HEWLETT-PACKARD JOURNAL

August 1996

HEWLETT PACKARD

table of contents

August 1996,
Volume 47, Issue 4

Articles

Implementing the Capability Maturity Model for Software Development

Continuous support for a software development improvement effort requires at least two things: a clearly defined improvement model and success at applying the model in the organization. One HP division was able to apply one such model and achieve measurable success on several product releases.

by Douglas E. Lowe and Guy M. Cox

Manufacturing entities are always looking for more efficient ways of producing products because they realize that an efficient process yields lower costs, better quality, and increased customer satisfaction. Software manufacturers are no different from their hardware counterparts in that they want to use the best software development process available. The Capability Maturity Model (CMM) for software, developed at the Software Engineering Institute (SEI) at Carnegie-Mellon University,* is a process model that provides excellent guidance to improve software development processes.

The Model

CMM is used to evaluate and improve the way software is built and maintained. First released in 1987, CMM was originally based on the experience of members of SEI. CMM has been continuously improved and refined since 1987 through successive revisions based on industrywide and worldwide input. Yet, even though it is based on experience, it is only a model, which is an abstract, general framework describing the processes used to develop and maintain software. Like any model, it requires interpretation to be used in a specific context. The approach used by CMM is to describe the principles and leave their implementation up to the managers and technical staff of each organization, who will tailor CMM according to the culture and the experiences of their own environment.

Perhaps the most well-known aspect of the CMM is its description of five stages, or maturity levels, of an organization's software process (see Fig. 1). The first level of the software development process, referred to simply as the *initial level*, is described as ad hoc, poorly controlled, and often with unpredictable results in terms of schedule, effort, and quality. At level 2, the *repeatable level*, the outputs of the process are consistent (in terms of schedule, effort, and quality) and basic controls are in place, but the processes that produce those results are not defined or understood.

Level 3, the *defined level*, is the point at which the software engineering practices that lead to consistent output are known and understood and are used across the whole organization. The *managed level*, or level 4, is where the defined elements from level 3 are quantitatively instrumented so that level 5, the *optimizing level*, can be achieved. Level 5 exists in organizations in which the development process operates smoothly as a matter of routine and continuous process improvement is conducted on the defined and quantified processes established in the previous levels.

Thus, CMM is seen as a maturity or growth model in which an organization works its way up the five levels and, even after having attained level 5, is still in the process of continually improving and maturing.

Each of the five levels is also defined by the key processes associated with it. There are 18 key process areas that make up the five levels (see Fig. 1). These processes were chosen because of their effectiveness in improving an organization's software process capability. They are considered to be requirements for achieving a maturity level.

Managerial processes are those that primarily affect the way management operates to make decisions and control the project. Technical processes are those that primarily affect the way engineers operate to perform the technical and engineering work.

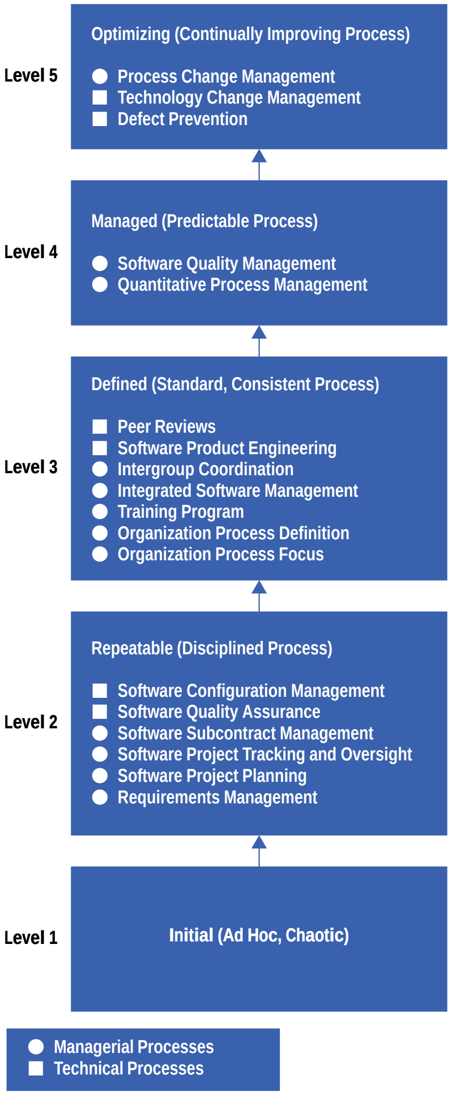
CMM provides a structure for each of the key process areas (see Fig. 2). Also, each key process area has one or more goals that are considered important for enhancing process capability. Fig. 3 shows the goals for three of the key process areas in level 2.

Finally, each key area has five common features or attributes:

- *Commitment to perform* describes the actions needed to ensure that the process is established and will endure and typically involves policies and senior management sponsorship.
- *Ability to perform* describes the preconditions that must exist in the project or organization to implement the software process competently. Ability to perform typically involves resources, organizational structures, and training.

* SEI is sponsored by the U.S. Department of Defense and was established in 1984 by the U.S. Congress as a federally funded research organization. Its mission is to provide leadership in advancing the state of the practice of software engineering to improve the quality of systems that depend on software.

Fig. 1. *The five levels of the Capability Maturity Model (CMM). The items listed at each level are called key process areas. These areas determine an organization's software development maturity.*


Level 5
Optimizing (Continually Improving Process)
● Process Change Management
■ Technology Change Management
■ Defect Prevention
Level 4
Managed (Predictable Process)
● Software Quality Management
● Quantitative Process Management
Level 3
Defined (Standard, Consistent Process)
■ Peer Reviews
■ Software Product Engineering
● Intergroup Coordination
● Integrated Software Management
● Training Program
● Organization Process Definition
● Organization Process Focus
Level 2
Repeatable (Disciplined Process)
■ Software Configuration Management
■ Software Quality Assurance
● Software Subcontract Management
● Software Project Tracking and Oversight
● Software Project Planning
● Requirements Management
Level 1
Initial (Ad Hoc, Chaotic)
● Managerial Processes
■ Technical Processes

- *Activities performed* describes the roles and procedures necessary to implement a key process area. These typically involve establishing plans and procedures, performing the work, tracking it, and taking corrective actions as necessary.
- *Measurement and analysis* describes the need to measure the process and analyze the measurements.
- *Verifying implementation* describes the steps needed to ensure that the activities are performed in compliance with the process that has been established. Verification typically encompasses reviews and audits by management and software quality assurance.

The intent of CMM is to describe what needs to be done to develop and maintain software reliably and well, not how to do it. CMM further describes practices that contribute to satisfying the intent of these attributes for each key process area. Any organization can use alternative practices to accomplish the CMM goals.

The Challenge

It usually takes most organizations about two to three years to go from level-1 to level-2 CMM compliance. However, based on very sound business reasons, our general manager at HP's Software Engineering Systems Division (SESD) committed us to reaching level 3 in 36 months. To show a commitment to this aggressively scheduled task, three people were assigned to the project.

During the planning stage, we discovered that because this was such a reasonable program, we could complete level 2 in less than 12 months with less than three full-time people. Perhaps more important, we found that this program could provide immediate benefits.

Fig. 2. *The elements that make up the structure of the level-2 key process area: Software Project Planning. Each key process area has a similar set of elements.*

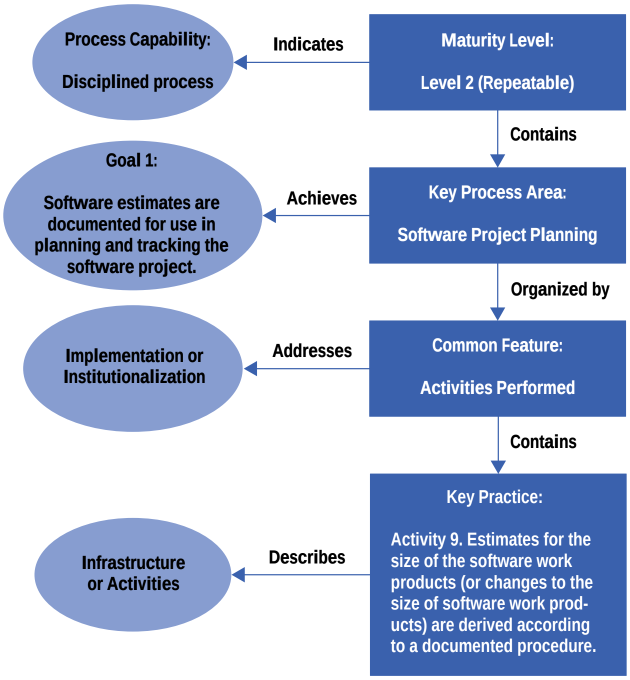
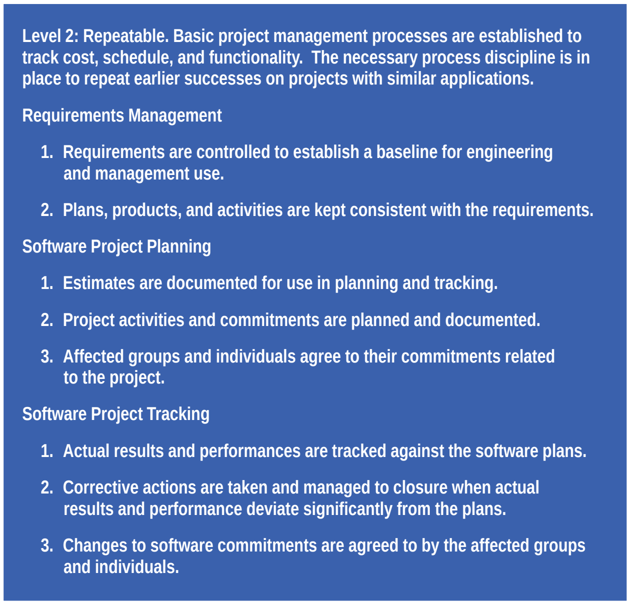

Process Capability:
Disciplined process
Indicates
Maturity Level:
Level 2 (Repeatable)
Contains
Goal 1:
Software estimates are documented for use in planning and tracking the software project.
Achieves
Key Process Area:
Software Project Planning
Organized by
Implementation or Institutionalization
Addresses
Common Feature:
Activities Performed
Contains
Key Practice:
Activity 9. Estimates for the size of the software work products (or changes to the size of software work products) are derived according to a documented procedure.
Infrastructure or Activities
Describes

Fig. 3. *The list of goals for three of the key process areas for level-2 CMM compliance.*



Level 2: Repeatable. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Requirements Management

1. Requirements are controlled to establish a baseline for engineering and management use.
2. Plans, products, and activities are kept consistent with the requirements.

Software Project Planning

1. Estimates are documented for use in planning and tracking.
2. Project activities and commitments are planned and documented.
3. Affected groups and individuals agree to their commitments related to the project.

Software Project Tracking

1. Actual results and performances are tracked against the software plans.
2. Corrective actions are taken and managed to closure when actual results and performance deviate significantly from the plans.
3. Changes to software commitments are agreed to by the affected groups and individuals.

In this article we describe how our product teams reached level-2 CMM compliance within a few months of starting the project, beginning with investigating the project in September and continuing with implementation in November 1994. By May of 1995 we had completed two deployment cycles with the product teams, and our internal audits of the teams' processes verified that we were operating at level-2 CMM. After several more audits of all the product teams, we found that the organization was operating at 100% level-2 CCM by August 1995. By May 1996 SESD had taken major steps toward achieving level 3 and is on track to achieve level 3 in 36 months. We expect that our description of how we accomplished this will provide insight for other organizations trying to achieve similar improvement in their software development processes.

The Improvement Project

HP's Software Engineering Systems Division (SESD) produces UNIX® software development tools, including SoftBench, C, C++, COBOL, UIM/X, HP Distributed Smalltalk,[1] and configuration management tools. At SESD, software engineers work together in cross-functional teams of 10 to 15 engineers. The cross-functional teams are made up of representatives from R&D, marketing, and learning products. These teams report to two business teams, one for the technical market and one for the commercial market. A combined quality and productivity function completes the organizational picture.

For several years SESD has attempted to change and improve its software engineering process. However, like most organizations the development priorities were to get products out first and work on improvement if time permitted. Usually, there was very little time to work on improvement. The development priority for SESD is still to get the product out (as it should be), but the software engineering process is seen as an integral part of achieving that priority.

When we began this project SESD had in place and in use a standard software life cycle, excellent software development tools, and good practices relative to configuration management, defect management, inspections, coding, and testing. SESD also had an excellent customer satisfaction program in progress, and a basic metrics collection program in place.

However, there were still weaknesses in our engineering process. It took a long time to define products and our design process needed some improvement. The life cycle was defined, but there was a lack of procedures for performing work and a lack of engineering discipline for following defined processes. As a result, products were consistently taking longer than expected, product completion dates were missed, and many features appeared in a product that weren't originally planned.

There was expert help available from HP Corporate Engineering's software initiative program. HP's software initiative program has built expertise in software process areas that are critical to software improvement, and we were able to enlist the help of this organization in the early stages of our project.

HP's software initiative group is a team of engineering and management experts who deliver knowledge and expertise through consulting in software engineering practices. The software initiative team works with multiple levels of the organization to optimize the organization's investment in development capability, accelerating the rate of making lasting strategic improvements and reducing the risk of change.

Beginning with the End in Mind

As mentioned above, we knew that it would be difficult to achieve level-3 CMM compliance in 36 months based on the experiences of other organizations inside and outside HP. Our problem was one of finding a way to institute process changes in an orderly way without adding major risks to the product teams' execution of their projects.

Adding to the sense of urgency were the very real business goals that needed to be achieved for SESD to be fully successful. The critical business issues were product time to market and quality improvement. These issues were evident in the past performance of the product teams in delivering products within an 18-to-24-month window and in the difficulty of delivering products that addressed major customer satisfaction issues. Responding to these critical business issues provided the real endpoint and goal that the entire organization could work to achieve.

The SoftBench[2] product team,which was the first group within our division to begin applying CMM level 2, had a business goal of releasing an update in a 12-month cycle, which would mean a 6-to-12-month reduction in typical cycle time. The team also had additional goals of reducing the number of lines of code in the product, delivering three major customer satisfaction enhancements and three competitive enhancements, and fixing all major customer reported problems.

We designed the software improvement project to help support the goals of the SoftBench product. The life cycle and the processes were defined to map into the objectives of a 12-month release cycle, provide methods for requirements analysis and specification in a short period, and provide aggressive management of defects during the development process.

