

Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion

Creating and maintaining a consistent set of specifications that result in software solutions that match customer's needs is always a challenge. A method is described that breaks the software life cycle into smaller chunks so that customer input is allowed throughout the process.

by Todd Cotton

Fusion provides a thorough and consistent set of models for translating the specification of customer needs into a well-structured software solution. For reasonably small projects, the sequential steps of Fusion map well into the sequential software life cycle commonly known as the waterfall life cycle. For larger projects, those representative of most commercial and IT software projects today, an incremental life cycle such as Evolutionary Development provides a much better structure for managing the risks inherent in complex software development. This paper introduces Evolutionary Fusion, the combination of Fusion, with its advantages provided by object orientation, and the key Evolutionary Development concepts of early, frequent iteration, strong customer orientation, and dynamic plans and processes.

Although based on the best of other object-oriented methods, Fusion is a relatively new method. The Fusion text¹ was published in October 1994, and as a member of the Hewlett-Packard software development community, the author was exposed to preliminary work by Derek Coleman and his team earlier in 1993. The response from the first few teams to apply Fusion to their work was extremely encouraging. As members of the Software Initiative, an internal consulting group focused on further extending Hewlett-Packard's software development competencies, the author and his colleagues have helped facilitate the rapid adoption of Fusion within Hewlett-Packard. Fusion is now used in nearly every part of Hewlett-Packard, contributing to products and services as diverse as network protocol drivers, real-time instrument firmware, printer drivers, internal information systems, and even medical imaging and management products. This paper is based on these collected experiences.

To simplify the presentation of concepts, the paper first discusses experiences gained working with small, collocated development teams. Later sections deal with the extensions that have been made to scale Evolutionary Fusion up for larger teams split across geographic boundaries. See the Sidebar: **"What is Fusion?"** for an explanation of this software development method.

Need for an Alternative to the Waterfall Life Cycle

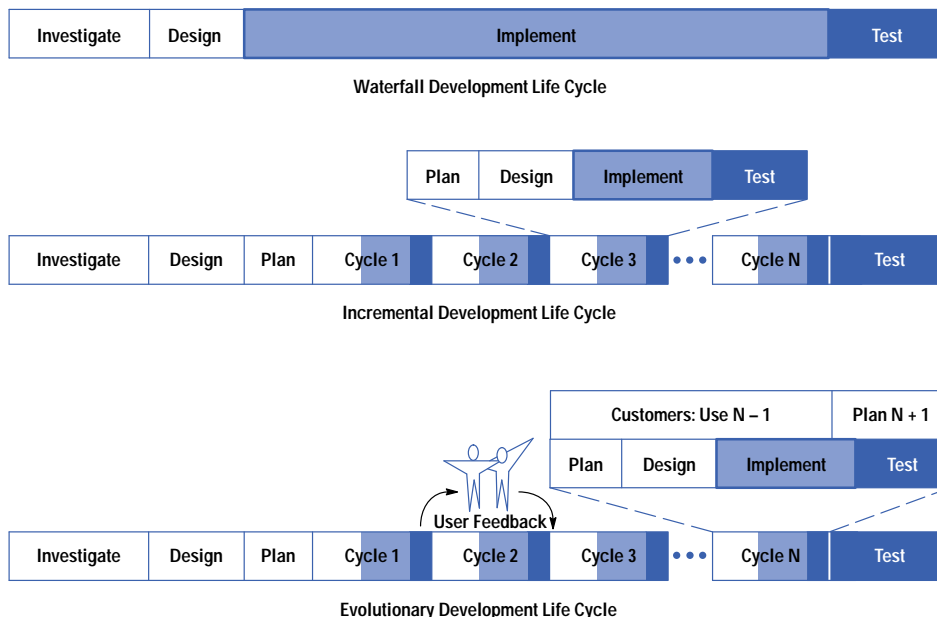
The traditional waterfall life cycle for software development has served software developers well. By breaking software projects up into several large sequential phases—typically an investigation or definition phase, a design phase, an implementation phase, and a test phase—project teams could move forward with confidence. System requirements were captured through significant customer interaction during the definition phase. Once these requirements were complete, the other phases could progress with focus and efficiency since few if any changes to the specification would be allowed. With limited competition and with products that would remain viable for years, it was safe to assume that the system requirements captured many months or even years earlier would still be accurate. Unfortunately, this is no longer the environment in which software is developed.

Today, our ability as software engineers and project managers to accommodate all risks and accurately schedule projects that may include tens or even hundreds of engineers over several years of development is seriously challenged. Customers' needs, competitive products, and even the development tools we use can change as often as every few months. We have at least two choices. We can try to further refine our estimation and scheduling skills, fixing more parameters of our projects at very early stages of knowledge and experience, or we can look for an alternative development life cycle that better supports the dynamic and complex nature of our business today.

* Adapted from *Object-Oriented Development at Work: Fusion in the Real World*, Ruth Malan, Reed Letsinger, and Derek Coleman, Editors, Hewlett-Packard Professional Books, Published by Prentice Hall PTR, Prentice-Hall Inc., 1996, ISBN 0-13-243148-3. All rights reserved.

One alternative to the waterfall life cycle is Barry Boehm's² spiral life cycle. Actually more of a meta life cycle, the spiral life cycle can be instantiated or "unwrapped" in a number of ways. One instantiation is the iterative life cycle, an approach advocated by industry-leading OO (object-oriented) methodologists such as Jim Rumbaugh³ and Grady Booch.⁴ An iterative life cycle replaces the monolithic implementation phase of the waterfall life cycle with much smaller implementation cycles (Fig. 1) that start by building a very small piece of the overall functionality of the system and then add to this base over time until a complete system is delivered. Incremental development "determines user needs and defines the system requirements, then performs the rest of the development in a sequence of builds."⁵

Fig. 1. Different models of the software development life cycle.



Another instantiation of the spiral life cycle is Evolutionary Development, proposed by Tom Gilb.⁶ Evolutionary Development adds to the iterative life cycle a much stronger customer orientation that is implemented through an explicit customer feedback loop. Evolutionary Development "differs from the incremental strategy in acknowledging that the user need is not fully understood and all requirements cannot be defined up front ... user needs and system requirements are partially defined up front, then are refined in each succeeding build."⁵ The Evolutionary Development life cycle has been used successfully within Hewlett-Packard since 1985 and was the natural choice to combine with Fusion when we needed an alternative to the waterfall life cycle.

Evolutionary Development

Evolutionary Development (EVO) is a software development method and life cycle that replaces traditional waterfall development with small, incremental product releases or builds, frequent delivery of the product to users for feedback, and dynamic planning that can be modified in response to this feedback. As originally presented by Tom Gilb, the method had the following key attributes:

1. Multiobjective-driven
2. Early, frequent iteration
3. Complete analysis, design, build, and test in each step
4. User orientation
5. Systems approach, not merely algorithm orientation
6. Open-ended basic systems architecture
7. Result orientation, not software development process orientation.

Using EVO, a product development team divides the project into small chunks. Ideally, each chunk is less than 5% of the overall effort. The chunks are then ordered so that the most useful and easiest features are implemented first and some useful subset of the overall product can be delivered every one to four weeks. Within each EVO cycle, the software is designed, coded, tested, and then delivered to users. The users give feedback on the product and the team responds, often by changing the product, plans, or process. These cycles continue until the product is shipped.

