these models fall into three major categories: (1) primarily pre-specified and sequential processes; (2) primarily evolutionary and concurrent processes (e.g., the rational unified process and various forms of the Vee and spiral models); and (3) primarily interpersonal and unconstrained processes (e.g., agile development, Scrum, extreme programming (XP), dynamic system development methods, and innovation-based processes).

This article discusses incremental and evolutionary development models (the second and third categories listed above) beyond variants of the Vee model. While there are a number of different models describing the project environment, the spiral model and the Vee Model have become the dominant approaches to visualizing the development process. Both the Vee and the spiral are useful models that emphasize different aspects of a system life cycle.

General implications of using iterative models for system design and development are discussed below. For a more specific understanding of how this life cycle model impacts systems engineering activities, please see the other knowledge areas (KAs) in Part 3. This article is focused on the use of iterative life cycle process models in systems engineering; however, because iterative process models are commonly used in software development, many of the examples below come from software projects. (See Systems Engineering and Software Engineering in Part 6 for more information on life cycle implications in software engineering.)

## Contents

- **1 Incremental and Evolutionary Development**
  - 1.1 Overview of the Incremental Approach
  - 1.2 Overview of the Evolutionary Approach
- **2 Iterative Software Development Process Models**
  - 2.1 Overview of Iterative-Development Process Models
- **3 The Incremental-Build Model**
  - 3.1 The Role of Prototyping in Software Development
  - 3.2 Life Cycle Sustainment of Software
  - 3.3 Retirement of Software
- **4 Primarily Evolutionary and Concurrent Processes: The Incremental Commitment Spiral Model**
  - 4.1 Overview of the Incremental Commitment Spiral Model
  - 4.2 Other Views of the Incremental Commitment Spiral Model
  - 4.3 Underlying ICSM Principles
  - 4.4 Model Experience to Date
Incremental and Evolutionary Development

Overview of the Incremental Approach

Incremental and iterative development (IID) methods have been in use since the 1960s (and perhaps earlier). They allow a project to provide an initial capability followed by successive deliveries to reach the desired system-of-interest (SoI).

The IID approach, shown in Figure 1, is used when:

- rapid exploration and implementation of part of the system is desired;
- the requirements are unclear from the beginning;
- funding is constrained;
- the customer wishes to hold the SoI open to the possibility of inserting new technology at a later time; and/or
- experimentation is required to develop successive prototype (glossary) versions.

The attributes that distinguish IID from the single-pass, plan-driven approach are velocity and adaptability.
Incremental development may also be “plan-driven” in nature if the requirements are known early on in the life cycle. The development of the functionality is performed incrementally to allow for insertion of the latest technology or for potential changes in needs or requirements. IID also imposes constraints. The example shown in Figure 2 uses the increments to develop high-risk subsystems (or components) early, but the system cannot function until all increments are complete.
Overview of the Evolutionary Approach

A specific IID methodology called evolutionary development is common in research and development (R&D) environments in both the government and commercial sector. Figure 3 illustrates this approach, which was used in the evolution of the high temperature tiles for the NASA Space Shuttle (Forsberg 1995). In the evolutionary approach, the end state of each phase of development is unknown, though the goal is for each phase to result in some sort of useful product.
The real-world development environment is complex and difficult to map because many different project cycles are underway simultaneously. Figure 4 shows the applied research era for the development of the space shuttle Orbiter and illustrates multi-levels of simultaneous development, trade-studies, and ultimately, implementation.

Iterative Software Development Process Models

Software is a flexible and malleable medium which facilitates iterative analysis, design, construction, verification, and validation to a greater degree than is usually possible for the purely physical
components of a system. Each repetition of an iterative development model adds material (code) to
the growing software base; the expanded code base is tested, reworked as necessary, and
demonstrated to satisfy the requirements for the baseline.

Process models for software development support iterative development on cycles of various
lengths. Table 1 lists three iterative software development models which are presented in more
detail below, as well as the aspects of software development that are emphasized by those models.