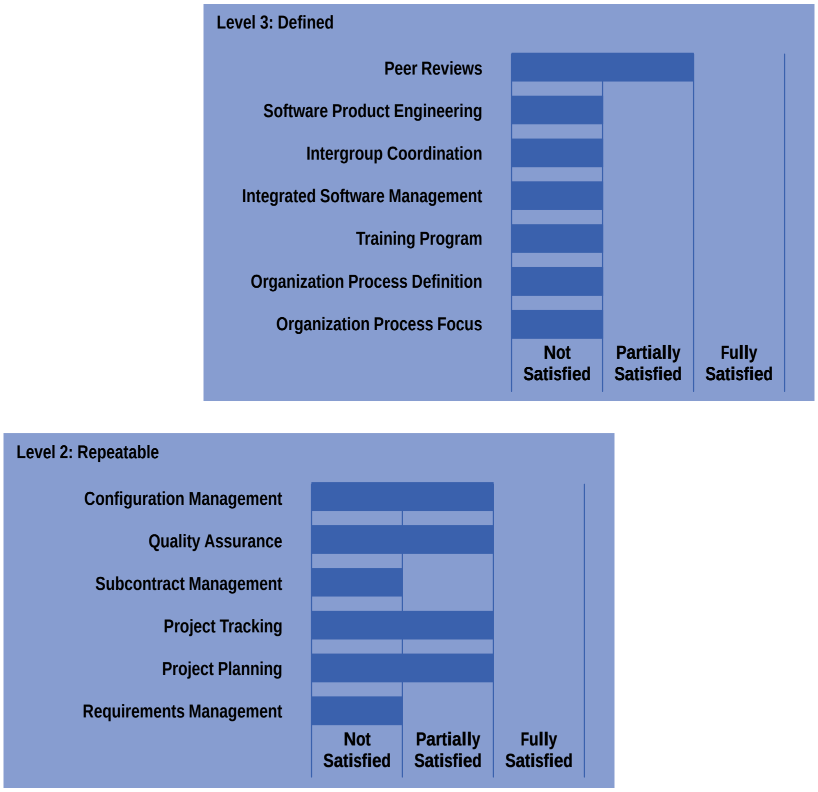
Investigation: Training and Planning

To evaluate our current practices we used a technique called a *software process profile*, which was developed by HP in collaboration with the Software Engineering Institute at Carnegie-Mellon University. The process profile is an assessment of the state of an organization's software development process, identifying strengths and weaknesses, highlighting the process improvements the organization values most, and recommending areas for change. The profile uses CMM as the standard for evaluating the software process. Using questionnaires and open-ended interviews, it provides results for all eighteen of the key process areas shown in Fig 1.

To complete the profile, engineers and managers in an organization must fill in a questionnaire that is designed to evaluate that organization's software process maturity against the CMM requirements. The results of this questionnaire are compiled into a software process profile for the organization. Fig. 4 shows SESD's software process profile.

We selected a representative group of 30 engineers and managers to participate in the assessment and, over a period of two days, we provided a short overview training session on CMM and a two-hour period for small groups to answer the assessment questions.

Fig. 4. *A software process profile indicating the compliance of SESD to the requirements of levels 2 and 3 of CMM at the time the software improvement project began.*



An example of a typical question from the project tracking and oversight area asks the participant:

“Does someone review the actual project results regularly with your section and lab managers?”

1–Almost Always 2–Often 3–Seldom 4–Almost Never 5–Doesn’t Apply 6–Don’t Know

For SESD, the process profile results (Fig. 4) indicated that four level-2 process areas were partially satisfied and two areas, requirements management and subcontract management, were areas that were not satisfied at all. Out of the seven level-3 processes, SESD partially satisfied only one area—peer reviews. These results meant that our organization would need to focus on implementing improved practices for all of the level-2 key process areas.

After the results were processed, we held several review sessions with engineers and managers to explain what the results meant. In one of the early sessions we asked the attendees to help us identify the benefits of improving our performance in each area and the roadblocks to achieving the requirements in each area. For example, Table I shows the assessment results from the software project tracking and oversight key process area.

We found that the assessment process was an important element for describing the improvements needed and determining how we should go about making the improvements. It quickly focused the entire organization on an improvement goal for our software processes and provided the starting point for getting participation in planning what actions we needed to take.

Applying the CMM Model Effectively

One of the critical steps in getting started was to understand CMM in detail. Over a period of several weeks three process consultants met daily to review the CMM specifications, develop our interpretation of the model, and translate it into language that could be applied within our organization.* This was an important step because CMM contains many practices that are better suited to large organizations and it was necessary to interpret which of these practices would apply to our organization. Also, during these meetings ideas of how best to deploy these practices were developed.

The process consultants examined each of the key process areas and decided what would be required to define our processes in a way that satisfied the requirements for level-2 CMM compliance. Several discoveries were made in the course of this work:

- SESD already had many process assets that could be leveraged to support the key process area requirements. These assets were our software life cycle, our formal documentation for patches, releases, and defect management procedures, and several informal methods of documentation for specifications, design, and testing.

- We discovered the need for a *process architecture* when we attempted to describe what deliverables would be needed to support the definition of our processes. A process architecture is analogous to other types of engineering architectures (e. g., buildings, software, hardware, etc.). It describes the layout of components needed to build a system for a specific purpose. Fig. 5 shows the process architecture elements we used to create parts or all of the documentation needed to provide SESD specifications for creating work products and performing software development activities.
- Level-2 CMM permits each project team to document the procedures that will be used in developing products. By writing a general procedure to cover the way product development is generally performed and then customizing the procedure for each project team, if necessary, we realized that we could save several months of effort and speed up the eventual move to level-3 CMM, where organizational processes must be standardized.

Table I
Assessment Results for Software Project Tracking and Oversight

Survey Questions	Percent of Survey Respondents		
	Fully	Partial	Not
Are the project's planned activities and deliverables tracked (e.g., schedule, effort, and tests)?	43	24	33
Are the actual results compared to your estimates throughout the project?	10	19	71
If there is a variance, does someone take corrective action or explain why the variance has occurred?	14	38	48
Are changes to the activities, deliverables, and schedule discussed with the people who will be affected?	19	58	23
Does someone review the project results regularly with your section and lab managers?	18	18	64

Formal Project Planning and Decisions

To give this improvement project the greatest chance of success and reduce the overall risk for the project, we developed a formal project plan covering every phase of the definitional work and the timing for deployment of the processes. This plan was reviewed and approved by the division staff before beginning the implementation.

Several key decisions needed to be made about how the project deliverables would be designed, reviewed, approved, and deployed into the product development team's operations.

Several models for process creation, approval, and deployment were examined and discussed before it was decided to use a define-deploy approach that mapped into each development project's life cycle.** This allowed the improvement project team to stage the work by life cycle phase (i.e., requirements, design, implementation, and test). This model also provided a structure within which process deliverables could be refined and improved.

Measuring Progress

Methods for measuring the progress of the project needed to be established. The only way to do this was through auditing the product development teams after each major checkpoint. Thus, we decided that a thorough audit of how the product development teams conducted work should be done after checkpoints when the pressure to complete the checkpoint had subsided and the team would be more open to listening to the audit findings. These findings were reviewed with the management team and any major issues were addressed by assigning owners and developing action plans. The results of the audits were summarized and published within the division to let everyone understand the accomplishments of the program.

Measurement of progress in each key process area was performed by interviewing project team members and using a checklist of requirements for that phase of the project. This checklist was based on the practices needed to satisfy the goals for each of the key process areas that apply to level-2 CMM.

* The process consultants were originally members of the SESD software quality assurance group. They had extensive experience in software engineering practices management and software consulting.

** A define-deploy approach means that the processes are defined in the project team just before application. This allows processes needed for the requirements phase to be defined and deployed as the requirements phase of the project is executed.

Fig. 5. *The process architectural elements and their purpose.*

Element Type	Purpose
Policy	• Specifies what will happen • Sets the cultural expectations • “That’s how we do things around here”
Procedure	• Specifies how it will happen • A set of steps for doing something • May specify who does what when
Checklist	• Specifies what or how in abbreviated format • A short form of procedures for easy reference or verification of actions
Template	• Specifies the content or quality of what will happen • Provides guidance for creating necessary work products
Training	• Provides organized information on processes that individuals need to perform their jobs • Covers policies, procedures, checklists, templates, or instructions • May be formal classroom or informal orientation
Work Product	• Specifies a plan or results • Created as an output of applying a defined process • May need to be “managed and controlled”

From previous experience, we realized that communication would play an increasingly important role for the project’s success as new procedures and supporting documentation were developed. We decided that all of the documentation would be integrated into the software life cycle so that there would be just one place to find the information. Secondly, this documentation would all be online and the Mosaic browser coupled with our configuration management tool would be used to store and control the documents and provide an easily navigable interface for users.

Implementation: Managing the Change

Getting an organization to adopt the changes necessary for level-2 CMM compliance was an important step in implementing our new processes. Defining the new policies and procedures and then providing training in these new policies and procedures was necessary but not sufficient. Changing the processes involved changing the culture, and this is where it was critical to get everyone thinking about the changes in a positive way. We approached this aspect of change management by deciding we would try to accomplish the following goals:

- Demonstrate success with a phased approach
- Leverage existing processes and minimize some areas of change
- Make the contributions of everyone very visible.

Demonstrate Success with a Phased Approach

The software improvement project needed to show results as early as possible to capture the attention of the organization and to keep things focused on process improvement. To accomplish the objective of showing results early, the deployment stages of the improvement project were designed to be about three months each.* This tactic provided visible results and feedback to the organization by coinciding with the life cycle phases of the SoftBench product.

The implementation stage of the improvement project consisted of a series of short steps for defining, reviewing, and approving the policies, procedures, and templates before deploying them to the product development teams. Communicating what was expected, describing changes from the way we used to do things, and providing group or individual training in new methods were very important steps in the way we deployed the process changes. We knew that the project teams needed to understand the rules early so that the project could proceed smoothly.

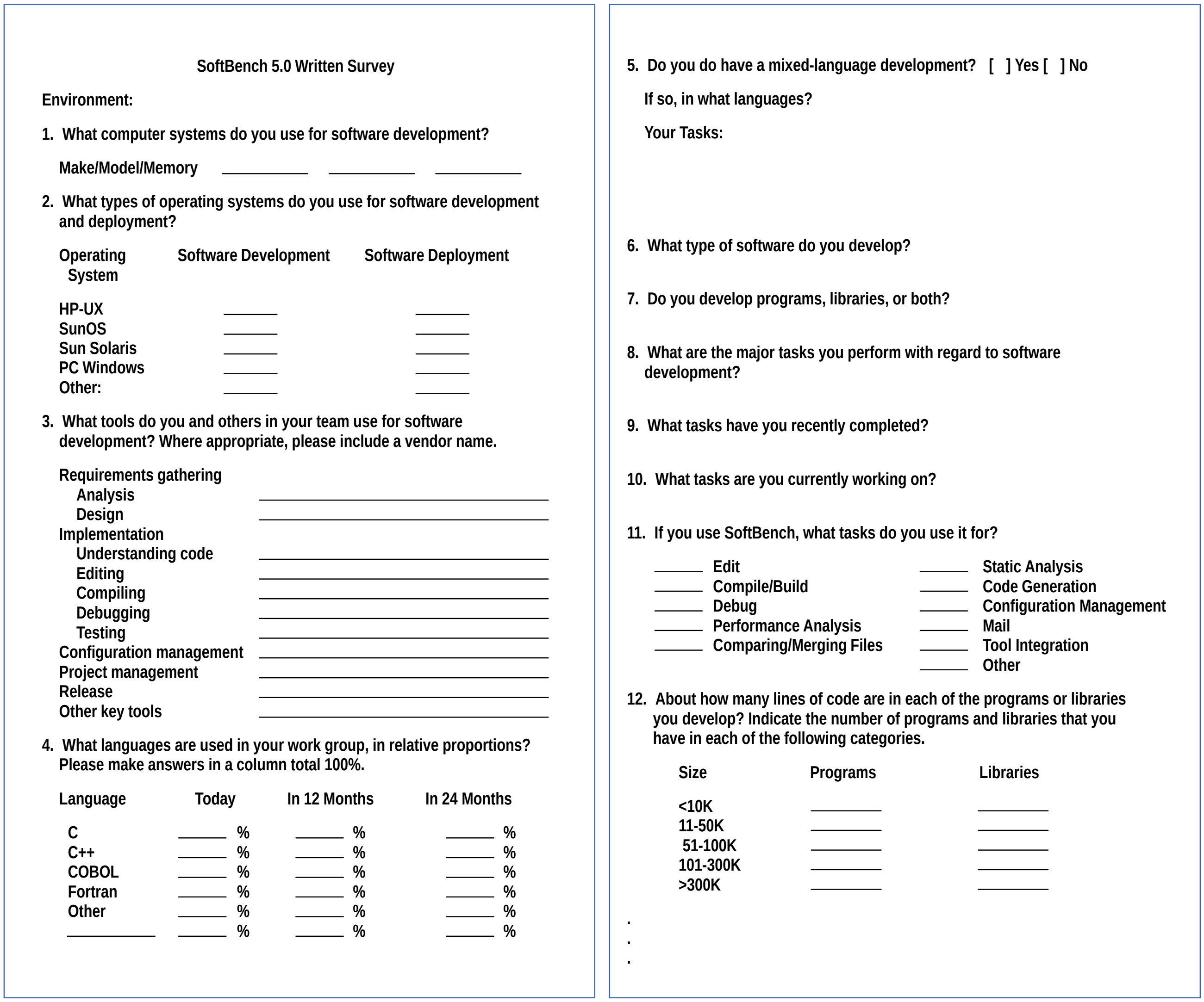
An Example: The Requirements Phase

During the requirements phase of the product life cycle, the key process areas we worked on were practices for requirements management and project planning. By using readily available customer survey data, structured processes, management review milestones, and training, we were able to reduce the time for the requirements phase from what was historically a six-to-eight-month process to three months. For the SoftBench teams, it was important to gain a deeper understanding quickly of the new features demanded by our customers and to translate this information into work steps that would be needed in the design phase.

* Deployment was a term we used to mean communication, training, and consulting with the product development team, followed by application of the procedures or templates by the engineers and managers. Templates consisted of the structured outlines for the work products, such as design specifications, test plans, data sheets, and so on.

One of the problems the teams faced was reducing the list of possible things to accomplish to a few high-impact requirements that could be accomplished within the schedule. Thus, to collect requirements information, survey questionnaires based on a user-centered design methodology were created to facilitate rapid feedback from customers using telephone surveys. The process consultants designed standard templates for the engineers to describe the customer requirements and the Mosaic browser was used to post work in progress, allowing managers to review the work. Fig. 6 shows a portion of one of these templates.

Fig. 6. *A portion of a template containing a survey that the SoftBench product team used to solicit customer requirements.*



SoftBench 5.0 Written Survey

Environment:

1. What computer systems do you use for software development?

Make/Model/Memory ________ ________ ________

2. What types of operating systems do you use for software development and deployment?

Operating System	Software Development	Software Deployment
HP-UX	_____	_____
SunOS	_____	_____
Sun Solaris	_____	_____
PC Windows	_____	_____
Other:	_____	_____

3. What tools do you and others in your team use for software development? Where appropriate, please include a vendor name.