EVO is thus characterized by early and frequent iteration, starting with an initial implementation and followed by frequent cycles that are short in duration and small in content. Drawing on ongoing user feedback, planning, design, coding, and testing are completed for each cycle, and each release or build meets a minimum quality standard. This method offers opportunities to optimize results by modifying the plan, product, or process at each cycle. The basic product concept or value proposition, however, does not change.

At Hewlett-Packard, we have found that it is possible to relax some of Gilb's ideas regarding EVO.^{6*} In particular, it is not absolutely necessary to deliver the product to real customers with customer-ready documentation, training, support, and so on, to benefit from EVO. For instance, customers participating in the feedback loop change during the development process. Results from the early cycles of development are typically given to other team members or other project teams for feedback. Less sensitive to the lack of complete documentation and training materials, they can still give valuable feedback. Results from the next several cycles are shared with surrogate customers represented by members of the broader Hewlett-Packard community. The goal is still to get the product into the hands of actual customers as early as possible.

There are two other variations to Tom Gilb's guidelines that we have found useful within Hewlett-Packard. First, the guideline that each cycle represent less than 5% of the overall implementation effort has translated into cycle lengths of one to four weeks, with two weeks being the most common. Second, ordering the content of the cycles is used within Hewlett-Packard as a key risk-management opportunity. Instead of implementing the most useful and easiest features first, many development teams choose to implement in an order that gives the earliest insight into key areas of risk for the project, such as performance, ease of use, or managing dependencies with other teams.

Benefits of EVO

The teams within Hewlett-Packard that have adopted Evolutionary Development as a project life cycle have done so with explicit benefits in mind. In addition to better meeting customer needs or hitting market windows, there have been a number of unexpected benefits, such as increased productivity and reduced risk, even the risks associated with changing the development process.

Better Match to Customer Need and Market Requirements. The explicit customer feedback loop of Evolutionary Development results in the delivery of products that better meet the customers' need. The waterfall life cycle provides an investigation or definition phase for eliciting customer needs through focus groups and storyboards, but it does not provide a mechanism for continual validation and refinement of customer needs throughout the long implementation phase. Many customers find it difficult to articulate the full range of what they want from a product until they have actually used the product. Their needs and expectations evolve as they gain experience with the product. Evolutionary Development addresses this by incorporating customer feedback early and often during the implementation phase. The small implementation cycles allow the development team to respond to customer feedback by modifying the plans for future implementation cycles. Existing functionality can be changed, while planned functionality can be redefined.

One Hewlett-Packard project used a variation of Evolutionary Development that also included an evolutionary approach to product definition.⁷ During the first month, the development team worked from static visual designs to code a prototype. In focus group meetings, the team discussed users' needs and the potential features of the product and then demonstrated their prototype. The focus groups expressed strong support for the product concept, so the project proceeded to a second phase of focus group testing incorporating the feedback from the first phase. Once the feedback from the second round of focus groups was incorporated, the feature set was established and the product definition completed.

Implementation consisted of four-to-six-week cycles, with software delivered to customers for use at the end of each cycle. The entire development effort spanned ten months from definition to product release. The result was a world-class product that has won many awards and has been easy to support.

Hitting Market Windows. To enhance productivity, many large software projects divide their tasks into independent subsets that can be developed in parallel. With few dependencies between subteams, each team can progress at its own pace. The risk in this approach is the significant effort that must be invested to bring all the work of these subteams together for final integration and system test. When issues are uncovered at this late stage of development, few options are available to the development team. It is difficult if not impossible to prune functionality in a low-risk manner when market windows, technology, or competition change. The only option open to the team is to continue on, finding and removing defects as quickly and as efficiently as possible (see Fig. 2).

With an EVO approach, the team has greater flexibility as the market window approaches. Two attributes of EVO contribute to this flexibility. First, the sequencing of functionality during the implementation phase is such that "must have" features are completed as early as possible, while the "high want" features are delayed until the later EVO cycles. Second, since each cycle of the implementation phase is expected to generate a "complete" release, much of the integration testing has already been completed. Any of the last several EVO cycles can become release candidates after a final round of integration and system test. When an earlier-than-planned release is needed, the last one or two EVO cycles can be skipped as long as a viable product already exists. If a limited number of key features are still needed, an additional EVO cycle or two can be defined and implemented as illustrated in Fig. 3.

* See also Article 5.

Fig. 2. Hitting market windows with a waterfall life cycle.

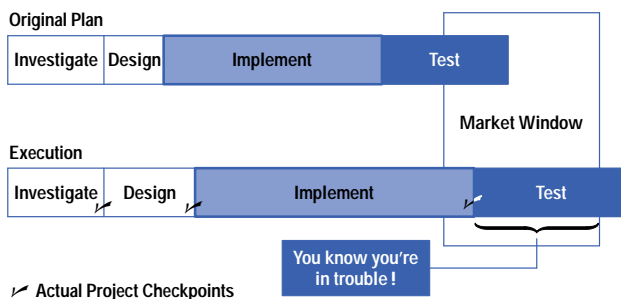
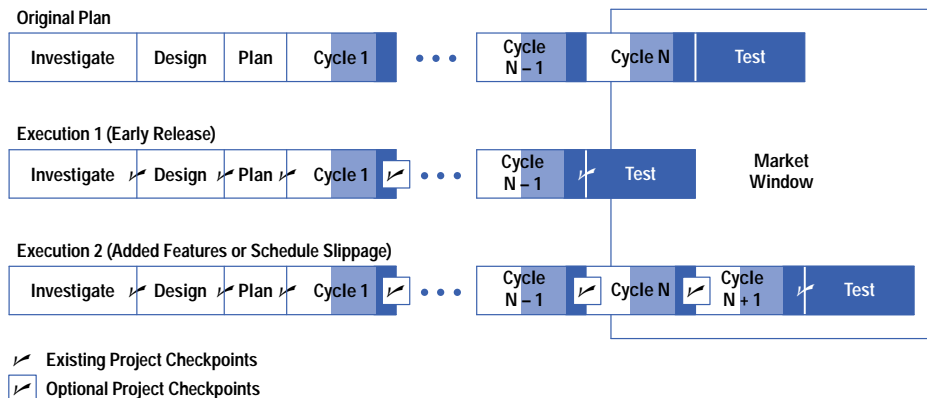


Fig. 3. Hitting market windows with an evolutionary life cycle.



Engineer Motivation and Productivity. Some of the gains in productivity seen by project teams using EVO have been attributed to higher engineer motivation. The long implementation phase of the waterfall life cycle is often characterized by large variations in engineer motivation. It is difficult for engineers to maintain peak productivity when it may be months before they can integrate their work with that of others to see real results. Engineer motivation can take an even greater hit when the tyranny of the release date prohibits all but the most trivial responses to customer feedback received during the final stages of system test.

EVO has led to higher productivity for development teams by maintaining a higher level of motivation throughout the implementation phase. The short implementation cycles keep everyone focused on a small set of features and tasks. The explicit customer feedback loop and the small implementation cycles also allow the development team more opportunity to respond to customer feedback and thereby deliver a product that they know represents their best work.

Quality Control. Although software development is in many ways a manufacturing process, software development teams have struggled to apply quality improvement processes such as Total Quality Control (TQC). Unlike the manufacturing organizations that can measure and refine processes with cycle times of hours, minutes, and even seconds, the waterfall life cycle gave cycle times of months or years before the software development process repeated. With EVO, the software implementation cycle is dramatically reduced and repeated multiple times for each project. All parameters of the implementation process are now available for review and improvement. The impact of changes in processes and tools can be measured and refined throughout the implementation phase.