### Table 1. Primary Emphases of Three Iterative Software Development Models. (SEBoK
Original)

<table>
<thead>
<tr>
<th>Iterative Model</th>
<th>Emphasis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental-build</td>
<td>Iterative implementation-verification-validations-demonstration cycles</td>
</tr>
<tr>
<td>Spiral</td>
<td>Iterative risk-based analysis of alternative approaches and evaluation of outcomes</td>
</tr>
<tr>
<td>Agile</td>
<td>Iterative evolution of requirements and code</td>
</tr>
</tbody>
</table>

Please note that the information below is focused specifically on the utilization of different life cycle
models for software systems. In order to better understand the interactions between software
engineering (SwE) and systems engineering (SE), please see the Systems Engineering and Software
Engineering KA in Part 6.

**Overview of Iterative-Development Process Models**

Developing and modifying software involves creative processes that are subject to many external
and changeable forces. Long experience has shown that it is impossible to “get it right” the first
time, and that iterative development processes are preferable to linear, sequential development
process models, such as the well-known Waterfall model. In iterative development, each cycle of the
iteration subsumes the software of the previous iteration and adds new capabilities to the evolving
product to create an expanded version of the software. Iterative development processes provide the
following advantages:

- Continuous integration, verification, and validation of the evolving product;
- Frequent demonstrations of progress;
- Early detection of defects;
- Early warning of process problems;
- Systematic incorporation of the inevitable rework that occurs in software development; and
- Early delivery of subset capabilities (if desired).

Iterative development takes many forms in SwE, including the following:

- An incremental-build process, which is used to produce periodic (typically weekly) builds of
  increasing product capabilities;
- Agile development, which is used to closely involve a prototypical customer in an iterative process
  that may repeat on a daily basis; and
- The spiral model, which is used to confront and mitigate risk factors encountered in developing the
  successive versions of a product.

**The Incremental-Build Model**

The incremental-build model is a build-test-demonstrated model of iterative cycles in which frequent
demonstrations of progress, verification, and validation of work-to-date are emphasized. The model
is based on stable requirements and a software architectural specification. Each build adds new
capabilities to the incrementally growing product. The process ends when the final version is
verified, validated, demonstrated, and accepted by the customer.
Table 2 lists some partitioning criteria for incremental development into incremental build units of (typically) one calendar week each. The increments and the number of developers available to work on the project determine the number of features that can be included in each incremental build. This, in turn, determines the overall schedule.

Table 2. Some partitioning criteria for incremental builds (Fairley 2009). Reprinted with permission of the IEEE Computer Society and John Wiley & Sons Inc. All other rights are reserved by the copyright owner.

<table>
<thead>
<tr>
<th>Kind of System</th>
<th>Partitioning Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application package</td>
<td>Priority of features</td>
</tr>
<tr>
<td>Safety-critical systems</td>
<td>Safety features first; prioritized others follow</td>
</tr>
<tr>
<td>User-intensive systems</td>
<td>User interface first; prioritized others follow</td>
</tr>
<tr>
<td>System software</td>
<td>Kernel first; prioritized utilities follow</td>
</tr>
</tbody>
</table>

Figure 5 illustrates the details of the build-verify-validate-demonstrate cycles in the incremental build process. Each build includes detailed design, coding, integration, review, and testing done by the developers. In cases where code is to be reused without modification, some or all of an incremental build may consist of review, integration, and testing of the base code augmented with the reused code. It is important to note that development of an increment may result in reworking previous components developed for integration to fix defects.

Incremental verification, validation, and demonstration, as illustrated in Figure 5, overcome two of the major problems of a waterfall approach by:

- exposing problems early so they can be corrected as they occur; and
- incorporating minor in-scope changes to requirements that occur as a result of incremental demonstrations in subsequent builds.

Figure 5 also illustrates that it may be possible to overlap successive builds of the product. It may be possible, for example, to start a detailed design of the next version while the present version is being validated.

Three factors determine the degree of overlap that can be achieved:
1. Availability of personnel;
2. Adequate progress on the previous version; and
3. The risk of significant rework on the next overlapped build because of changes to the previous in-progress build.

The incremental build process generally works well with small teams, but can be scaled up for larger projects.

A significant advantage of an incremental build process is that features built first are verified, validated, and demonstrated most frequently because subsequent builds incorporate the features of the earlier iterations. In building the software to control a nuclear reactor, for example, the emergency shutdown software could be built first, as it would then be verified and validated in conjunction with the features of each successive build.