Requirements gathering	
Analysis	____________________
Design	____________________
Implementation	
Understanding code	____________________
Editing	____________________
Compiling	____________________
Debugging	____________________
Testing	____________________
Configuration management	____________________
Project management	____________________
Release	____________________
Other key tools	____________________

4. What languages are used in your work group, in relative proportions? Please make answers in a column total 100%.

Language	Today	In 12 Months	In 24 Months
C	_____ %	_____ %	_____ %
C++	_____ %	_____ %	_____ %
COBOL	_____ %	_____ %	_____ %
Fortran	_____ %	_____ %	_____ %
Other	_____ %	_____ %	_____ %
________	_____ %	_____ %	_____ %

5. Do you do have a mixed-language development? [ ] Yes [ ] No

If so, in what languages?

Your Tasks:

6. What type of software do you develop?

7. Do you develop programs, libraries, or both?

8. What are the major tasks you perform with regard to software development?

9. What tasks have you recently completed?

10. What tasks are you currently working on?

11. If you use SoftBench, what tasks do you use it for?

_____ Edit
_____ Compile/Build
_____ Debug
_____ Performance Analysis
_____ Comparing/Merging Files
_____ Static Analysis
_____ Code Generation
_____ Configuration Management
_____ Mail
_____ Tool Integration
_____ Other

12. About how many lines of code are in each of the programs or libraries you develop? Indicate the number of programs and libraries that you have in each of the following categories.

Size	Programs	Libraries
<10K	_______	_______
11-50K	_______	_______
51-100K	_______	_______
101-300K	_______	_______
>300K	_______	_______

.
.
.

One criterion for determining what new features to include was an estimation of the resources and effort needed to design the new features. A standard procedure was provided for the engineers to do this (Fig. 7). These estimates were then available when the managers needed them for decision making and for the engineers to do more detailed planning of the design phase activities.

The decision making process was essential for narrowing the scope of the project and finalizing the work commitments for the next phase of the product development.

An Example: The Design Phase

During the design phase of the product life cycle, the key process areas were the practices in project tracking (i.e., managing and controlling work products, especially changes in requirements), and in project planning for the next phase. The success in applying the software life cycle and CMM level-2 processes to this phase of the work was evident from two major accomplishments. The first was that the team completed every aspect of the design work, including reviews and inspections, in three months. In the past, design specifications and design reviews were cut short because of schedule pressures. The team was also able to make decisions early in the project about eliminating features that would be too costly

Fig. 7. *A template for engineers to estimate the documentation delivery dates and code size of new software components.*

Tool/Component	External Specifications			Internal Specifications		Size Estimates	
	High Level	Prototype	Final	High Level	Low Level	Current	Estimated
A	mm/dd/yy	mm/dd/yy	mm/dd/yy	mm/dd/yy	mm/dd/yy	<n KNCSS>	<n KNCSS>
B	mm/dd/yy	mm/dd/yy	mm/dd/yy	mm/dd/yy	mm/dd/yy	<n KNCSS>	<n KNCSS>

n KNCSS = Number of KNCSS (Thousand Lines of Noncomment Source Statements)

to implement. For example, unit testing capability was a feature considered for the product, but it was eliminated during the requirements phase because of staffing trade-offs. This action substantially reduced the risk of schedule slippage later on.

Leverage Excellence to Minimize Change

As mentioned earlier, during the assessment and planning of the improvement project, we recognized that many practices used by SESD were already at level-2 CMM compliance. Therefore, it was important to leverage as many of the current practices and procedures to minimize the effort required and reduce the amount of change being introduced. We already had a standard software life cycle defined and in use, an excellent software development toolset, and good practices and tools in configuration management, defect management, inspections, coding, and testing. Part of our standard operations included a customer satisfaction survey and software metrics. These were all areas that were acknowledged as excellent and that should be maintained and supported.

The development environment consisted of a suite of tools integrated with SoftBench. UIM/X (user interface Motif) was used as a rapid prototyping tool. SoftBench provided the framework for editing, compiling, and debugging the code. Capabilities in SoftBench were also being used to assist in program understanding through call graphs and documentation of object-oriented designs for the new features. Integrated into SoftBench was a configuration management tool (SoftCM) for managing source code, and the ability to access our defect tracking tool DDTs from QualTrack. Having these tools already in place and used by the engineers was a major factor in maintaining developer productivity while making process changes in other areas.

Inspections and software metrics were already a well-established part of our culture. Although both of these areas are elements of level-3 CMM and we were working on level 2, we used them anyway because we did not want to lose the benefits we were achieving with these practices. Because inspections and metrics (defect tracking, schedule slippage, and effort reporting) were institutionalized in our engineering and project management practices, we recognized that this would be a distinct advantage for us when we went to level-3 CMM.

There were other opportunities to minimize change. These were in the areas of software configuration management, software quality assurance, and subcontract management. The first two areas needed only minor changes to be level-2 compliant. Our practices in these areas were robust enough but needed to be documented, with better definitions of roles and responsibilities. SESD wasn’t doing any software subcontracting, so it was decided to leverage a best practice from within HP to document a starting point for future use.

Make Contributions Visible

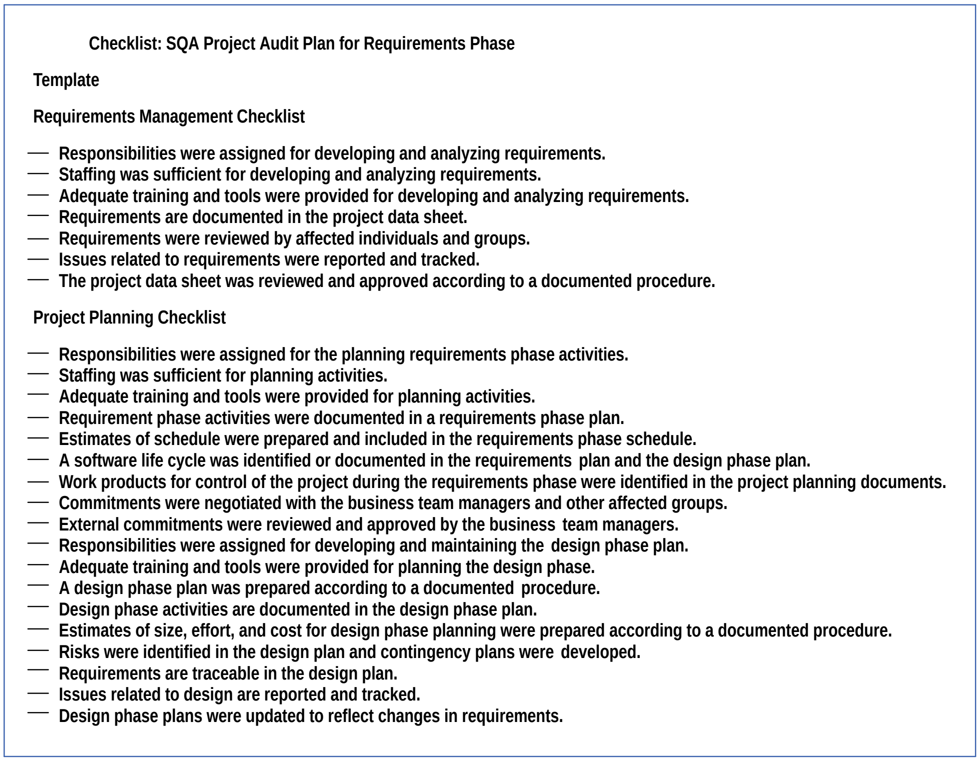
Managing change requires good communication of the change, acknowledgment of the progress made, and encouragement toward the goal. After each product life cycle checkpoint, the software quality assurance team performed an audit by interviewing the product team members using a checklist for level-2 requirements. The software quality assurance members were actually the process consultants from SEI. They were able to bring experience and maturity to the audit interviewing, reporting, and follow-up consulting. Fig. 8 shows portions of the checklist for the requirements phase.

These audits had several major benefits. First, the project team knew ahead of time that a process they used during a phase would be objectively evaluated by an independent team. This had the effect of elevating the importance of using a defined process. Second, the audit interviews uncovered critical issues and risks for the product’s development that were almost always a result of deviating from the project’s plan or processes.

For example, during the audit we identified a problem with inadequate staffing for test planning activities. The project plan called for the completion of test plans before design was finished. The audit found incomplete test plans for the context feature changes before the design complete checkpoint. This introduced additional schedule risk because the schedule and staffing were not accurately estimated. It turned out that this part of the project was actually critical path during the implementation and testing phase.

These issues and risks were reviewed by the management team and decisions were made to take corrective actions. This had the effect of identifying tangible contributions of the level-2 model.

Fig. 8. *Portions of the audit checklist for the requirement phase. These checklists were used to assess the SESD life cycle against the level-2 CMM requirements.*



Checklist: SQA Project Audit Plan for Requirements Phase

Template

Requirements Management Checklist

— Responsibilities were assigned for developing and analyzing requirements.
— Staffing was sufficient for developing and analyzing requirements.
— Adequate training and tools were provided for developing and analyzing requirements.
— Requirements are documented in the project data sheet.
— Requirements were reviewed by affected individuals and groups.
— Issues related to requirements were reported and tracked.
— The project data sheet was reviewed and approved according to a documented procedure.

Project Planning Checklist

— Responsibilities were assigned for the planning requirements phase activities.
— Staffing was sufficient for planning activities.
— Adequate training and tools were provided for planning activities.
— Requirement phase activities were documented in a requirements phase plan.
— Estimates of schedule were prepared and included in the requirements phase schedule.
— A software life cycle was identified or documented in the requirements plan and the design phase plan.
— Work products for control of the project during the requirements phase were identified in the project planning documents.
— Commitments were negotiated with the business team managers and other affected groups.
— External commitments were reviewed and approved by the business team managers.
— Responsibilities were assigned for developing and maintaining the design phase plan.
— Adequate training and tools were provided for planning the design phase.
— A design phase plan was prepared according to a documented procedure.
— Design phase activities are documented in the design phase plan.
— Estimates of size, effort, and cost for design phase planning were prepared according to a documented procedure.
— Risks were identified in the design plan and contingency plans were developed.
— Requirements are traceable in the design plan.
— Issues related to design are reported and tracked.
— Design phase plans were updated to reflect changes in requirements.

Key Results and Benefits

The improvement project to achieve level-2 CMM has had several direct results. First, the cycle time for the SoftBench release was reduced from 18 to 24 months to 14 months. This resulted in a significant reduction in engineering time, and consequently, a saving in the cost of developing the product. While the 12-month release cycle required a little longer than planned, the commitment date at the time of design completion was met with no schedule slip and no reduction in product reliability standards. Across all of SESD's products there was a reduction from an average of 4.6 open serious defects at manufacturing release in 1994 to 1.6 open serious defects per product in 1995. In fact, the product team fixed all outstanding major customer service requests during this period. In addition, the product team reduced the overall code size by 12%, reduced documentation size by 35%, and delivered three major customer satisfaction improvements and three major competitive improvements. All of these are, of course, significant business results coming from the level-2 CMM process. On SoftBench alone there was a cost reduction of two million dollars per year, or a return on investment of approximately 9 to 1.

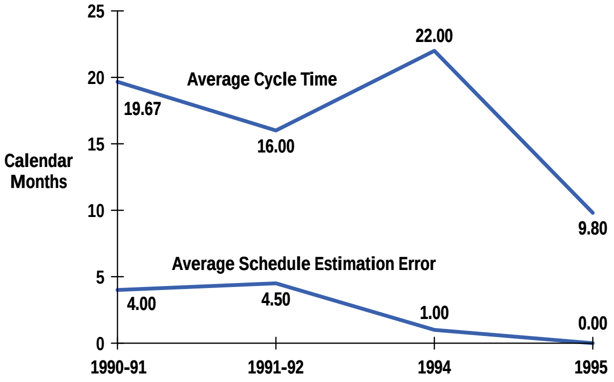
Other results of the improvement effort are that all of the product releases in 1995, which also met level-2 CMM compliance, were completed in under 12 months with no schedule slip. Fig. 9 shows the cycle times for similar projects over the past five years. The average cycle time for the 1995 projects was 9.8 months, which is a 46% reduction in cycle time from the running average of 18 months for previous years. The schedule estimation error (from design completion to project completion) was reduced to zero during 1995, compared with much higher errors in previous years.

Improved Execution. In past projects, the investigation phase usually lasted from 6 to 12 months, mainly because of the unstructured, exploratory nature of the process used. By adopting a formal structure (i.e., tasks and schedules), the investigation phase was reduced to four months.

Given the 12-month release cycle, it was critical to meet the intermediate phase deadlines to keep the project on track. In past projects, these milestone deadlines were consistently delayed by months. As a result of the careful planning and tracking processes used, the SoftBench team was able to meet the checkpoint deadlines within a very narrow margin of error (i.e., a few weeks).

Customer Orientation. Historically, our product requirements process had been focused on prototyping features and functionality that engineers identified through an infusion of "next bench" ideas. Product marketing would test these ideas with customers and use this feedback to select the best set of features to include in the next release. Everyone recognized the limitations of this approach in getting fresh ideas and direction for a product. Process improvement efforts were underway to define a user-centered design process that would really focus on what customers had asked for. The major change that occurred as a result of adopting the level-2 CMM practices is that we forced ourselves to adopt a new paradigm in the way we acquired and evaluated customer input.

Fig. 9. *The average cycle times (design complete to project complete) for projects similar to SoftBench over the last five years.*



The Ability to Respond to Changes. In most of our earlier projects, when changes in requirements or personnel occurred, there would be several weeks of confusion before revised plans or schedules could be started. In the new model of project management and level-2 practices, when a change occurs the management team knows that replanning must start immediately. Operating at level 2, the SoftBench team was able to estimate the impact of proposed changes and then schedule the time for engineers to replan and estimate the new schedule. When this occurred, we did not have the usual confusion about what to do. Instead, the team was able to respond with confidence and with very little lost time.

Conclusion

We believe that our circumstances and development environment are not unique. Many other organizations are trying various quality improvement programs for software development and finding it very difficult to make the changes necessary to be more rigorous and disciplined in their engineering practices. We also believe that we have discovered many of the essential ingredients to make a software improvement program succeed in a very short period of time.

Our work at SESD has convinced us that the Capability Maturity Model from SEI provides an excellent framework for defining software engineering process improvement for a small software organization. Achieving level-2 status has largely been a matter of establishing a direction with leadership from top management and then instituting a credible program for improving the practices used by software projects in planning and managing the work according to the CMM guidelines. We believe that other organizations can achieve similar results in one year if leadership and execution of the software improvement project are a priority for the management team.