Reducing Risk when Changing the Development Process. Many teams experience considerable anxiety as they make the transition to an object-oriented approach to development. The transition to OO usually entails a number of changes in the way a software engineer works. There are new analysis and design models to apply, new notations to master, and new, occasionally eccentric, tools and compilers to learn. There is also valid concern about adopting a new method at the beginning of the development process. Few teams are willing to make a full commitment to a new method when they have little experience with it. There may even be organizational changes anticipated if the organization is looking for large-scale productivity gains through formalized reuse.

Development teams and managers want some way to manage the risks associated with making so many simultaneous changes to their development environment. EVO can help manage the risks. The repeating cycles during the implementation phase provide for continual review and refinement of each parameter of the development environment. Any aspect of the development environment can be dropped, modified, or strengthened to provide the maximum benefit to the team.

Costs of EVO

Adopting Evolutionary Development is not without cost. It presents a new paradigm for the project manager to follow when decomposing and planning the project, and it requires more explicit, organized decision making than many managers and teams are accustomed to.

In traditional projects, subsystems or code modules are identified and then parceled out for implementation. As a result, planning and staffing of large projects were driven by the structure of the system and not by its intended use. In contrast, Evolutionary Development focuses on the intended use of the system. The functionality to be delivered in a given cycle is determined first. It is common practice to implement only those portions of subsystems or modules that support that functionality during that cycle. This approach to building a work breakdown structure presents a new paradigm to the project manager and the development team. Subsystem and module completion cannot be used for intermediate milestone definition because their full functionality is not in place until the end of the project. The time needed to adopt this new paradigm and create an initial plan can be a major barrier for some project teams.

Many development teams lack a well-defined, efficient decision-making process. Often they make decisions implicitly within a limited context, risking the compromise of the broader project goals and slowing progress dramatically. Evolutionary Development forces many decisions to be made explicitly in an organized way, because feedback on the product is received regularly and schedules must be updated for each implementation cycle.

The continual stream of information that the project team receives must be translated into three categories of decisions: changes to the product as it is currently implemented, changes to the plan that will further the product implementation, and changes to the development process used to develop the product. Fortunately, because of EVO's short cycle time, teams have many opportunities to assess the results of decisions and adjust accordingly.

Evolutionary Fusion

Fusion and Evolutionary Development are complementary. One of the primary assumptions of EVO is that one can decompose the functionality of a project into small manageable chunks. It is also expected that these chunks will provide some measurable value to the intended user and can thus be given to the user for feedback. Fusion provides the method of decomposition. At the highest level, Fusion decomposes the functionality of a system into use scenarios. Use scenarios are defined from the perspective of a user or agent of the system and are expected to capture a use of the system that provides some value to the agent.

EVO also presupposes that an architecture capable of accommodating all the expected functionality of the system can be defined prior to implementation. This architecture must be flexible enough to accommodate new or redefined functionality resulting from customer feedback. Fusion helps create this flexible architecture. The object model provides an architecture that encapsulates common functionality into classes and provides flexibility and extensibility through generalization and specialization. Fusion also accommodates large-scale change through the well-defined linkages between models. If necessary, changes to functionality can be rolled all the way up to the use scenarios and then cascaded back down through the appropriate analysis and design models, replacing guesswork in assessing the impact of a change with a more systematic approach.

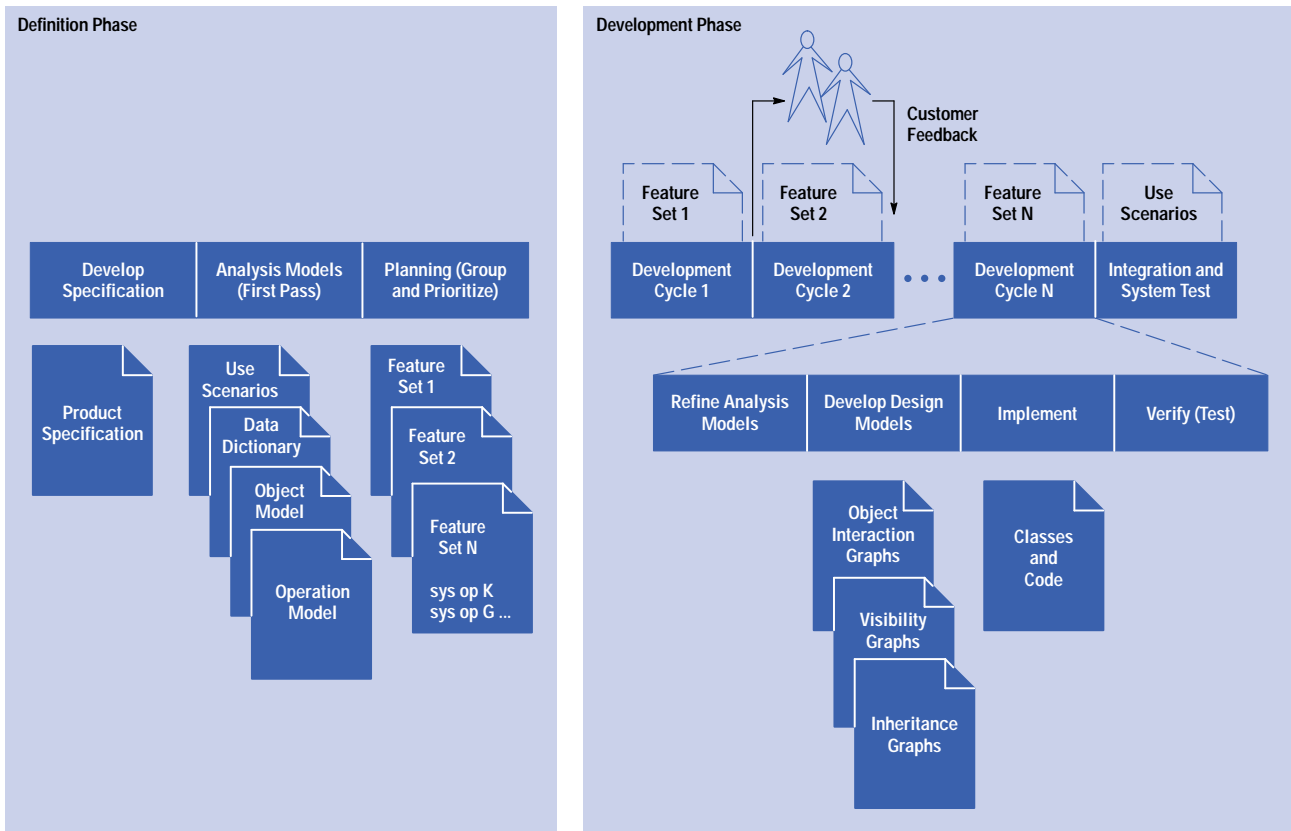
Evolutionary Fusion divides a project into two major phases: the definition phase and the development phase (Fig. 4). During the definition phase, a project's functionality is specified and its viability as a product or system is first estimated. The Fusion analysis models play a key role in this phase. The use scenarios serve to remodel the specification document, checking it for clarity and completeness. They can also be reviewed with customers to validate the development team's understanding of customer needs. The object model captures the initial architecture for the system and provides additional checks of the specification. The data dictionary captures the team's emerging common vocabulary and understanding of the problem domain. The operation model, through its system operation descriptions, gives an indication of the size and complexity of the project. This information is critical for estimating resource needs and developing the initial plan for the development phase.