In summary, the incremental build model, like all iterative models, provides the advantages of continuous integration and validation of the evolving product, frequent demonstrations of progress, early warning of problems, early delivery of subset capabilities, and systematic incorporation of the inevitable rework that occurs in software development.

The Role of Prototyping in Software Development

In SwE, a prototype is a mock-up of the desired functionality of some part of the system. This is in contrast to physical systems, where a prototype is usually the first fully functional version of a system (Fairley 2009, 74).

In the past, incorporating prototype software into production systems has created many problems. Prototyping is a useful technique that should be employed as appropriate; however, prototyping is not a process model for software development. When building a software prototype, the knowledge gained through the development of the prototype is beneficial to the program; however, the prototype code may not be used in the deliverable version of the system. In many cases, it is more efficient and more effective to build the production code from scratch using the knowledge gained by prototyping than to re-engineer the existing code.

Life Cycle Sustainment of Software

Software, like all systems, requires sustainment efforts to enhance capabilities, adapt to new environments, and correct defects. The primary distinction for software is that sustainment efforts change the software; unlike physical entities, software components do not have to be replaced because of physical wear and tear. Changing the software requires re-verification and re-validation, which may involve extensive regression testing to determine that the change has the desired effect and has not altered other aspects of functionality or behavior.

Retirement of Software

Useful software is rarely retired; however, software that is useful often experiences many upgrades during its lifetime. A later version may bear little resemblance to the initial release. In some cases, software that ran in a former operational environment is executed on hardware emulators that provide a virtual machine on newer hardware. In other cases, a major enhancement may replace and rename an older version of the software, but the enhanced version provides all of the capabilities of the previous software in a compatible manner. Sometimes, however, a newer version of software may fail to provide compatibility with the older version, which necessitates other changes to a system.

Primarily Evolutionary and Concurrent Processes: The
Incremental Commitment Spiral Model

Overview of the Incremental Commitment Spiral Model

A view of the Incremental Commitment Spiral Model (ICSM) is shown in Figure 6.

In the ICSM, each spiral addresses requirements and solutions concurrently, rather than sequentially, as well as products and processes, hardware, software, human factors aspects, and business case analyses of alternative product configurations or product line investments. The stakeholders consider the risks and risk mitigation plans and decide on a course of action. If the risks are acceptable and covered by risk mitigation plans, the project proceeds into the next spiral.

The development spirals after the first development commitment review follow the three-team incremental development approach for achieving both agility and assurance shown and discussed in Figure 2, "Evolutionary-Concurrent Rapid Change Handling and High Assurance" of System Life Cycle Process Drivers and Choices.

Other Views of the Incremental Commitment Spiral Model

Figure 7 presents an updated view of the ICSM life cycle process recommended in the National Research Council Human-System Integration in the System Development Process study (Pew and Mavor 2007). It was called the Incremental Commitment Model (ICM) in the study. The ICSM builds on the strengths of current process models, such as early verification and validation concepts in the Vee model, concurrency concepts in the concurrent engineering model, lighter-weight concepts in the agile and lean models, risk-driven concepts in the spiral model, the phases and anchor points in the rational unified process (RUP) (Kruchten 1999; Boehm 1996), and recent extensions of the spiral model to address systems of systems (SoS) capability acquisition (Boehm and Lane 2007).
The top row of activities in Figure 7 indicates that a number of system aspects are being concurrently engineered at an increasing level of understanding, definition, and development. The most significant of these aspects are shown in Figure 8, an extension of a similar “hump diagram” view of concurrently engineered software activities developed as part of the RUP (Kruchten 1999).
As with the RUP version, the magnitude and shape of the levels of effort will be risk-driven and likely to vary from project to project. Figure 8 indicates that a great deal of concurrent activity occurs within and across the various ICSM phases, all of which need to be "synchronized and stabilized," a best-practice phrase taken from *Microsoft Secrets* (Cusumano and Selby 1996) to keep the project under control.

The review processes and use of independent experts are based on the highly successful AT&T Architecture Review Board procedures described in “Architecture Reviews: Practice and Experience” (Maranzano et al. 2005). Figure 9 shows the content of the feasibility evidence description. Showing feasibility of the concurrently developed elements helps synchronize and stabilize the concurrent activities.
The operations commitment review (OCR) is different in that it addresses the often-higher operational risks of fielding an inadequate system. In general, stakeholders will experience a two- to ten-fold increase in commitment level while going through the sequence of engineering certification review (ECR) to design certification review (DCR) milestones, but the increase in going from DCR to OCR can be much higher. These commitment levels are based on typical cost profiles across the various stages of the acquisition life cycle.