References

1. E. Keremetsis and I. Fuller, “HP Distributed Smalltalk: A Tool for Distributed Applications,” *Hewlett-Packard Journal*, Vol. 46, no. 2, April 1995, pp. 85-92.
2. *Hewlett-Packard Journal*, Vol. 41, no. 3, June 1990, pp. 36-68.

Software Failure Analysis for High-Return Process Improvement Decisions

Software failure analysis and root-cause analysis have become valuable tools in enabling organizations to determine the weaknesses in their development processes and decide what changes they need to make and where.

by Robert B. Grady

When I was growing up, my father was fond of using sayings to encourage me to remember important lessons. One of his favorites was “Do you know the difference between a wise man and a fool?” He would then go on to say that a wise man makes a mistake only once. A fool makes the same mistake over and over again.

Applying my father’s saying to software defects, it sometimes seems as if there are many “fools” among software developers. However, there aren’t. Individually, they learn from their mistakes. What’s missing is organizational learning about our software mistakes. I guess that many organizations have earned my dad’s “fool” label.

One useful way to evaluate software defects is to transfer process learning from individuals to organizations. It includes not only analyzing software defects but also brainstorming the root causes of those defects and incorporating what we learn into training and process changes so that the defects won’t occur again. There are five steps:

1. Extend defect data collection to include root-cause information. Start shifting from reactive responses to defects toward proactive responses.
2. Do failure analysis on representative organization-wide defect data. *Failure analysis is the evaluation of defect patterns to learn process or product weaknesses.*
3. Do root-cause analysis to help decide what changes must be made. *Root-cause analysis is a group reasoning process applied to defect information to develop organizational understanding of the causes of a particular class of defects.*
4. Apply what is learned to train people and to change development and maintenance processes.
5. Evolve failure analysis and root-cause analysis to an effective continuous process improvement process.

How do these steps differ from other popular methods for analyzing processes? One popular method is process assessments, for example, SEI (Software Engineering Institute) process assessments.[1] Most assessments document peoples’ answers to subjective questions that are designed around somebody’s model of ideal software development practices. If such models are accurate and if peoples’ answers reflect reality, the models provide a good picture of an organization’s status. Thus, the results may or may not be timely, representative, or motivational.

The combination of failure analysis and root-cause analysis is potentially more valuable than subjective assessments, because it quantifies defect costs for a specific organization. The key point to remember is that software defect data is your most important available management information source for software process improvement decisions. Furthermore, subsequent data will provide a measurable way of seeing results and evaluating how methods can be further adapted when a specific set of changes is done.

Reactive Use of Defect Data (A Common Starting Point)

After initial analysis, everyone reacts to defects either by fixing them or by ignoring them. Customer dissatisfaction is minimized when we react quickly to fix problems that affect a customer’s business. This is often done with fast response to issues and by following up with patches or workarounds, when appropriate. Some Hewlett-Packard divisions track the resolution of “hot sites.” Fig. 1 shows an example.[2] Such a chart is a valuable way to track responsiveness, but it does little to prevent future defects. Furthermore, hot sites and patch management are very expensive.

Cumulative defects for long-lived software products are also tracked. For example, Fig. 2 shows the incoming service requests or discrepancy reports, the closed service requests or discrepancy reports, and the net progress for one NASA software project.[3] Some HP divisions also track progress like this,[2] although HP’s progress measure subtracts incoming defects from closed defects so that positive progress represents a net reduction in defects. NASA appears to do the reverse.

Fig. 1. *Tracking the number of hot sites during any particular week. For example, for the week indicated there were M−N hot sites that had been hot for a long time and N hot sites that had been hot for a short time. © 1992 Prentice-Hall used with permission.*

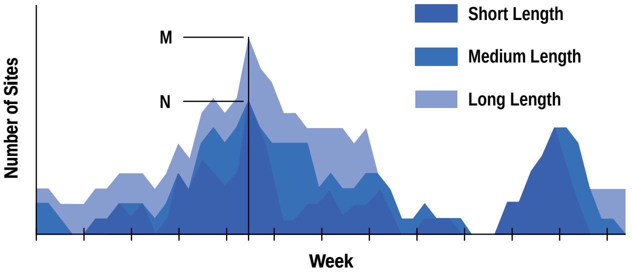
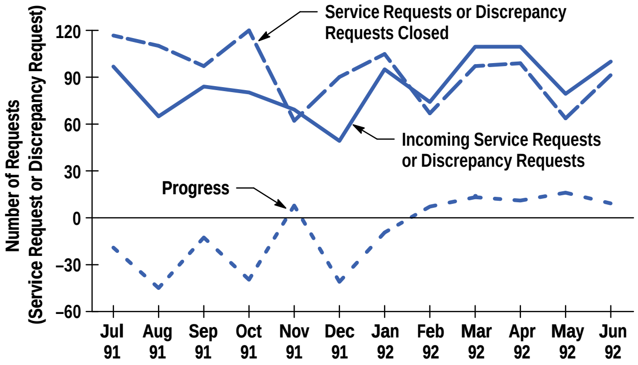


Fig. 2. *Incoming maintenance requests, closed maintenance requests, and net progress for one NASA project. This figure is reprinted by permission of the publisher from "A Software Metric Set for Program Maintenance Management," by G. Stark, G.L. Kern, and C. Vowell, Journal of Systems and Software, Vol 24, p. 243. ©1994 by Elsevier Science Inc.*



Both the hot site graph and the defect closure progress graph show reactive uses of defect data. In the examples, the respective organizations were using the data to try to improve their immediate customer situations. The alternative is to ignore the data or to react much more slowly.

Ignoring defect data can lead to serious consequences for an organization's business. For example, the division producing one HP software system decided to release its product despite a continuing incoming defect trend during system test. The result was a very costly update shortly after release, a continued steady need for defect repairs, and a product with a bad quality reputation. This is the kind of mistake that can cause an entire product line's downfall. A recent article described how one company learned this lesson the hard way.[4]

Responses should not be limited only to reaction. Besides endangering customer satisfaction and increasing costs, here are some other dangers that could occur if reactive processes aren't complemented with proactive steps to eliminate defect sources:

1. People can get in the habit of emphasizing reactive thinking. This, in turn, suggests that management finds shipping defective products acceptable.

2. Managers get in the habit of primarily rewarding reactive behavior. This further reenforces fixing defects late in development or after release. Late fixes are both costly and disruptive.

3. People place blame too easily in highly reactive environments because of accompanying pressure or stress. This is demoralizing, since the root causes of most defects are poor training, documentation, or processes, not individual incompetence.

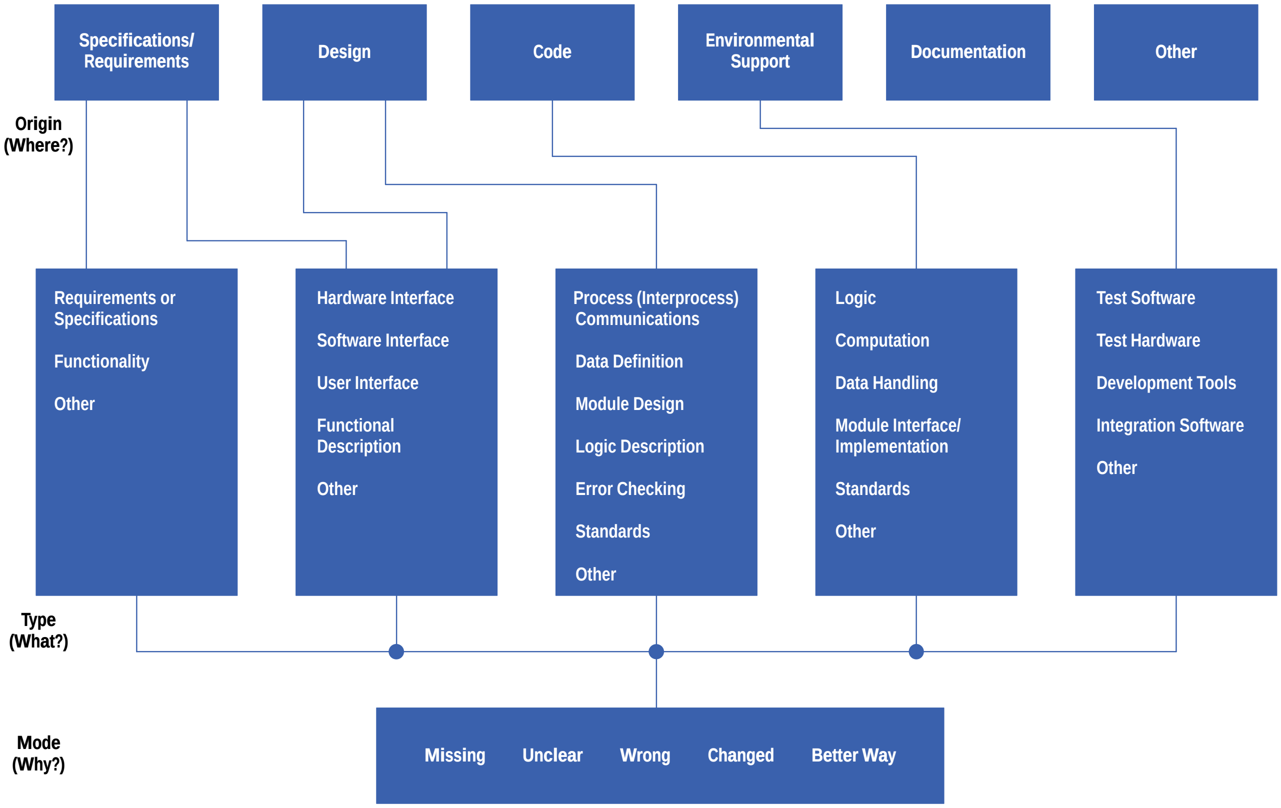
Remember that effectively reacting to defects is an important part of successfully producing software products. However, because business conditions change rapidly, many organizations can't seem to find the time to break old habits of using defect data reactively without considering ways of eliminating similar future problems. The elimination of the causes of potential future defects must be included in any successful long-term business strategy.

Failure Analysis (Changing Your Mental Frame of Reference)

The proactive use of defect data to eliminate the root causes of software defects starts with a change in mental frame of reference. The reactive frame generally focuses on single defects and asks "How much do they hurt?" It also considers how important it is to fix particular defects compared with others and asks "When must they be fixed?" The proactive frame asks, "What caused those defects in the first place? Which ones cause the greatest resource drain? How can we avoid them next time?"

Various reports have described successful efforts to analyze defects, their causes, and proposed solutions. But the terminology among them has differed, and the definitions could mean different things to different people. In the fall of 1986, the HP Software Metrics Council addressed the definition of standard categories of defect causes. Our goal was to provide standard defect terminology that different HP projects and labs could use to report, analyze, and focus efforts to eliminate defects and their root causes. Fig. 3 is the model that has evolved from our original definitions.[2]

Fig. 3. *Categorization of software defects. © 1992 Prentice-Hall used with permission.*



The model is used by selecting one descriptor each from origin, type, and mode for each defect report as it is resolved. For example, a defect might be a design defect in which part of the user interface described in the internal specification was missing. Another defect might be a coding defect in which some logic was wrong.

Fig. 4 gives some idea of how defects vary from one entity to another.[5] The different shadings reflect the origin part of Fig. 3. The pie wedges come from the middle layer of Fig. 3, the defect types. The eight largest sources of defects for different HP divisions are shown in each pie. All four results profile defects found only during system and integration tests.

We can immediately see from Fig. 4 that the sources of defects vary greatly across the organizations. No two pie charts are alike. These differences are not surprising. If everyone developed the same way and experienced the same problems, then we would have fixed those problems by now. Instead, there are many different environments. While many proposed solutions to our problems apply to different situations, they don’t necessarily apply equally well to all problems or all environments.

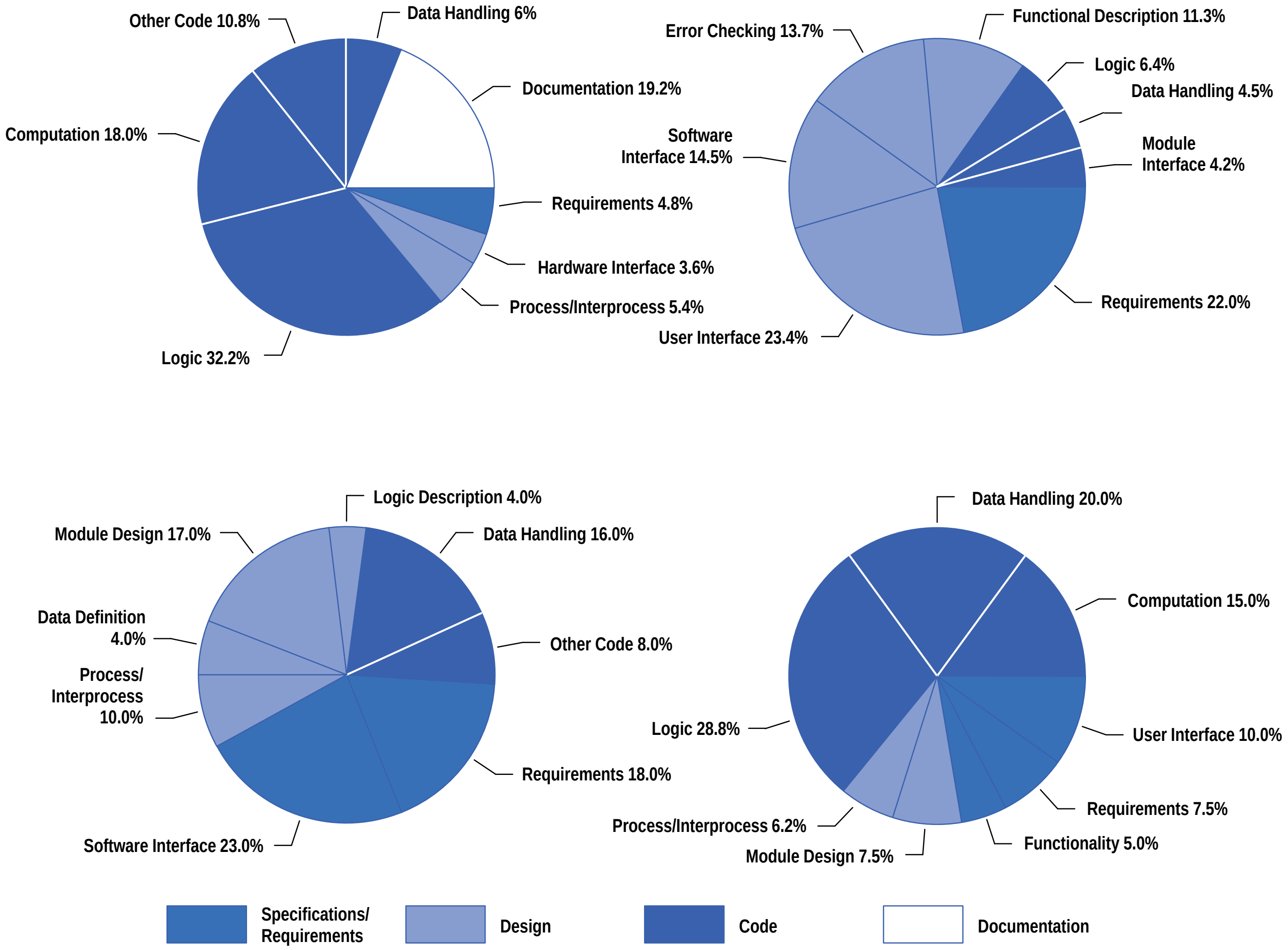
Some of the differences are because of inconsistencies in peoples’ use of the origin and type definitions. Because the definitions are just a means to focus process improvement efforts on the costliest rework areas, groups resolve inconsistencies when they define root causes to problems and brainstorm potential fixes. It is the triggering of these discussions that makes the data in Fig. 4 so important. Discussing root causes is a way to instill a process improvement attitude in an organization. Defect data will provide a measurable basis for decisions that must be made. By continuing to track defect data, an organization can also measure how successful its solutions are.