The second phase is the development phase, in which code is incrementally designed, implemented, and tested to meet the specification. Each development cycle follows the same pattern. First, the analysis models are reviewed for completeness with respect to the functionality to be implemented during that cycle. Next, the Fusion design models are created or updated to support the functionality. And finally, the code is written and regression tests executed against the code. In parallel with the development activities of the team, selected users or customers of the system are working with and providing feedback on the release from the previous cycle. This feedback is used to adjust the plan for the following cycles. To complete the development phase, a final round of integration and system testing is done. The next two sections discuss these two phases in more detail.

Definition Phase

The definition phase is best characterized as a period of significant communication and thought. Communication must occur between all members of the project team to make sure that everyone shares a common understanding of the project's goals. Thought must be put into the specification document to make sure that it is complete and unambiguous and that it meets the requirements. Communication must occur between the development team and the intended users of the system to

Fig. 4. Evolutionary Fusion life cycle.



ensure that the system, at least as it can be specified on paper during this early stage of the project, will meet their needs. Thought must go into defining an architecture capable of supporting the intended functionality of the full system. The goal is to identify and resolve as many issues as possible during this phase. Specification errors that are not resolved during this phase can be extremely costly to repair later.

Our experience has shown that the Fusion analysis models are ideal for stimulating the thought and supporting the communication that must occur during the definition phase.

Analysis Models—First Pass

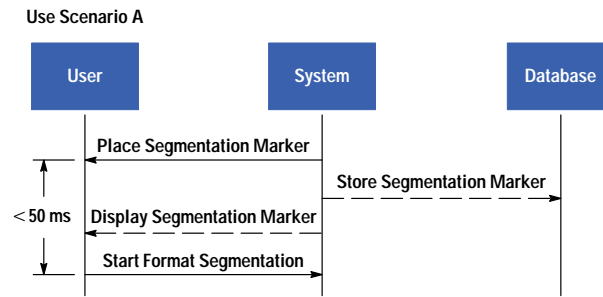
Like Fusion, Evolutionary Fusion requires some form of system specification as a starting point, and just about any level of detail in the system specification will do. When the specification is at a high level, the analysis models serve to identify large numbers of issues and questions that need to be resolved before development can begin. When the specification is at a more detailed level, the analysis models serve to remodel and recapture high-level structure and functionality that may be lost in the detail. We have yet to define what level of detail in the system specification yields the most efficient definition phase for Evolutionary Fusion. Regardless of the level of specification detail, the analysis models provide the beginning of a common vocabulary and understanding of the problem domain that will serve the team well throughout the project.

The most critical component of the system specification is the *value proposition*.⁸ The value proposition clearly articulates why the intended customer of the system will choose to use it over the other options available. The functionality defined in the specification is the development team's initial best estimate as to how to deliver that value proposition. There are usually countless other ways to deliver it. The explicit customer feedback loop of Evolutionary Fusion will validate the best estimate over time and will suggest better ways to deliver the value proposition. The value proposition itself should remain constant throughout the entire development process. If the value proposition changes during the development phase, it will be quite difficult for the team to make all the modifications necessary to implement a new one and still end up with a coherent set of product features.

Use Scenarios

The first analysis model to be created is the set of use scenarios. To provide some structure for this activity, it is useful to first generate a list of all the agents that exist in the system's environment. It can often be a challenge to decide what constitutes an agent. For example, the file system provided by the operating system is clearly part of any system's environment. It can be expected to provide services to and make demands on the system being defined. Representing the file system as an agent does not add any additional clarity to the team's understanding of the system under definition. However,

Fig. 5. Use scenario with time-constraint annotation.



representing specific files as agents, such as configuration files, legacy databases, or data input files, does add clarity. In one project, it was useful to model, as an agent, a critical data input file generated externally to the system. A general rule of thumb is that an agent must add to the understanding of the system if it is to be included at this early stage.

Once the list of agents is complete, each agent can be examined with respect to the demands it will make on the system. These demands are captured as use scenarios. As with defining agents, determining an appropriate level of granularity for the use scenarios can be a challenge. Another rule of thumb is that use scenarios should provide complete chunks of value from the perspective of the agent. In the project mentioned above, the system was modeled as providing value to the input file by accepting records of data from the file and translating those records into a format that could be used by the rest of the system. This approach will help avoid the issue of trying to keep all use scenarios at the same level of granularity. It is the agent that defines the appropriate level of granularity, not the system as a whole.

Once the use scenarios have been specified, each is diagrammed to decompose it further into discrete system operations and events. It is also useful to annotate in the margins of the use scenario diagram any time constraints that may exist (see Fig. 5). For systems of reasonable size, it is difficult to define a correct set of use scenarios on the first try. Building the use scenarios is itself an iterative process of refinement.

Object Model

As Ould⁹ states in his text on software engineering strategies,

“The success of the incremental delivery approach rests on the ability of the designer to create—from the start—an architecture that can support the full functionality of the system so that there is not a point during the sequence of deliveries where the addition of the next increment of functionality requires a massive re-engineering of the system at the architectural level (p. 59).”

The Fusion object model, the next analysis model to be created, serves as that architecture.

Once the use scenarios are complete, the development team has a much clearer understanding of the demands that will be placed on the system. The use scenarios are an excellent source of information for building the object model. The use scenario diagrams can be stepped through, making sure that analysis classes exist to support the need of each system operation. It is also quite common that building the object model will generate further refinements and improvements to the use scenarios.

Operation Model

The last analysis model to be created during the definition phase is the operation model. It documents in a declarative fashion the change in the state of the system as it responds to a system operation. Each system operation is described using only terms from the use scenarios, object model, and data dictionary.

A complete specification of the system exists when the operation model is completed. The use scenarios capture the intended uses of the system from the agents' point of view. The object model captures the high-level architecture of the system. The operation model documents the effect that each system operation has on the system. The creation of each model has stimulated the thought necessary to identify and resolve issues, while the notation for each model establishes a common communication format for the team.

Managing the Analysis Process

An appropriate question to ask at this point is how much time should be invested in making a first pass at the analysis models. Although there is no formula that we can offer for Evolutionary Fusion, the application of a progress measurement technique used by many development teams during implementation works surprisingly well at this early stage of development. During the integration and system test phase, many teams compare the rate at which defects are being identified to the rate at which defects are being isolated and repaired. In the early part of this phase, the rate of defect identification exceeds the rate of defect repair. At some later point in this phase, the rate of repair exceeds the rate of

identification, and estimates can be made on when the desired defect density will be reached and the product can be released.

A similar approach can be used to track progress during the creation of the analysis models in Evolutionary Fusion's definition phase. Any issue identified during the creation of the analysis models can be considered a potential defect in the specification of the system. As with testing code, the initial attempts to build the analysis models will generate a large number of potential issues, or defects. As the creation of the analysis models progresses, fewer and fewer issues, or defects, will be found. Once the rate of resolving, or repairing, these issues exceeds the rate of finding new issues, a completion date for the first pass at the analysis models can be estimated.

An additional parameter often assigned to defects is a classification that represents the severity of the defect. Few systems are shipped with known defects that can cause unrecoverable data loss, but many are shipped with known defects that have only limited impact on the system's use. It can be helpful to apply a similar classification scheme to the issues found during analysis.¹⁰ Many issues identified will be of such impact that they must be resolved before moving on to the development phase. Other issues will be of lesser impact and, as such, resolution can be delayed until the development phase. There is also a third class of issues that relates directly to design or implementation. These must be reclassified as design or implementation issues and marked for resolution during that phase.