### Underlying ICSM Principles

ICSM has four underlying principles which must be followed:

1. Stakeholder value-based system definition and evolution;
2. Incremental commitment and accountability;
3. Concurrent system and software definition and development; and
4. Evidence and risk-based decision making.

### Model Experience to Date


### Agile and Lean Processes

According to the INCOSE Systems Engineering Handbook 3.2.2, “Project execution methods can be described on a continuum from 'adaptive' to 'predictive.' Agile methods exist on the 'adaptive' side of this continuum, which is not the same as saying that agile methods are 'unplanned' or 'undisciplined,'” (INCOSE 2011, 179). Agile development methods can be used to support iterative life cycle models, allowing flexibility over a linear process that better aligns with the planned life cycle for a system. They primarily emphasize the development and use of tacit interpersonal knowledge as compared to explicit documented knowledge, as evidenced in the four value propositions in the "Agile Manifesto":

*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value*
- **Individuals and interactions** over processes and tools;
- **Working software** over comprehensive documentation;
- **Customer collaboration** over contract negotiation; and
- **Responding to change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more. (Agile Alliance 2001)

Lean processes are often associated with agile methods, although they are more scalable and applicable to high-assurance systems. Below, some specific agile methods are presented, and the evolution and content of lean methods is discussed. Please see “Primary References”, “Additional References”, and the Lean Engineering article for more detail on specific agile and lean processes.

**Scrum**

Figure 10 shows an example of Scrum as an agile process flow. As with most other agile methods, Scrum uses the evolutionary sequential process shown in Table 1 (above) and described in Fixed-Requirements and Evolutionary Development Processes section in which systems capabilities are developed in short periods, usually around 30 days. The project then re-prioritizes its backlog of desired features and determines how many features the team (usually 10 people or less) can develop in the next 30 days.

Figure 10 also shows that once the features to be developed for the current Scrum have been expanded (usually in the form of informal stories) and allocated to the team members, the team establishes a daily rhythm of starting with a short meeting at which each team member presents a roughly one-minute summary describing progress since the last Scrum meeting, potential obstacles, and plans for the upcoming day.
Architected Agile Methods

Over the last decade, several organizations have been able to scale up agile methods by using two layers of ten-person Scrum teams. This involves, among other things, having each Scrum team’s daily meeting followed up by a daily meeting of the Scrum team leaders discussing up-front investments in evolving system architecture (Boehm et al. 2010). Figure 11 shows an example of the Architected Agile approach.
Agile Practices and Principles

As seen with the Scrum and architected agile methods, "generally-shared" principles are not necessarily "uniformly followed". However, there are some general practices and principles shared by most agile methods:

- The project team understands, respects, works, and behaves within a defined SE process;
- The project is executed as fast as possible with minimum down time or staff diversion during the project and the critical path is managed;
- All key players are physically or electronically collocated, and "notebooks" are considered team property available to all;
- Baseline management and change control are achieved by formal, oral agreements based on “make a promise—keep a promise” discipline. Participants hold each other accountable;
- Opportunity exploration and risk reduction are accomplished by expert consultation and rapid model verification coupled with close customer collaboration;
- Software development is done in a rapid development environment while hardware is developed in a multi-disciplined model shop; and
- A culture of constructive confrontation pervades the project organization. The team takes ownership for success; it is never “someone else’s responsibility.”

Agile development principles (adapted for SE) are as follows (adapted from Principles behind the Agile Manifesto (Beedle et al. 2009)):

1. First, satisfy the customer through early and continuous delivery of valuable software (and other system elements).
2. Welcome changing requirements, even late in development; agile processes harness change for the customer’s competitive advantage.
3. Deliver working software (and other system elements) frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business personnel and developers must work together daily throughout the project.
5. Build projects around motivated individuals; give them the environment, support their needs, and trust them to get the job done.
6. The most efficient and effective method of conveying information is face-to-face conversation.
7. Working software (and other system elements) is the primary measure of progress.
8. Agile processes promote sustainable development; the sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.