Acting on Causal Data

Collecting defect source data is only the first step. Persuasive as the data might be, improvements won’t happen automatically. Both managers and engineers must agree on what the data means and the importance of acting on it. One of the best ways to help ensure that this happens is to tie proposed improvements to stated business goals. This also keeps improvement priorities high enough to help ensure sustained management support.

Besides management support, some first-line managers and engineers affected by a proposed change must be motivated to do something and be assigned responsibility to plan and do the necessary changes. Finally, as for any effective project, there must be a way of monitoring progress and gauging success.

Fig. 4. Sources of defects found during testing in four HP divisions.



As a group, software developers now have several decades of software development experience. It is time to break out of our pressure-driven reactive habits and use our accumulated knowledge to drive lasting improvements. Failure analysis changes the way managers and developers look at software defects. This finally opens the way to a proactive frame of reference.

Root-Cause Analysis Processes

There are many possible ways to analyze root-cause data. Any successful way must be sensitive to project pressures and personnel motivation. HP has used several approaches in different organizations. For this discussion, I will label three that seem to evolve naturally from each other as *one-shot root-cause analysis*, *post-project root-cause analysis*, and *continuous process improvement cycle*. These three approaches include many common steps. Since the first is an introductory process, the most detailed explanation is saved for the post-project root-cause analysis.

One-Shot Root-Cause Analysis

A good starting approach for organizations that have not previously categorized their defect data by root causes is a one-shot root-cause analysis. This approach minimizes the amount of organizational effort invested by using someone from outside the organization to facilitate the process. At HP most divisions have defect tracking systems with complete enough information to extract such data.

The one-shot process has six steps.

1. Introduce a group of engineers and managers to the failure-analysis model (Fig. 3) and the root-cause analysis process. (About one hour.) Make it clear that the goals of the one-shot process are to:

   - Create a rough picture of divisional defect patterns.
   - Identify some potential improvement opportunities.

2. Select 50 to 75 defects from the defect tracking system using a random process. Make sure that the team thinks the defects have enough information to enable them to extract the necessary causal information. (About two hours sometime before the meeting.)

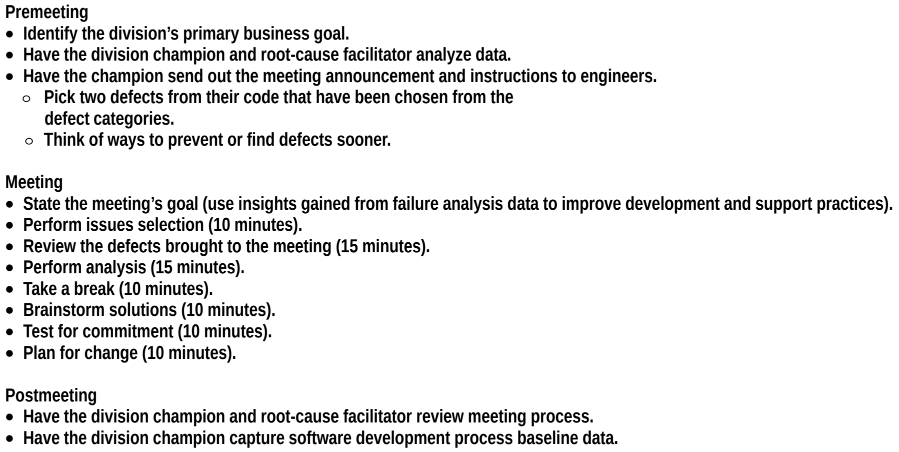
3. Have the people in the group classify one defect per person and discuss the findings as a group. Then have them classify enough defects so that you have about 50 total. Draw a pie chart of the top eight defect types. (About two hours.)
4. Pick two defect types to focus on. Create fishbone diagrams from the combined root causes and additional comments. A fishbone diagram is a brainstorming tool used to combine and organize group thoughts.[2,6] (About half an hour.)
5. Develop some recommendations for improvements. (About half an hour)
6. Present the results and recommendations to management. Make assignments to do initial changes. (About one hour)

Participants in this process have been generally surprised and excited that they could learn so much in a very short time. They have also been uniformly interested in adopting the analysis process permanently. How quickly they have followed through has varied, depending on many business variables such as immediate product commitments, other in-progress changes, or a tight economic climate.

Post-Project Root-Cause Analysis

The major difference between this process and the one-shot process is that organizations that start with the one-shot process have not previously collected causal data. Organizations that already collect failure-analysis data and have an understanding of their past defect patterns analyze their data and act on their results more efficiently. The steps in this approach follow the meeting outline shown in Fig. 5. Note that the times shown in Fig. 5 are intended to force the meeting to keep moving. It is best to schedule a full two hours, since all that time will be needed. The example used here to illustrate this process came from a root-cause analysis meeting done at an HP division shortly after a team at that division released a new product.

Fig. 5. *Root-cause analysis meeting outline.*



Premeeting

- **Identify the division's primary business goal.**
- **Have the division champion and root-cause facilitator analyze data.**
- **Have the champion send out the meeting announcement and instructions to engineers.**
  - ○ **Pick two defects from their code that have been chosen from the defect categories.**
  - ○ **Think of ways to prevent or find defects sooner.**

Meeting

- **State the meeting's goal (use insights gained from failure analysis data to improve development and support practices).**
- **Perform issues selection (10 minutes).**
- **Review the defects brought to the meeting (15 minutes).**
- **Perform analysis (15 minutes).**
- **Take a break (10 minutes).**
- **Brainstorm solutions (10 minutes).**
- **Test for commitment (10 minutes).**
- **Plan for change (10 minutes).**

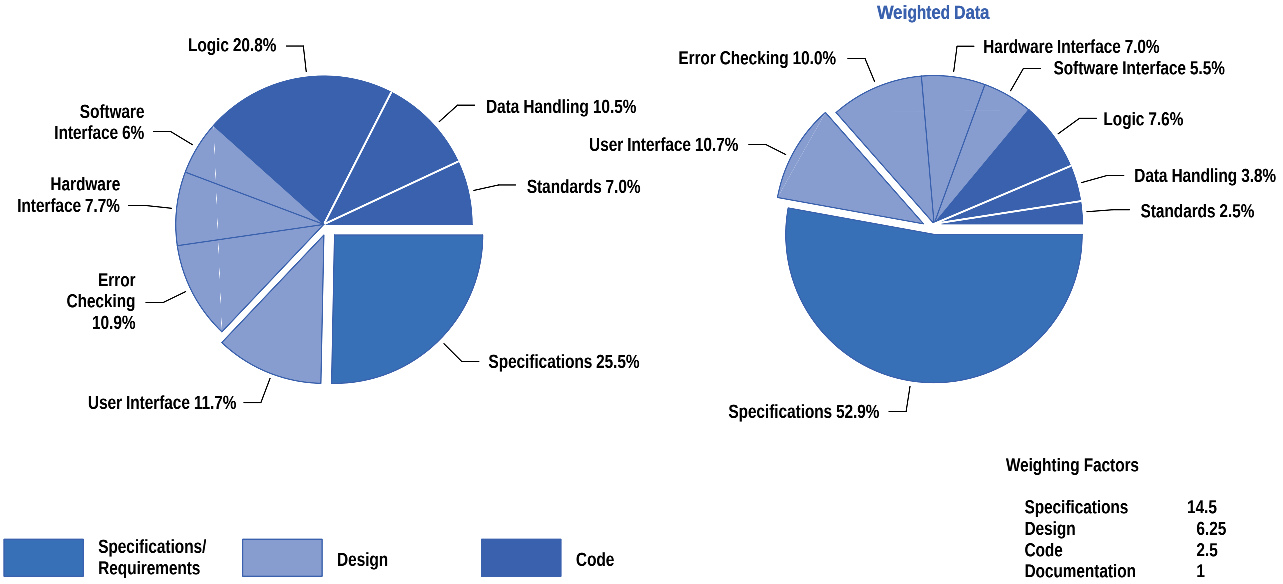
Postmeeting

- **Have the division champion and root-cause facilitator review meeting process.**
- **Have the division champion capture software development process baseline data.**

Premeeting:

- Identify the organization's primary business goal. This goal is an important input when prioritizing which high-level defect causes should be addressed first. It also helps to frame management presentations to ensure sustained management support. Typical business goals might be framed around maximizing a particular customer group's satisfaction, evolving a product line to some future state, or controlling costs or schedule to get new customers.
- The division champion and root-cause facilitator analyze the data. The champion is a person who promotes a process or improvement activity, removes obstacles, enthusiastically supports implementers and users, and leads through active involvement. The root-cause facilitator is a person who runs the root-cause analysis meeting. The champion and the facilitator need to be skilled at meeting processes and dynamics and be familiar with software development and common defect types. One simple data-analysis approach is to enter the data into an electronic spreadsheet. Draw pie charts of the top eight defect types by quantity and by find and fix effort (either actual or estimated). Fig. 6 shows the system-test data for four projects at one HP division. The shading represents defect origin information, and the pie wedges are defect types. The left pie chart shows the eight most frequently recorded causes of defects. The right pie chart shows the data adjusted to reflect that design and specification defects found during system test cost much more to fix than coding defects do. Since the HP division that provided this data did not collect their defect-fix times, the weighting factors are based on six industry studies summarized in reference 2. The right pie chart in Fig. 6 was prepared by multiplying the left pie chart wedge percentages (or counts) by the appropriate weighting factor and then converting back to 100%.

Fig. 6. Top eight causes of defects for one division.



- Select two defect types to brainstorm based on the best estimate of the organization's concern or readiness to implement solutions. The two defect types selected for this meeting were user-interface defects and specifications defects. The specifications defect type was picked because it was the largest division category (64 out of 476 defects were classified as specifications defect types for this project team). User-interface defects were picked because they were the largest category (110 defects) that the particular brainstorming team had experienced. Both categories represented significant divisional improvement opportunities.
- Send out instructions to engineers. The organization champion should have each engineer bring hard-copy information on two defects from their code, based on the chosen types. Tell invitees to think back to the most likely root cause for each defect and to propose at least one way to prevent or find each defect sooner.

Meeting:

- State the meeting's goal (use insights gained from failure-analysis data to improve the organization's development and maintenance practices). Present the defect categorization model, show typical patterns for other organizations, and show your organization's pattern. Set a positive tone for the meeting. Remind participants that they will be looking at process flaws, and that they must avoid even joking comments that might belittle the data or solutions discussed.
- Issues selection. Reiterate the reasons for selecting this meeting's particular defect types. Let people make initial comments. Address concerns about potential data inaccuracies (if they come up at this point) by emphasizing the solution-oriented nature of the brainstorming process. Suggest that inaccuracies matter less when combining pie wedges to consider solutions. For example, for the sample division meeting, some engineers had a hard time calling some defects "user interface" as opposed to "specifications." We simply used both labels for such defects during the meeting instead of getting sidetracked on resolving the differences. You want to get people ready to share their defects by discussing a future time (like their next major product release) when they will have done something in their process to eliminate the reasons for the defects.
- Review the defects brought to the meeting. Have engineers read their own defects, root causes, and solutions. The major reason to do this is to get attendees involved in the meeting in a nonthreatening way. Thus, don't criticize those who did not prepare, rather encourage them to contribute in real time. Unlike inspections, root-cause analysis meetings require very little preparation time for attendees. After their first meeting, attendees will realize this, and it will be easier to get them to review their defects before the next meeting.

Get in a creative, brainstorming mood by showing the engineers that all their inputs are right, and begin to form a shared understanding of terminology and definitions, and an acceptable level of ambiguity. This section also gives you some idea whether there is some enthusiasm for any particular defect types. You can use such energy later to motivate action.

The following two examples are from the root-cause meeting held by the example HP division. There were 12 engineers and managers at this meeting.

1. User-interface defect: There was a way to select (data) peaks by hand for another part of the product, but not for the part being analyzed.

   Cause: Features added late; unanticipated use.

Proposed way to avoid or detect sooner: Walkthrough or review by people other than the local design team.