There is an expectation that a team must complete all the analysis phase models before moving on to implementation. Our experience has shown that this is not the case. It is only necessary to complete a high-level view of the complete system and to resolve the critical and serious "defects" that have been logged against the analysis models. This approach can also help teams avoid "analysis paralysis," the malady that afflicts many teams when they try to resolve every known issue before moving on to design and implementation. The analysis models will be revisited as the first step of each implementation cycle, so further additions and refinements can be made then.

It is difficult to accurately estimate the length of the analysis phase, especially if it is the team's first use of object technology. Fortunately, using the approach described here can provide early indication of progress so that resources can be managed accordingly.

Building the Plan

The last task of the Evolutionary Fusion definition phase is to plan the next phase, development. This task consists of three major steps: assigning ownership for the key roles that must be played during this phase, defining the standard EVO cycle, and determining the sequence in which functionality will be developed.¹¹

Key Roles. For the development phase to progress in a smooth and efficient manner, it is helpful to define and assign ownership for three key roles: project manager, technical lead, and user liaison. On large project teams, these roles may be shared by more than one person. On smaller project teams, a person may play more than one role.

Project manager: Many aspects of the project manager's role become even more critical with Evolutionary Development. The project manager must work with the marketing team and the customers to establish the project's value proposition, identify key project risks, document all commitments and dependencies, and articulate how Evolutionary Development will contribute to the project's success. Agreement on the value proposition is critical, as it will help keep the decision-making process focused. The key project risks will be used to sequence the implementation so that these risks can be characterized and addressed as early as possible. The commitments and dependencies will also be a key consideration when sequencing the implementation cycles. It is also important that the project manager solicit and address any concerns that the project team has with the Evolutionary Development approach.

The project manager must also define and manage the decision-making process. Although this is often an implicit task of the project manager, the large amount of information and the increased number of decisions that must be made using Evolutionary Fusion require that this process be made explicit. Based on the kinds of changes anticipated during the project, the project manager must consider how information will be gathered, how decisions will be made, and how decisions will be communicated. With very short development cycles, delayed decisions can slow progress dramatically.

Working with the technical lead, the project manager may also decide to include explicit design cycles in the schedule. For software architectures and designs that are expected to survive many years, supporting multiple releases or even multiple product lines, it is important to invest in the evolution of the architecture. As the development phase progresses, certain isolated decisions that compromise some aspect of the architecture will be made. There will also be new insights into the architecture and its robustness that could not have been anticipated during the definition phase. Design cycles dedicated to the architecture will deliver no new functionality for the user. By including tasks such as architecture refinement, design development, and design inspections, these cycles will deliver to future EVO cycles an architecture that is better equipped to meet the demands that will be placed on it.

Technical lead: The technical lead is responsible for managing the architecture of the project as well as tracking and helping to resolve technical issues and dependencies that arise between engineers and between subsystems. The technical lead also plays a key part in defining the detailed task plans for each implementation cycle. With a broad view of the system, the technical lead can make sure that tasks scheduled for an implementation cycle are feasible and that they all contribute to the stated deliverable for the cycle.

Fig. 6. Sample two-week EVO cycle.

Monday	Tuesday	Wednesday	Thursday	Friday
Final Test of Last Week's Build	Release Last Week's Build to Users			
Review and Enhance Analysis Models for New Features	Create Design Models for New Features			
	Begin Implementation of New Features	Incremental Build Overnight		Weekend Build from Scratch

Monday	Tuesday	Wednesday	Thursday	Friday
	All User Feedback Collected	Functionality Freeze—No New Features Added Beyond this Point	Test New Functionality	Test New Functionality
		Incremental Build Overnight	Review Feedback, Determine Changes for Next Release	Weekend Build from Scratch

User liaison: The user liaison manages the team's interaction with the users, including setting up the user feedback process by defining expectations of the users, locating and qualifying users against these expectations, and coordinating any initial training that the users will need on the system. Once the development phase is underway, the user liaison will be responsible for collecting feedback, tracking user participation and satisfaction with the process, and ensuring that users are kept informed of the development team's response to their feedback.

It is important to keep in mind that the users providing feedback on the system may change over time. In the early development phase, it may be unrealistic to deliver the system to actual users, since there may simply not be enough functionality in the system. For these releases, other members of the project team or other members of the organization can act as surrogates for actual users.

Defining the Standard EVO Cycle. The next step in planning the development phase is to define the standard EVO cycle to be used. This task includes establishing the length of the cycle as well as the milestones within the cycle. The general rule of thumb is to keep the cycle length as short as possible. Within Hewlett-Packard, projects have used a cycle length as short as one week and as long as four weeks. The typical cycle time is two weeks (see Fig. 6). The primary factor in determining the cycle length is how often management wants insight into the project's progress and how often they want the opportunity to adjust the project plan, product, and process. Since it is more likely that a team will lengthen their cycle time than shorten it, it is best to start with as short a cycle as possible.

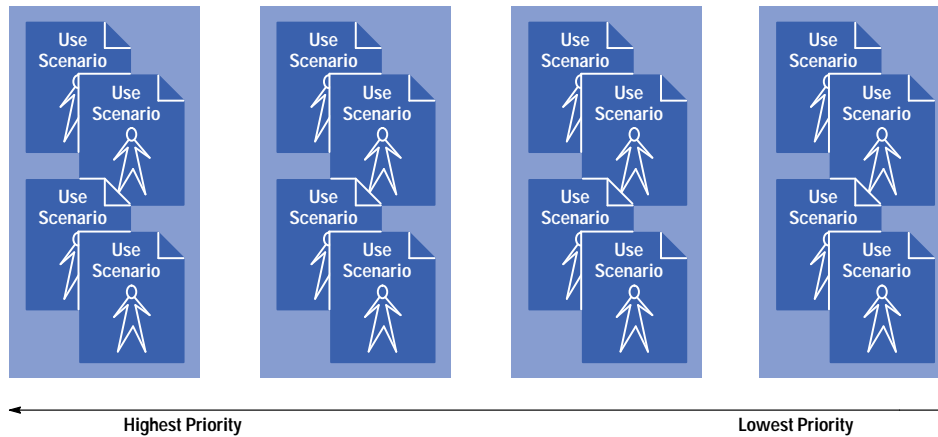
Grouping and Prioritizing Functionality. With key roles assigned and the standard cycle defined, the last step in planning the development phase is to group and prioritize the functionality into implementation chunks. The chunks must be no larger than can be delivered in the standard cycle time. Prioritization ensures that critical or high-risk features are completed early and that low-risk features are delivered last. Some of the most common criteria used for grouping and prioritizing functionality will be discussed later in this section.

The deliverable from the planning phase is an implementation schedule that maps all functionality for the system into implementation cycles and provides enough detail for the first three or four cycles so that actual implementation can begin. To help develop this schedule and to maintain a user perspective, the Fusion use scenarios and system operations provide a useful grouping of system functionality. System operations, which may appear in multiple use scenarios, are grouped together to define use scenarios.

The first step is to divide the system development into four or five major chunks and to group those use scenarios that include top-priority functionality into the first chunk (Fig. 7). The rest of the use scenarios can then be grouped into the following major chunks, with the use scenarios containing the lowest priority functionality in the last chunk. At this stage each chunk should contain approximately the same number of use scenarios.