A team should reflect on how to become more effective at regular intervals and then tune and adjust its behavior accordingly. This self-reflection is a critical aspect for projects that implement agile processes.

Lean Systems Engineering and Development

Origins

As the manufacturing of consumer products such as automobiles became more diversified, traditional pre-planned mass-production approaches had increasing problems with quality and adaptability. Lean manufacturing systems such as the Toyota Production System (TPS) (Ohno 1988) were much better suited to accommodate diversity, to improve quality, and to support just-in-time manufacturing that could rapidly adapt to changing demand patterns without having to carry large, expensive inventories.
Much of this transformation was stimulated by the work of W. Edwards Deming, whose Total Quality Management (TQM) approach shifted responsibility for quality and productivity from planners and inspectors to the production workers who were closer to the real processes (Deming 1982). Deming’s approach involved everyone in the manufacturing organization in seeking continuous process improvement, or "Kaizen".

Some of the TQM techniques, such as statistical process control and repeatability, were more suited to repetitive manufacturing processes than to knowledge work such as systems engineering (SE) and software engineering (SwE). Others, such as early error elimination, waste elimination, workflow stabilization, and Kaizen, were equally applicable to knowledge work. Led by Watts Humphrey, TQM became the focus for the Software Capability Maturity Model (Humphrey 1987; Paulk et al. 1994) and the CMM-Integrated or CMMI, which extended its scope to include systems engineering (Chrissis et al. 2003). One significant change was the redefinition of Maturity Level 2 from "Repeatable" to "Managed".

The Massachusetts Institute of Technology (MIT) conducted studies of the TPS, which produced a similar approach that was called the "Lean Production System" (Krafcik 1988; Womack et al. 1990). Subsequent development of "lean thinking" and related work at MIT led to the Air Force-sponsored Lean Aerospace Initiative (now called the Lean Advancement Initiative), which applied lean thinking to SE (Murman 2003, Womack-Jones 2003). Concurrently, lean ideas were used to strengthen the scalability and dependability aspects of agile methods for software (Poppendieck 2003; Larman-Vodde 2009). The Kanban flow-oriented approach has been successfully applied to software development (Anderson 2010).

Principles

Each of these efforts has developed a similar but different set of Lean principles. For systems engineering, the current best source is Lean for Systems Engineering, the product of several years’ work by the INCOSE Lean SE working group (Oppenheim 2011). It is organized into six principles, each of which is elaborated into a set of lean enabler and sub-enabler patterns for satisfying the principle:

1. **Value.** Guide the project by determining the value propositions of the customers and other key stakeholders. Keep them involved and manage changes in their value propositions.

2. **Map the Value Stream (Plan the Program).** This includes thorough requirements specification, the concurrent exploration of trade spaces among the value propositions, COTS evaluation, and technology maturity assessment, resulting in a full project plan and set of requirements.

3. **Flow.** Focus on the project’s critical path activities to avoid expensive work stoppages, including coordination with external suppliers.

4. **Pull.** Pull the next tasks to be done based on prioritized needs and dependencies. If a need for the task can’t be found, reject it as waste.

5. **Perfection.** Apply continuous process improvement to approach perfection. Drive defects out early to get the system Right The First #Time, vs. fixing them during inspection and test. Find and fix root causes rather than symptoms.

6. **Respect for People.** Flow down responsibility, authority, and accountability to all personnel. Nurture a learning environment. Treat people as the organization’s most valued assets. (Oppenheim 2011)

These lean SE principles are highly similar to the four underlying incremental commitment spiral model principles.

- **Principle 1: Stakeholder value-based system definition and evolution**, addresses the lean SE principles of value, value stream mapping, and respect for people (developers are success-critical stakeholders in the ICSM).
Principle 2: Incremental commitment and accountability, partly addresses the pull principle, and also addresses respect for people (who are accountable for their commitments).

Principle 3: Concurrent system and software definition and development, partly addresses both value stream mapping and flow.

Principle 4: Evidence and risk-based decision making, uses evidence of achievability as its measure of success. Overall, the ICSM principles are somewhat light on continuous process improvement, and the lean SE principles are somewhat insensitive to requirements emergence in advocating a full pre-specified project plan and set of requirements.

See Lean Engineering for more information.

References

Works Cited


**Primary References**


Additional References