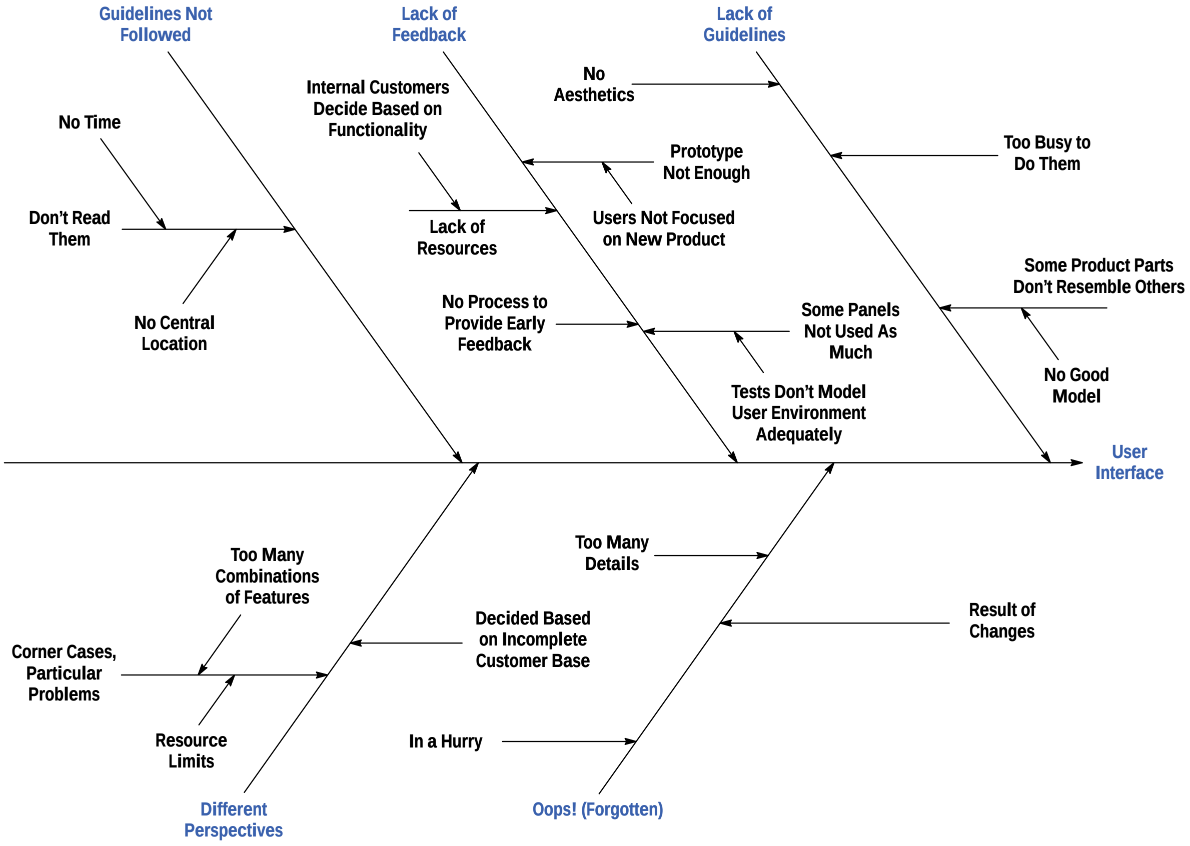
2. Specifications defect: Clip function doesn't copy sets of objects.

   Cause: Inherited code, neither code nor error message existed. Highly useful feature, added, liked, but never found its way back into specifications or designs.

   Proposal to avoid or detect sooner: Do written specifications and control creeping features.

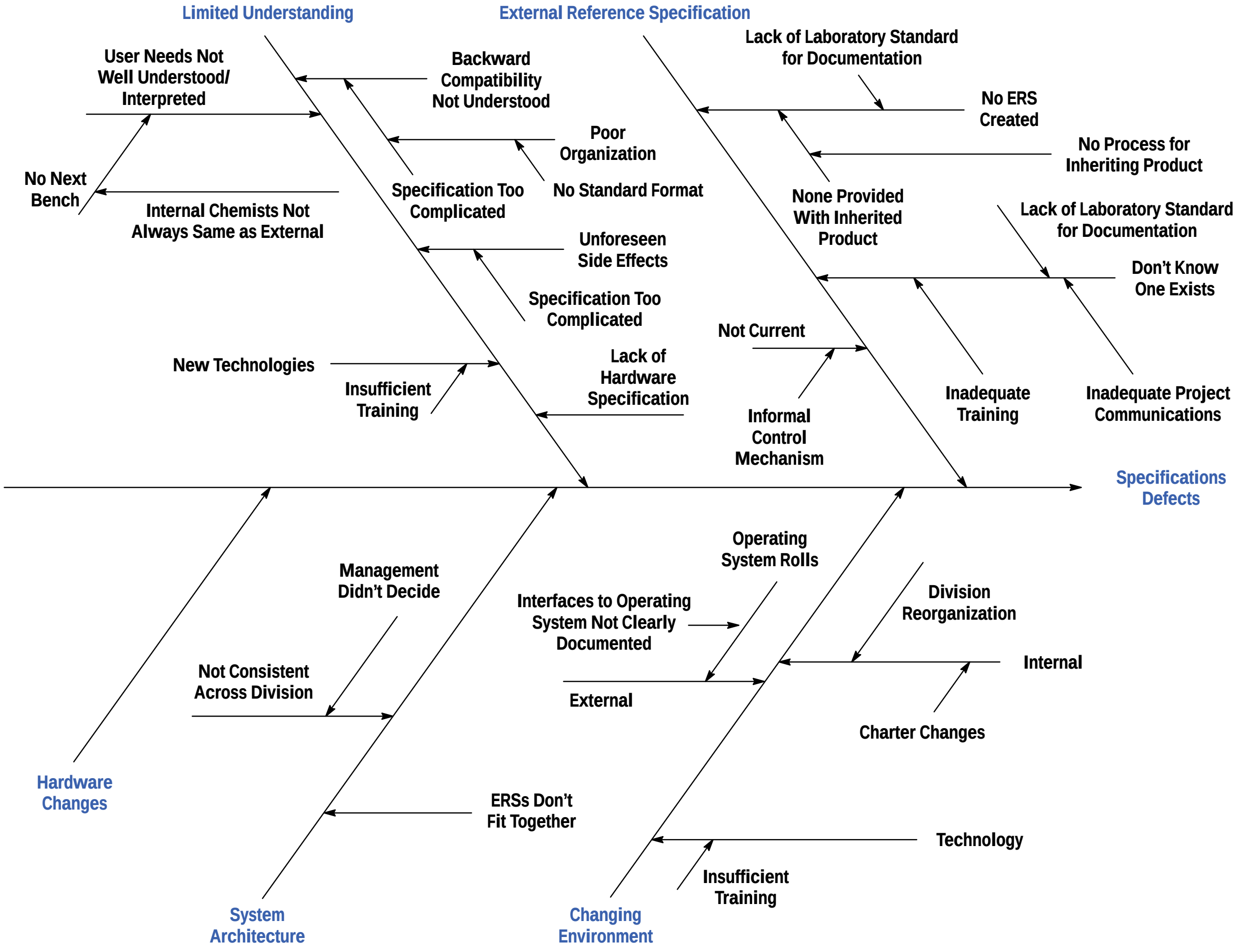
- Perform analysis. Create fishbone diagrams[2,6] from combined root causes and additional comments. Use this discussion to bring the group from their individual premeeting biases regarding defects to a group consensus state. A useful technique for grouping the defects is to write the suggested causes on movable pieces of paper. Then have the group silently move the papers into groupings of related areas. If some of the papers move back and forth between two groups, duplicate them. The resulting groupings are called an *affinity diagram*.[7] These are major bones of the fishbone that the group must name. Don't expect the fishbone to be perfect here or even complete. The next session will potentially contribute more. Also, don't get concerned about form. Let the group know that a fishbone is just a means to an end, that it will be cleaned up after the meeting, and that it is likely to change even after that point. The fishbone diagrams in Figs. 7 and 8 are from analyzing the two defect types mentioned above.
- Take a break. This type of meeting takes a lot of energy and focus. It's hard to sustain that for two full hours.
- Brainstorm solutions. Use this time as an orthogonal approach to analyzing the issues at hand. This is also the transition from analysis to action planning for change. Think about what group energy can be turned into solution planning.

Fig. 7. *Fishbone diagram for the causes of user-interface defects.*


Guidelines Not Followed
Lack of Feedback
Lack of Guidelines
No Time
Internal Customers Decide Based on Functionality
No Aesthetics
Too Busy to Do Them
Prototype Not Enough
Don't Read Them
Lack of Resources
Users Not Focused on New Product
Some Product Parts Don't Resemble Others
No Process to Provide Early Feedback
Some Panels Not Used As Much
No Central Location
No Good Model
Tests Don't Model User Environment Adequately
User Interface
Too Many Combinations of Features
Too Many Details
Result of Changes
Corner Cases, Particular Problems
Decided Based on Incomplete Customer Base
Resource Limits
In a Hurry
Different Perspectives
Oops! (Forgotten)

For our sample team, there was a lot of group interest in both defect types. Because a task force already was working on specification defects as a result of the previous root-cause analysis meetings, planning focused on user-interface defects. In the solution list they created, some of the solutions may seem vague. Remember that the brainstorm list is only an intermediate step toward defining action steps. Just be sure that the group understands what it means by the solutions. If members seem to understand the solutions, there is no need to slow down the brainstorming process for more precise definitions. These can be added later.

Fig. 8. *Fishbone diagram for the causes of specifications defects.*



Limited Understanding
External Reference Specification
User Needs Not Well Understood/ Interpreted
Backward Compatibility Not Understood
Lack of Laboratory Standard for Documentation
No ERS Created
No Next Bench
Poor Organization
No Process for Inheriting Product
Internal Chemists Not Always Same as External
Specification Too Complicated
No Standard Format
None Provided With Inherited Product
Lack of Laboratory Standard for Documentation
Unforeseen Side Effects
Don't Know One Exists
Specification Too Complicated
Not Current
New Technologies
Insufficient Training
Lack of Hardware Specification
Informal Control Mechanism
Inadequate Training
Inadequate Project Communications
Specifications Defects
Operating System Rolls
Management Didn't Decide
Interfaces to Operating System Not Clearly Documented
Division Reorganization
Internal
Not Consistent Across Division
External
Charter Changes
Hardware Changes
ERSs Don't Fit Together
Technology
Insufficient Training
System Architecture
Changing Environment

The solution list they created is as follows:

1. Learn from past experience—track user interfaces, particularly when changes occur.
2. When new functionality is thought of or added, always design and specify user-interface implications.
3. Evaluate other applications.
4. Use a checklist when designing panels.
5. Use the Caseworks design tool.
6. Complete an entire feature when you do it.
7. Give a new feature to someone else to use right away.
8. Solicit thoughtful feedback. Create guidelines for feedback and watch users use interfaces.
9. Perform usability walkthroughs and training.
10. Use standard modules (e.g., common dialog boxes).

- Test for commitment. Normally there is no need for this section, but some organizations that are more tightly controlled than others may not feel empowered to implement solutions. In these organizations, solutions should be directed toward doing what the group feels it is empowered to do. When those solutions are successful, they can be more broadly or completely applied. You may need to test to identify the roadblocks to change (e.g., time, schedule, etc.).

Our example HP division seemed very committed. This was reinforced in the next step when several people volunteered to initiate specific changes.

- Plan for change. Discuss which defects can be eliminated with the proposed solution. Create an action plan with responsibilities and dates. A model action plan might contain the following steps:

1. Establish working group	10/8
2. Meet and define outputs	10/15
3. Present objectives and gather inputs	11/1
4. Create a change process and artifacts	12/1
5. Inspect and fix process and artifacts	12/15
6. Celebrate	
7. Use and measure results.	2/1

Our example division team decided to create guidelines for user interface designs that addressed many of its fishbone-diagram branches. The division's action plan consisted of the following steps.

1. Patty will create a checklist for designing panels. (First pass by 12/17)
2. The project manager will set expectations that all new functionality will be accompanied by design and specification implications. (Consider using new specification formats.)
3. Art will give the project team a presentation on Caseworks.
4. Follow up the project presentation with a discussion on the use of prototyping.

Remember to end the meeting with a clear understanding of ownership and responsibility. Use standard project-management techniques to plan and schedule follow-up.

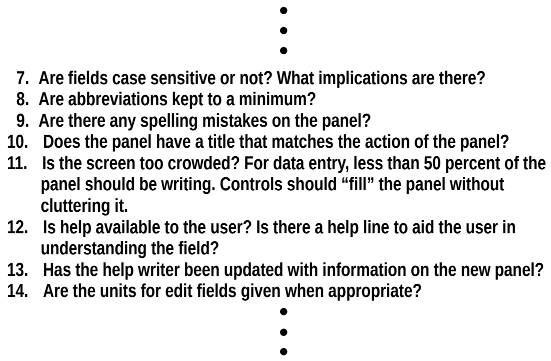
Postmeeting:

- Review meeting process. The organization champion and root-cause facilitator review the process and develop changes to meeting format, data collection, analysis, and responsibilities. They should redo the fishbone diagram, being careful not to change it so much that participants no longer feel that it is theirs. Promptly send out meeting notes that include the fishbone diagram, responsibilities and action items, and schedule dates.
- Capture process baseline data. As part of structuring a process improvement project for success, someone (the organization champion) should record a minimum amount of process information before and after the project.[2] It is particularly important to document the basic divisional processes so that when the improvement is done, the group can better understand other influences besides the particular changes that were made. In this example, the team didn't do this step.

Results from Eliminating Defect Root Causes

The team from the example division did their checklist and used it during their next project. It had 30 items to watch out for, based on their previous experience and their defects. Fig. 9 shows an excerpt from their checklist. Over 20 percent of the defects on their previous project had been user-interface defects (though the division-wide average was lower). The results of their changes were impressive.

Fig. 9. *A checklist of things to look for while developing dialog boxes.*



•
•
•

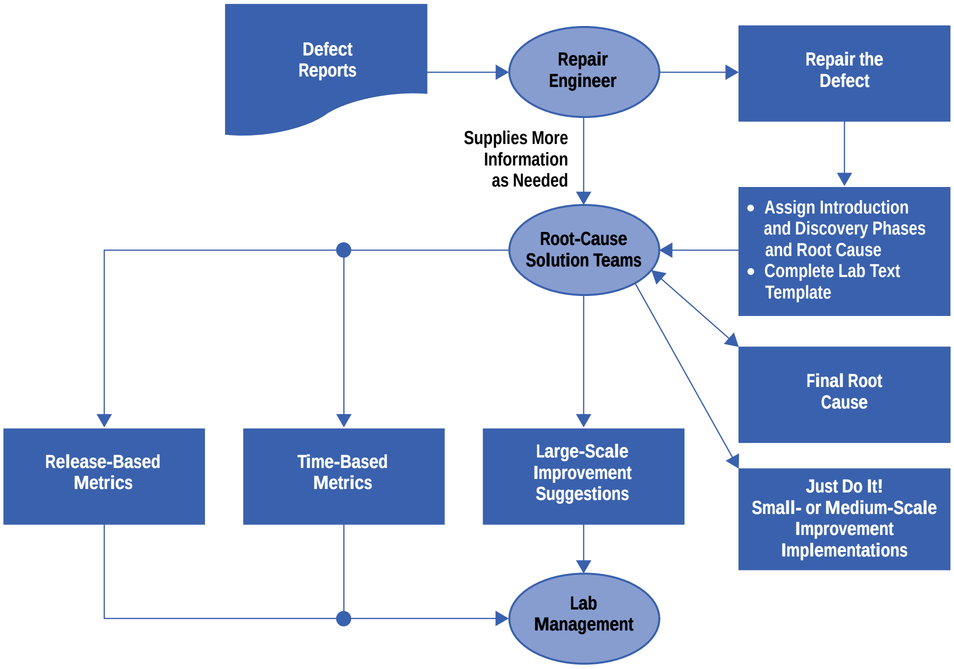
7. Are fields case sensitive or not? What implications are there?
8. Are abbreviations kept to a minimum?
9. Are there any spelling mistakes on the panel?
10. Does the panel have a title that matches the action of the panel?
11. Is the screen too crowded? For data entry, less than 50 percent of the panel should be writing. Controls should "fill" the panel without cluttering it.
12. Is help available to the user? Is there a help line to aid the user in understanding the field?
13. Has the help writer been updated with information on the new panel?
14. Are the units for edit fields given when appropriate?