The next step is to order the use scenarios within the first chunk using the same criteria as before (Fig. 8). When producing this ordering, it is not uncommon to move scenarios between groups to achieve a better balance and sequence. Since system operations may appear in multiple use scenarios, many of the system operations that are contained in the use scenarios of later groupings will be implemented with use scenarios in earlier groupings. Therefore, it is best to have the fewest use scenarios in the first chunk and the most in the last chunk.

Fig. 7. Prioritize use scenarios.



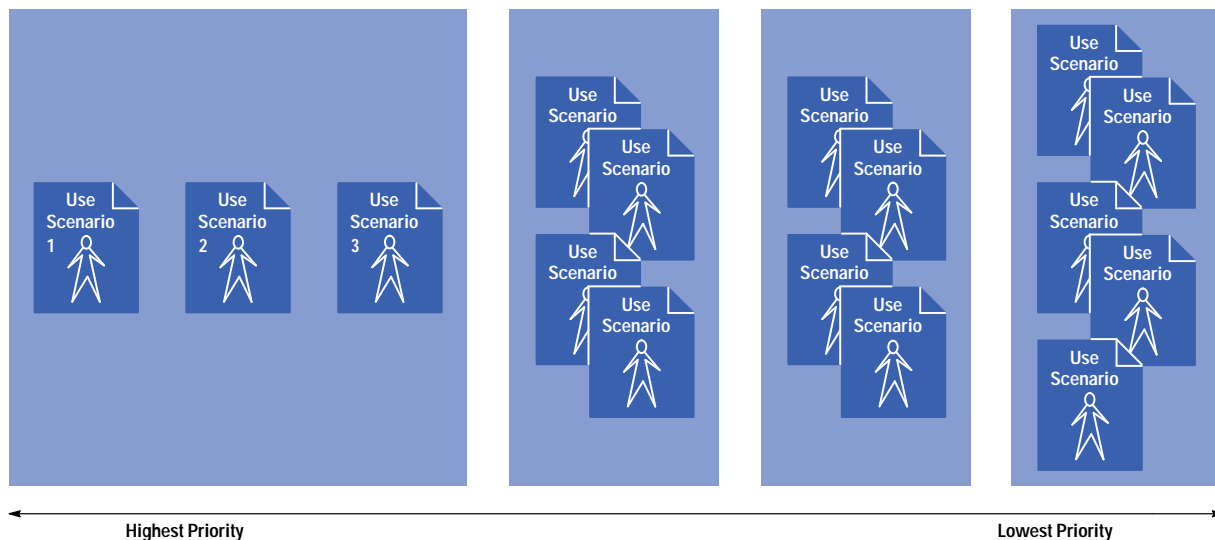
The system operations from the use scenarios in the first group can now be grouped and sequenced into the first few implementation cycles (Fig. 9). Keep in mind that the deliverables from each cycle should be defined in such a way that they can be validated by a user of the system. For these early cycles, the limited functionality may be best validated by another member of the development team. The key concept is that you must be able to validate the success of the cycle in some way.

When estimating the number of system operations that the development team can implement in a cycle, experience has shown that taking the common wisdom of the team and dividing that number in half yields the best results. Because this approach to development may be new to the team, it is extremely important from a motivational perspective that these first few implementation cycles be successful. Also, keep in mind that there is a fair amount of infrastructure developed and put in place during these first few implementation cycles as well. The tools and the process will undergo significant refinement during these first few cycles. For these reasons, keep the functionality content of the first few implementation cycles to a minimum.

A technique used widely within Hewlett-Packard is to adopt a naming scheme for the implementation cycles. One team used the names of wineries from their local Northern California region. As they completed each cycle, their project manager would buy a bottle of wine from that winery and store it away. Once several cycles were completed, the team would celebrate by taking the wine to a fine restaurant for lunch.

The final step is to estimate the number of cycles needed for the rest of the intended functionality and to project a final implementation completion date (Fig. 10). This is accomplished by counting the new system operations that must be implemented in the rest of the chunks and dividing by the number of system operations that can be completed in each cycle to give the total number of implementation cycles. In the example used to illustrate the planning process, the estimated length of the implementation phase is 32 weeks. To facilitate communication, it is useful to assign themes to each of the implementation chunks. The project team and the users will need both a detailed and a high-level view of the project, but

Fig. 8. Order the first group of use scenarios.



there are typically many members of the organization that prefer to see just the “big picture.” The themes can help convey that big picture.

With the deliverables now defined for the first several EVO cycles, the technical lead can prepare the detailed task list for these cycles. This detailed task list should include a clear description of the task, an owner for the task, and any dependencies that the task may have on other tasks within the cycle.

It is not necessary to provide any additional detail for the groupings of use scenarios beyond the first. It is only necessary to make sure that all functionality as it is defined at this early stage is accounted for and that an overall estimate of the effort is calculated. It is expected that experiences from the first few implementation cycles will affect future cycles in many ways. These later implementation cycles will be defined in more detail several cycles before their start date. On small projects with one or two collocated teams, detailing the next three or four implementation cycles is adequate. On larger projects, it may be necessary to maintain detailed schedules that reach further out in time.

Fig. 9. First implementation cycles defined.