•
•
•

- They reduced the percentage of user-interface defects in test for their new year-long project to roughly five percent of their total system test defects.
- Even though the project produced 34 percent more code, they spent 27 percent less time in test.

Of course, other improvement efforts also contributed to their success. But the clear user interface defect reduction showed them that their new guidelines and the attention they paid to their interfaces were major contributors.[8] Finally, the best news is that customers were very pleased with the user interface, and initial product sales were very good.

Fig. 10. *Root-cause analysis process.*


Defect Reports
Repair Engineer
Repair the Defect
Supplies More Information as Needed
Root-Cause Solution Teams
• Assign Introduction and Discovery Phases and Root Cause
• Complete Lab Text Template
Final Root Cause
Release-Based Metrics
Time-Based Metrics
Large-Scale Improvement Suggestions
Just Do It! Small- or Medium-Scale Improvement Implementations
Lab Management

Two other project teams finished their projects recently, and their results were equally impressive. Both projects used new standard divisional specification templates created to eliminate many of the root causes shown in Fig. 8. A cross-project team task force had created two two-page specification templates (one for user-interface-oriented routines, one for software-interface-oriented ones) that they felt would help. Both teams substantially reduced specification defects compared with their previous project levels. While the reason for one team's reduction could possibly be that the project was second-generation, the other project wasn't.

While the action steps discussed here follow those of successful improvement projects at one HP division, they can also be applied in organizations with different defect patterns and business needs. One of the division people who worked with all three project teams summarized their results:
" . . . We must conclude that the root-cause approach is an effective mechanism to identify and introduce change into our software development process."[9]

Continuous Process Improvement Cycle

Some organizations have felt that root-cause analysis is so beneficial that they now use it to pursue continuous process improvement. It appears to be a natural evolution from post-process root-cause analysis successes. This approach extends the supporting infrastructure and requires an ongoing management commitment.

The first step that an organization generally takes is to widely adopt root-cause information logging by engineers. Causal information is then included as a normal part of the defect-handling process. Analysis is triggered in a variety of ways, often by a product or system release. Sometimes it is triggered by the end of a development phase or a series of inspections. It can also be triggered by an arbitrary time period. Fig. 10 shows how one HP division runs its process. Root-cause solution teams are empowered by management to initiate smaller process improvements.[10] More far-reaching improvements still require lab management approval.

Knowing which defects occur most often in test or later helps to focus improvement efforts. We saw two examples of this in the post-project root-cause analysis discussion. The continuous process improvement cycle encourages examination of similar data throughout the development process. Take the HP division whose test data was shown as the lower-right pie chart in Fig. 4. It also captured data for specifications, design, and code inspections. All this data is shown in Fig. 11. Some caution should be used in interpreting this specific data, since it was not uniformly collected. For example, there may have been a higher percentage of design work products than code work products, but still less than there was code tested. Nevertheless, this figure suggests some interesting questions and reveals possible insights.

The bars above the centerline show counts for different defects that were found in the same phase in which they were created. Tall bars represent good opportunities to reduce these defect sources significantly. For example, the large number of module design defects suggests that a different design technique might be needed to replace or complement existing methods.

The bars below the line show counts for defects found in phases after the ones in which they were created. The later defects are found, the more expensive they are to fix. Therefore, the tall bars are sources of both better prevention and earlier detection opportunities. For example, the requirements, functionality, and functional description defects combine to suggest that designs may be changing because of inadequate early product definition. It might be useful to use prototypes to reduce such changes.

It is clear that this type of data can contribute to more informed management decisions. It also provides a way of evaluating the results of changes with better precision than in the past. The amount of effort required to sustain a continuous process improvement cycle will vary, depending largely on the cost of implementing the changes suggested by analyses. Which changes are chosen for implementation will depend on other business aspects besides the projected costs and benefits. Just remember that the cost to sustain failure-analysis practice and modest improvements is small, and the returns have proven to far outweigh those costs.[2,5,8]

Conclusion

Process improvement projects are started in many ways, for many reasons. In the software field especially, processes are changing and adapting daily, and software products and businesses are also rapidly evolving. One of the most effective ways to both motivate and evaluate the success of net improvements is to look at defect trends and patterns. This paper has shown how software defect data is a powerful management information source. Using it effectively will help achieve an optimal balance between reacting to defect information and proactively taking steps toward preventing future defects. HP divisions have used several successful approaches to handling defect causal data. The three root-cause analysis processes described in this paper are positioned against a suggested five-level maturity model shown in Fig. 12.

Like many other best practices, failure analysis can be applied with increasing levels of maturity that lead to different possible paybacks. HP's experience says that the biggest benefits of driving to higher maturity levels are:

- Increased likelihood of success when implementing process changes, particularly major ones
- Accelerated spread of already-proven best practices
- Increased potential returns because necessary infrastructure components are in place.

Our successful results from three failure-analysis approaches are very encouraging. While the time it takes to progress to higher maturity levels will vary among groups, our experience suggests that failure analysis starts providing returns almost immediately, particularly in visualizing progress.

Ironically, the main limiter to failure-analysis success is that many managers still believe that they can quickly reduce total effort or schedules by 50 percent or more. As a result, they won't invest in more modest process improvements. This prevents them from gaining 50 percent improvements through a series of smaller gains. Because it takes time to get any improvement adopted organization-wide, these managers will continue to be disappointed.

It has not been difficult to initiate use of the Fig. 3 defect model and the root-cause analysis process. The resulting data has led to effective, sometimes rapid, improvements. There are few other available sources of information that are as useful in identifying key process weaknesses specific to an organization. This information will help to drive process improvement decisions and commitment in an organization.

Fig. 11. *A defect profile, an interesting way of analyzing defect data during the continuous process improvement phase.*

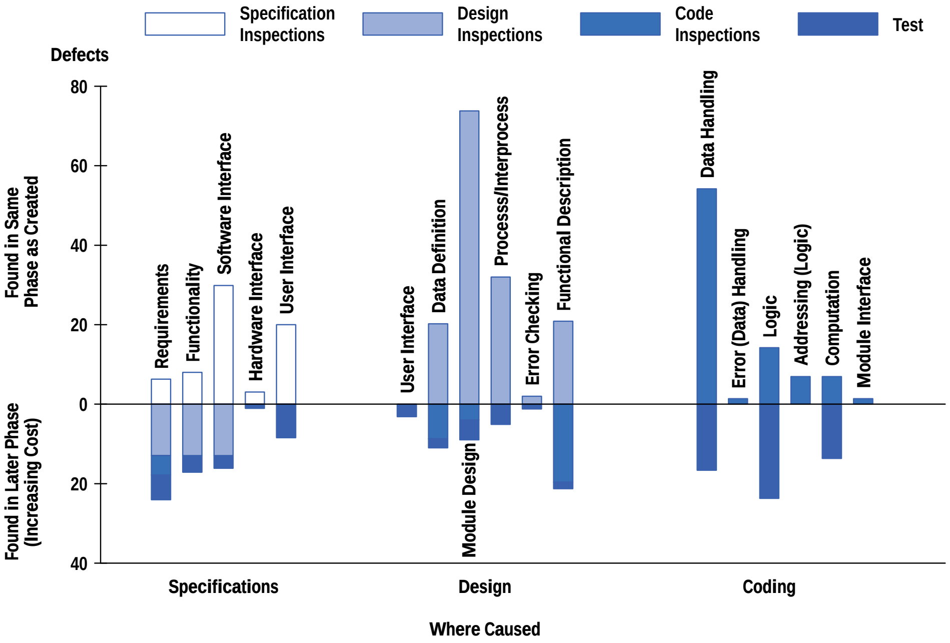
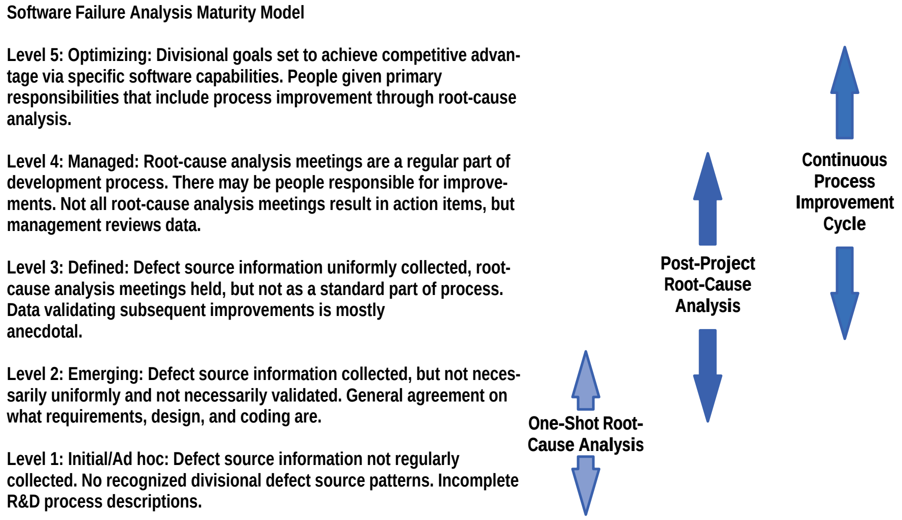


Fig. 12. *A five-level software failure-analysis maturity model.*


Software Failure Analysis Maturity Model

Level 5: Optimizing: Divisional goals set to achieve competitive advantage via specific software capabilities. People given primary responsibilities that include process improvement through root-cause analysis.

Level 4: Managed: Root-cause analysis meetings are a regular part of development process. There may be people responsible for improvements. Not all root-cause analysis meetings result in action items, but management reviews data.

Level 3: Defined: Defect source information uniformly collected, root-cause analysis meetings held, but not as a standard part of process. Data validating subsequent improvements is mostly anecdotal.

Level 2: Emerging: Defect source information collected, but not necessarily uniformly and not necessarily validated. General agreement on what requirements, design, and coding are.

Level 1: Initial/Ad hoc: Defect source information not regularly collected. No recognized divisional defect source patterns. Incomplete R&D process descriptions.

Continuous Process Improvement Cycle
Post-Project Root-Cause Analysis
One-Shot Root-Cause Analysis

Acknowledgments

I'd like to thank Jan Grady, Debbie Caswell, Cate Meyers, Barbara Zimmer, Dell Fields, Tom Van Slack, and Jean MacLeod for their helpful suggestions in the development of this article. Finally, thanks to Brad Yackle, Marc Tischler, and others at HP's Scientific Instrument Division for sharing their failure-analysis results.

References

1. M. Paulk, B. Curtis, M. Chrissis, and C. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software,* July 1993, pp. 18-27.
2. R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Inc., 1992, pp. 37, 79, 129, 130, 137-157.
3. G. Stark, G., L. Kern, and C. Vowell, "A Software Metric Set for Program Maintenance Management," *Journal of Systems and Software 24*, 1994, pp. 239-249.
4. D. Clark, "Change of Heart at Oracle Corp.," *San Francisco Chronicle*, July 2, 1992, pp. B1 and B4.
5. R. Grady, "Practical Results from Measuring Software Quality," *Proceedings of the ACM*, Vol. 36, no. 11, November 1993, pp. 62-68.
6. K. Ishikawa, *A Guide to Quality Control*, Tokyo: Asian Productivity Organization, 1976.
7. M. Brassard, *The Memory Jogger Plus+*, GOAL/QPC, 1989.
8. R. Grady, ,"Successfully Applying Software Metrics," *IEEE Computer*, September 1994, pp. 18-25.
9. M. Tischler, e-mail message, Aug. 10, 1994.
10. D. Blanchard, "Rework Awareness Seminar: Root-Cause Analysis," March 12, 1992.

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 text[1] 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

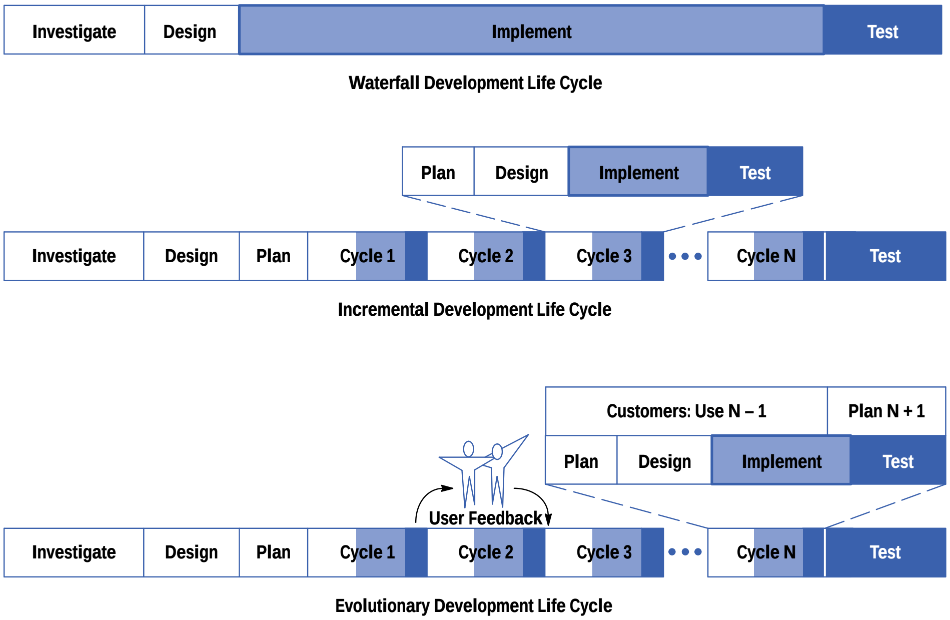
The traditional waterfall life cycle for software development has served software developers well. By breaking software projects up into several large sequential phases—typically an investigation or definition phase, a design phase, an implementation phase, and a test phase—project teams could move forward 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's[2] 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 Rumbaugh[3] 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.*

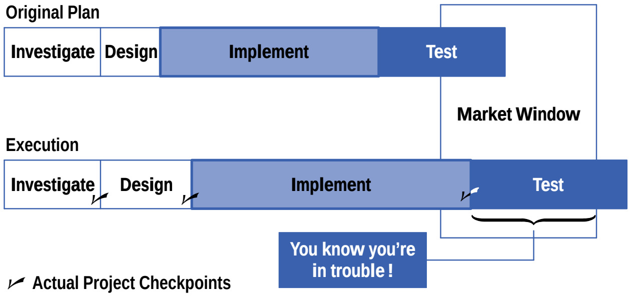
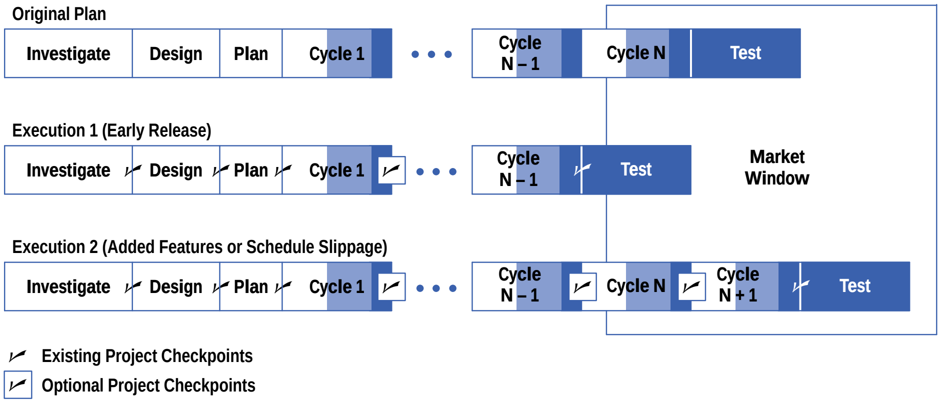


Fig. 3. *Hitting market windows with an evolutionary life cycle.*



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.