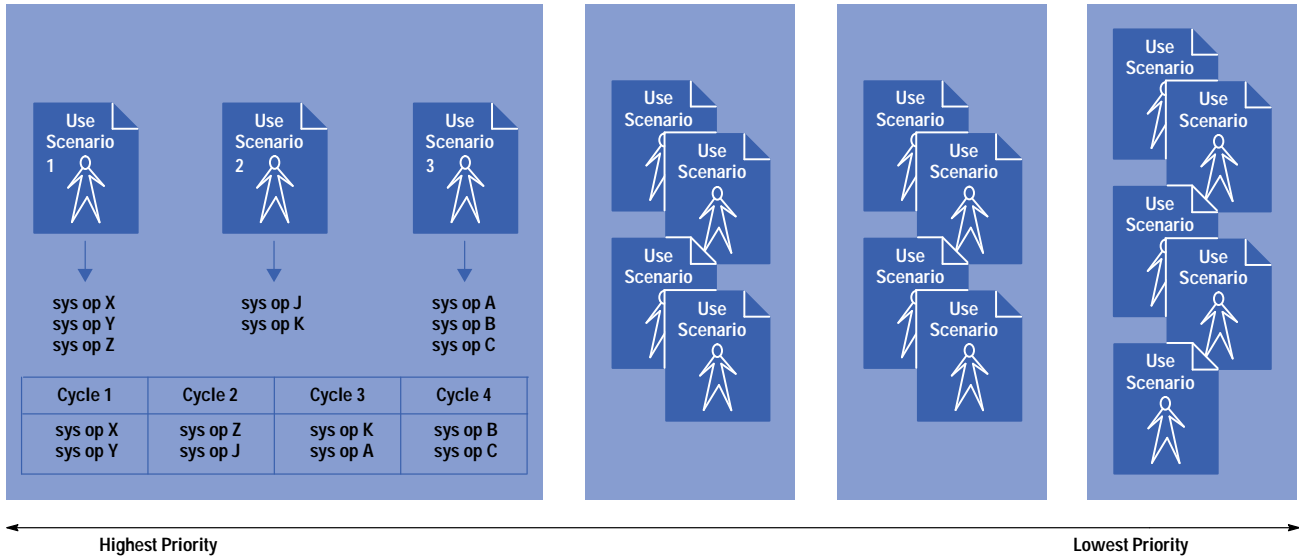
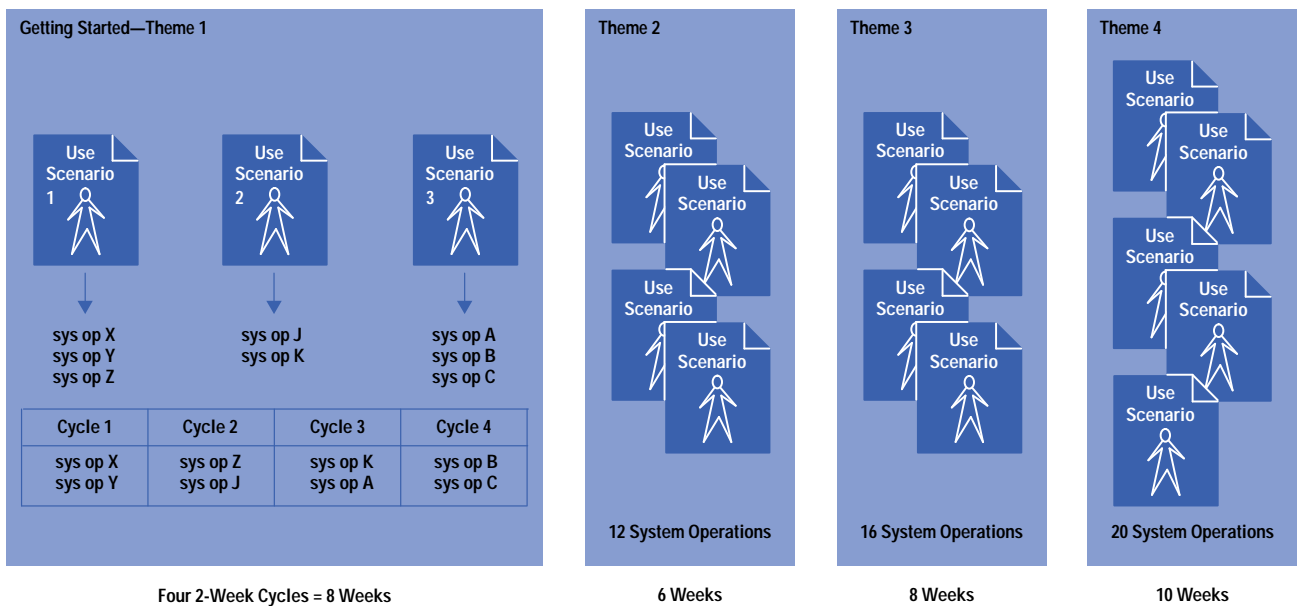


Fig. 10. Completed implementation plan.



Some of the criteria commonly used in setting priorities during this initial planning activity are the following:

- Features with greatest risk. The most common criterion used for prioritizing the development phase implementation cycles is risk. When adopting object technology, many teams are concerned that the system performance will

not be adequate. Ease-of-use is another common risk for a project. The use scenarios that will provide the best insight into areas of greatest risk should be scheduled for implementation as early as possible.

- Coordination with other teams. Most software development teams today have commitments to or are dependent on other teams. For example, firmware development depends on some form of hardware development. Reusable software platforms make a strong commitment to the products that are built on them. It may be necessary to adjust the priority assigned to functionality to accommodate these dependencies and commitments.
- “Must have” versus “want” functionality. All product features are not created equal. Some features are considered critical to the success of a project, while some features would simply be nice to have. Some development projects must meet well-defined standards and may even have to pass certification tests of their functionality that are defined by governing regulatory agencies. On these projects, it is often best to complete the required or “must have” functionality before the value-added or “want” functionality. Those use scenarios that capture the required functionality should be given higher priority than those that capture only desired functionality.

This same criterion can also apply to core or fundamental functionality that must be in place before additional functionality can be implemented. It may be necessary to build up in a layered fashion the core functionality that all other functionality will depend on. It is imperative that each cycle contributing to the core functionality be defined so that some validation or feedback can be obtained.

- Most popular or most useful features first. If project risks are minor and if project commitments and dependencies are insignificant, then prioritization of use scenarios can be based on value to the intended user. Those use scenarios that are the most popular or will be of the most value to the user should be completed first.
- Infrastructure development: A significant amount of development environment infrastructure must be put in place during the first few implementation cycles. The tools that will be used, such as the compiler, debugger, and software asset configuration manager, as well as the processes that are adopted, can be developed in an evolutionary fashion in parallel with the functionality intended for the user. Some teams have found it valuable to make the infrastructure tasks an explicit category in the plan for each implementation cycle.

Development Phase

With both the development phase plan and the detailed plans for the first few EVO cycles in place, the implementation process can begin. Each EVO cycle consists of the same basic steps: refining the analysis models, developing the design models, and writing and validating the code. The customer feedback process is executed in parallel with these tasks. The deliverables from the previous EVO cycle are evaluated by selected users or their surrogates, and decisions are made that shape the content of the subsequent EVO cycles.

Refining the Analysis Models. The EVO cycle begins with a review of the existing Fusion analysis models against the functionality or system operations defined as deliverables for that cycle. For each cycle, new functionality may be defined for delivery and existing functionality may be identified for modification.

The process for moving through the Fusion analysis models remains the same. Use scenarios that include the system operations must be reviewed for changes that were the result of feedback and refinement from previous EVO cycles. The object model must be reviewed for similar changes. Additional detail may be required in the object model. The system operation descriptions are reviewed for any changes and to ensure a common understanding by all members of the team.

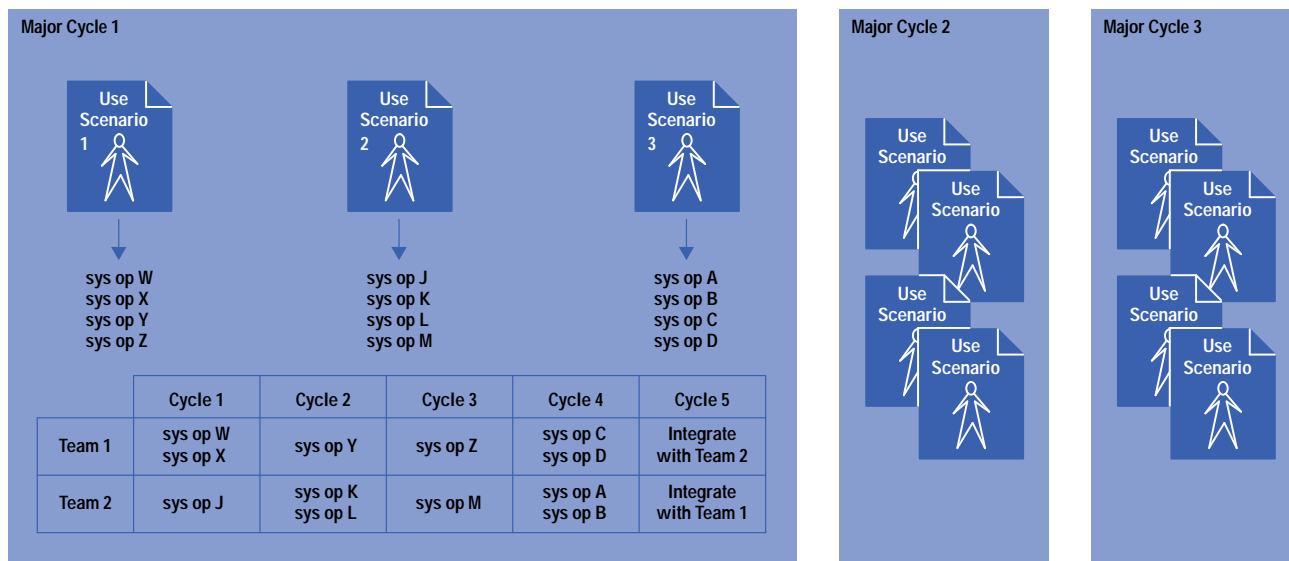
The technical lead is a key player during the refinement of the analysis models. Because they represent the overall architecture for the system, any extensions or enhancements of the models must be made without serious compromise to the integrity of the architecture. If compromises must be made, they should be logged as defects against the architecture and considered for possible repair in a later EVO cycle.

Design Models. Based on the clear understanding of the deliverables for the cycle generated by the review and refinement of the analysis models, the Fusion design models can be created or updated. Object interaction graphs will determine the new classes that will be needed or the new methods that will be added to existing classes. The Fusion design models determine what coding must be done for the cycle.

Coding and Validation. In addition to the code that must be generated to implement the design models, any tests needed to validate this work in later cycles must also be completed. Many teams make use of test harnesses to validate their code during the early cycles of development. These test harnesses are software modules or subsystems that can exercise the method interfaces of other software subsystems. They are particularly useful during the early cycles of development when major portions of the architecture have not been implemented. They also provide great value in later EVO cycles as tools for focused and automated regression testing.

Customer Feedback. The customer feedback loop operates simultaneously with the implementation tasks. Beginning with the second cycle and continuing throughout the development phase, some group of users or surrogate users will be validating the product that the team has completed so far. The feedback that they provide must be evaluated against the value proposition of the project for appropriate decision making. It is important that the project manager, technical lead, and user liaison allocate enough time during each cycle to review plans, processes, and architectural documents to assess the impact of each decision.

Fig. 11. Hierarchical EVO Cycles.



System Test Using Use Scenarios. Although the use scenarios can be helpful in conducting unit and integration testing for each implementation cycle, they can provide the greatest value during system test. Since the use scenarios are not structured along architectural or subsystem boundaries, they tend to provide a broad level of system testing that generates paths of execution through the entire system. They may be augmented to generate boundary and stress-test conditions, and they can also serve as a basis for creating user-level documentation.

Scaling up for Large Projects

In the use of Evolutionary Fusion with large projects, and especially with those that include multiple development teams that may not even be collocated, there are a number of additional issues to consider. It may not be appropriate to integrate the deliverables from all project teams every EVO cycle. It is useful to define a higher-level set of EVO cycles and to integrate all work together at the end of those cycles. To manage these multiple levels of EVO cycles, as well as the broad set of technologies that may be involved, it is also useful to employ multiple technical leads, or architects.

Hierarchical EVO Cycles. As the size of a project team grows, a larger and larger portion of the standard EVO cycle is dedicated to integrating the work of the many project team members. To keep the standard EVO cycle as small and as efficient as possible and to let project teams progress in parallel, it is necessary to introduce hierarchical EVO cycles. These hierarchical cycles are essentially a formalized version of the chunks of functionality or groupings of use scenarios introduced earlier, under “Grouping and Prioritizing Functionality.”

The four or five major chunks or groupings that the use scenarios are initially broken into become the highest-level EVO cycles. As before, the use scenarios for the first chunk or EVO cycle are sequenced and the system operations allocated between multiple teams (Fig. 11). For large teams, it is also useful to add an integration EVO cycle at the end of each major EVO cycle.

Each team is expected to define its own user feedback and validation process for its minor EVO cycles. There will also be a feedback and validation process for each major EVO cycle of the system.

Role of Architects. Since it is difficult to define subsets of functionality that are completely independent of one another, it is important to have an identified individual or group of individuals to manage the dependencies throughout each major EVO cycle. This role is best played by the technical leads of each team, the architects. The architects play a key role in allocating system operations among the various teams during each planning phase, and they are best positioned to resolve any technical issues that emerge as a result of the parallel implementation approach. For large projects within Hewlett-Packard, weekly meetings or conference calls are typical for the architect teams.

Conclusion

Much of Hewlett-Packard’s success is attributable to the fact that it is a diverse company composed of many independent organizations. However, relatively few software development best practices have achieved widespread adoption in this environment of autonomy and diversity. Fusion appears to be an exception to this rule. Fusion’s appeal is largely a result of the respect that its creators have for software development teams. Fusion does not attempt to address every possible nuance of software development with complex notations and model variations. It does provide a reasonably simple, complete set of models that supports a team through most of the development process, acknowledging that software

engineers are highly educated and talented professionals and that they are best suited to adapt a method to meet their unique project needs and working styles.

Evolutionary Development has been positioned here as a life cycle for software development, but it really has much broader application to any complex system. Fusion, the method, is changing to better meet user needs using an evolutionary approach. Based on user feedback, we merged Evolutionary Development with Fusion as the deliverable from one evolutionary cycle. There have been a number of other changes to the method, as well as to the method of delivery, again all based on user feedback. As our experience with Fusion grows, so will the method. It is our hope that the Fusion user community will continue to share experiences and to evolve the method in a direction that is both respectful and useful to all software development teams. See the sidebar: ***Fusion in the Real World*** for a brief synopsis of the book.

Acknowledgments

It is impossible to thank all those that have contributed in some way to the material covered in this paper, but I must try. First, I would like to thank the many Hewlett-Packard development teams that I have had the privilege to work with. Their unwavering dedication to creating innovative products and to adopting innovative ways of working make Hewlett-Packard a very successful company and an extremely rewarding place to work. Next, I would like to thank my colleagues of the Software Initiative who have worked with me to make Fusion and object technology as easy to learn, adopt, and adapt as possible. I would like to offer a very special thanks to the reviewers of this material, Ruth Malan, Reed Letsinger, Elaine May, and Tom Gilb. Their wealth of knowledge and experience generated insights and suggestions that have added significantly to the clarity and presentation of this material. And finally, to Derek Coleman and his team, for providing us all with the very powerful and useful set of models and notation that we call fusion.

References

1. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, 1994.
2. B. Boehm, "A Spiral Model of Software Development and Enhancement," *ACM SIGSOFT Software Engineering Notes*, Vol. 11, no. 4, 1986.
3. J. Rumbaugh, J. "OMT: The Development Process," *Journal of Object-Oriented Programming*, May 1995.
4. G. Booch, "The Macro Process of Object-Oriented Software Development," *Report on Object Analysis and Design*, Vol. 1, no. 4, 1994, pp. 11-13.
5. *Software Development and Documentation, MIL-STD-498*, December, 1994.
6. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.
7. E. May and B. Zimmer, *Evolutionary Product Development at Hewlett-Packard*, Hewlett-Packard internal publication, 1994.
8. G. A. Moore, *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers*, Harper Business, 1991.
9. M. Ould, *Strategies for Software Engineering—The Management of Risk and Quality*, Wiley, 1990.
10. R. Crough and R. Walstra, *Structured Common Sense: A Design Approach for Front-End Software Development*, Hewlett-Packard internal publication, 1993.
11. E. May and T. Cotton, *Evolutionary Planning Workshop*, Hewlett-Packard internal publication.

-
-
- ▶ [Go to Article 4](#)
 - ▶ [Go to Table of Contents](#)
 - ▶ [Go to HP Journal Home Page](#)