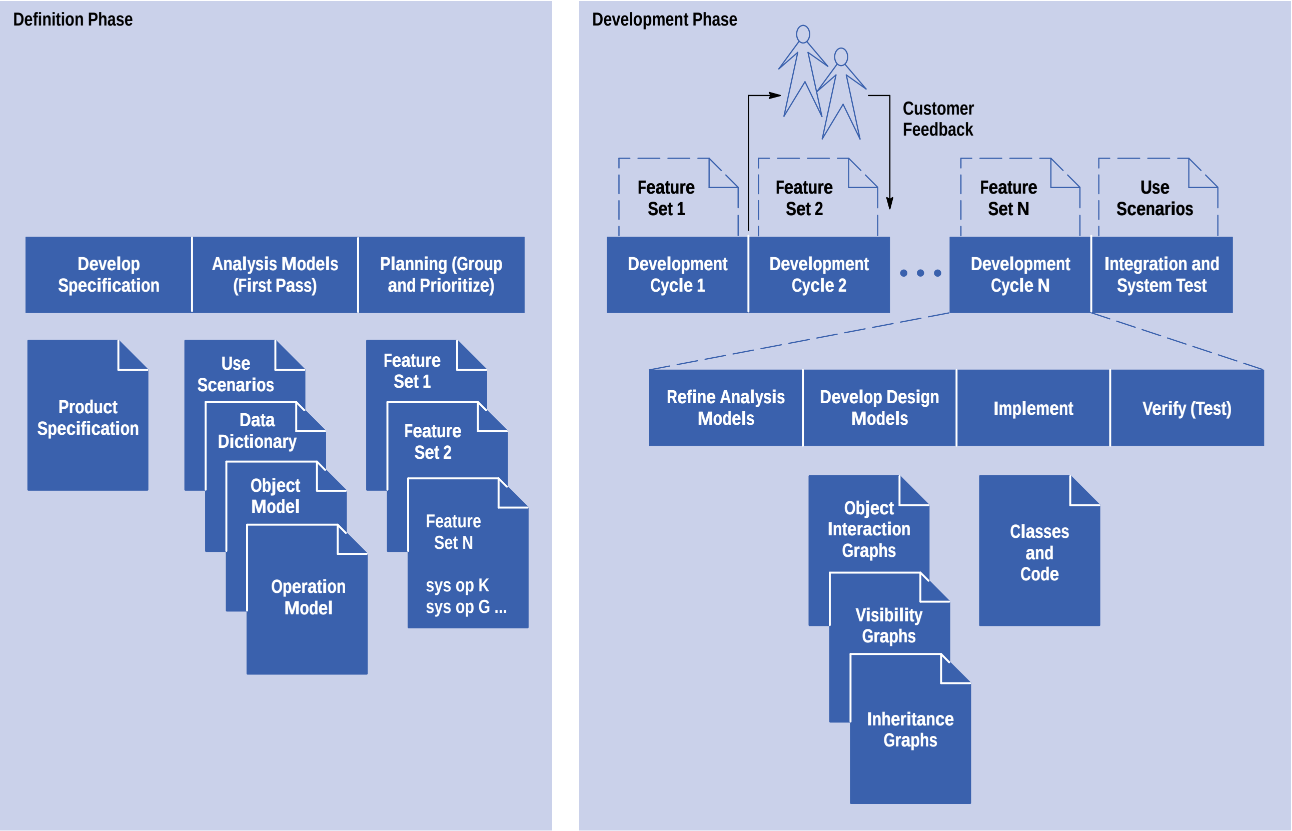
Evolutionary Fusion divides a project into two major phases: the definition phase and the development phase (Fig. 4). During the definition phase, a project's functionality is specified and its viability as a product or system is first estimated. The Fusion analysis models play a key role in this phase. 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

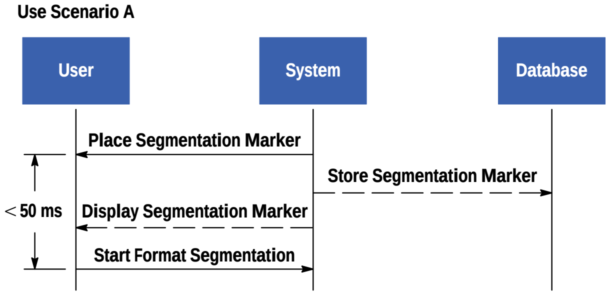
Like Fusion, Evolutionary Fusion requires some form of system specification as a starting point, and just about any level of detail in the system specification will do. When the specification is at a high level, the analysis models serve to identify large numbers of issues and questions that need to 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.*


Use Scenario A
User
System
Database
Place Segmentation Marker
Store Segmentation Marker
< 50 ms
Display Segmentation Marker
Start Format Segmentation

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 Ould[9] 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.

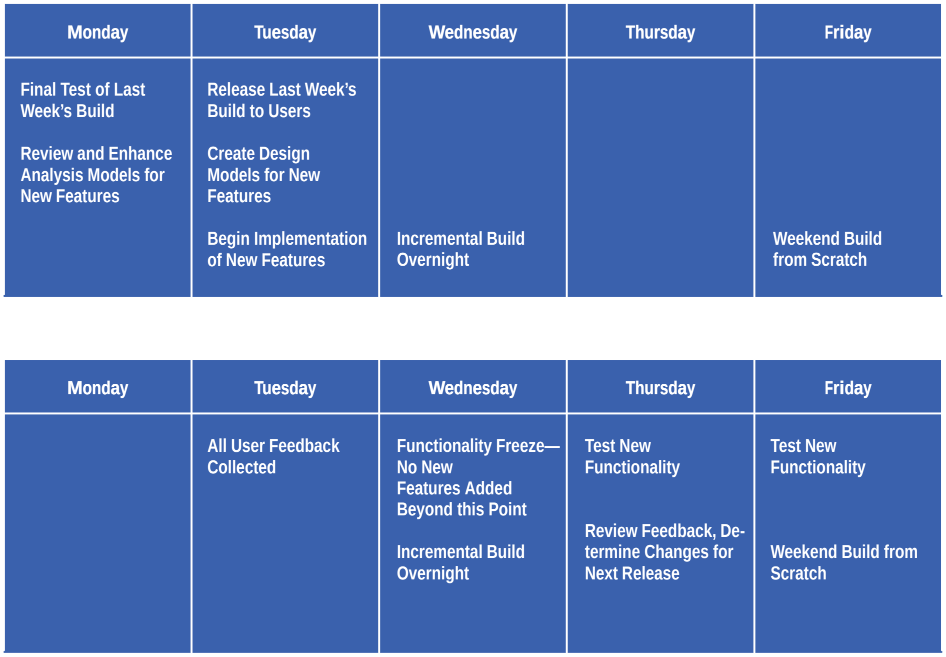
Project manager: Many aspects of the project manager's role become even more critical with Evolutionary Development. The project manager must work with the marketing team and the customers to establish the project's value proposition, identify key project risks, document all commitments and dependencies, 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 Review and Enhance Analysis Models for New Features	Release Last Week's Build to Users 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 Incremental Build Overnight	Test New Functionality Review Feedback, Determine Changes for Next Release	Test New Functionality 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.

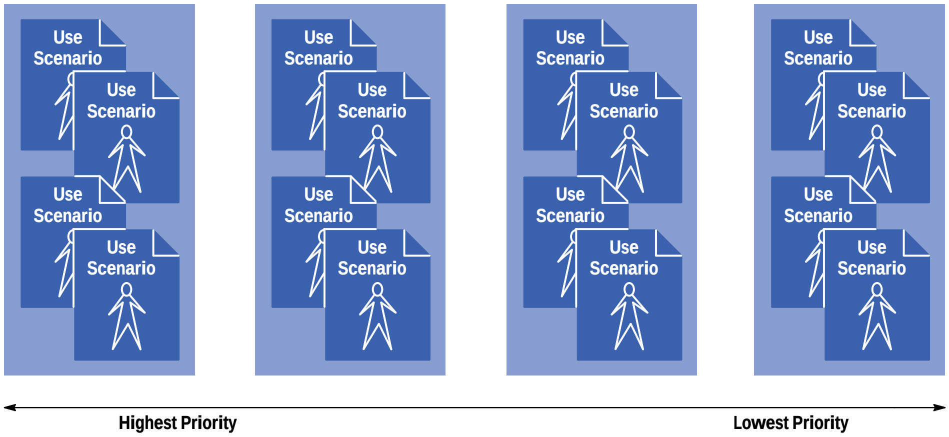
Grouping and Prioritizing Functionality. With key roles assigned and the standard cycle defined, the last step in planning the development phase is to group and prioritize the functionality into implementation chunks. The chunks must be no larger than can be delivered in the standard cycle time. Prioritization 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.*



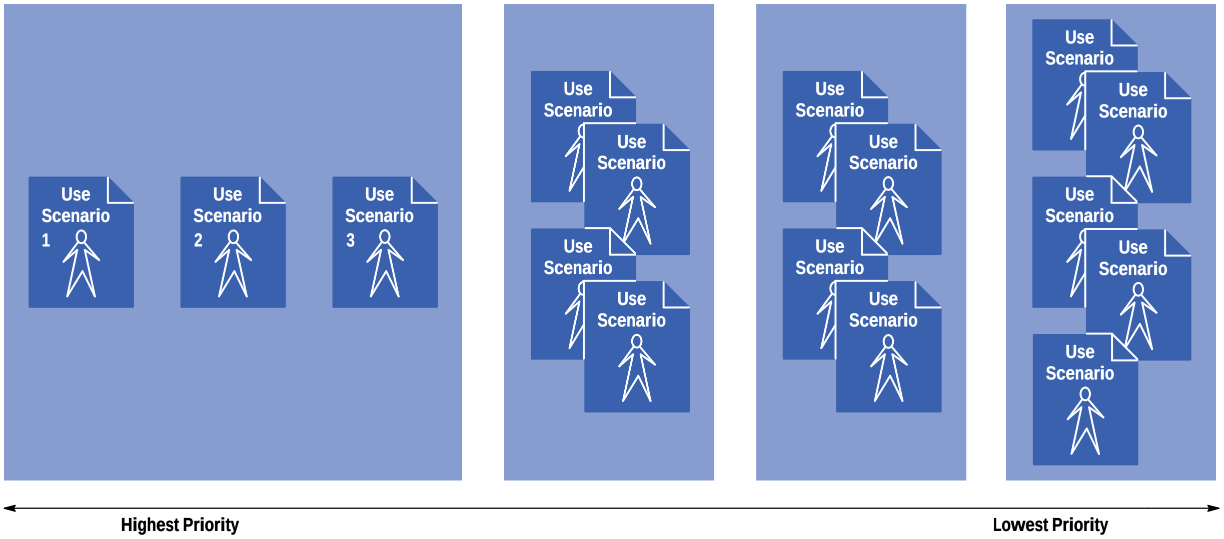
The system operations from the use scenarios in the first group can now be grouped and sequenced into the first few implementation cycles (Fig. 9). Keep in mind that the deliverables from each cycle should be defined in such a way that they can be validated by a user of the system. For these early cycles, 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.*

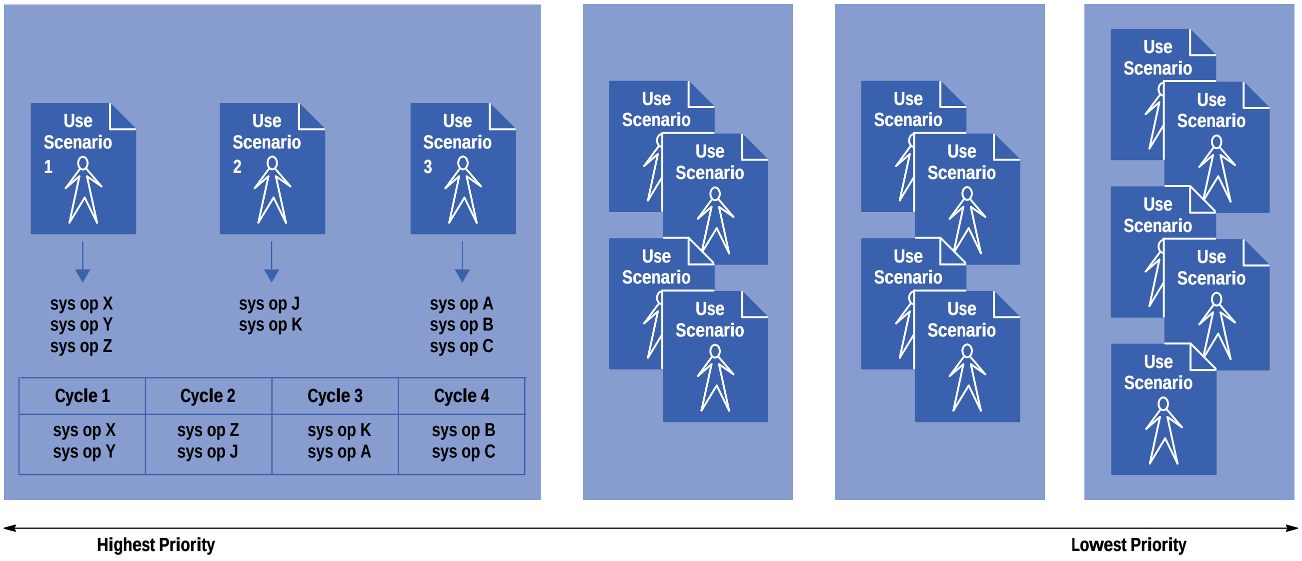


there are typically many members of the organization that prefer to see just the "big picture." The themes can help convey that big picture.

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.

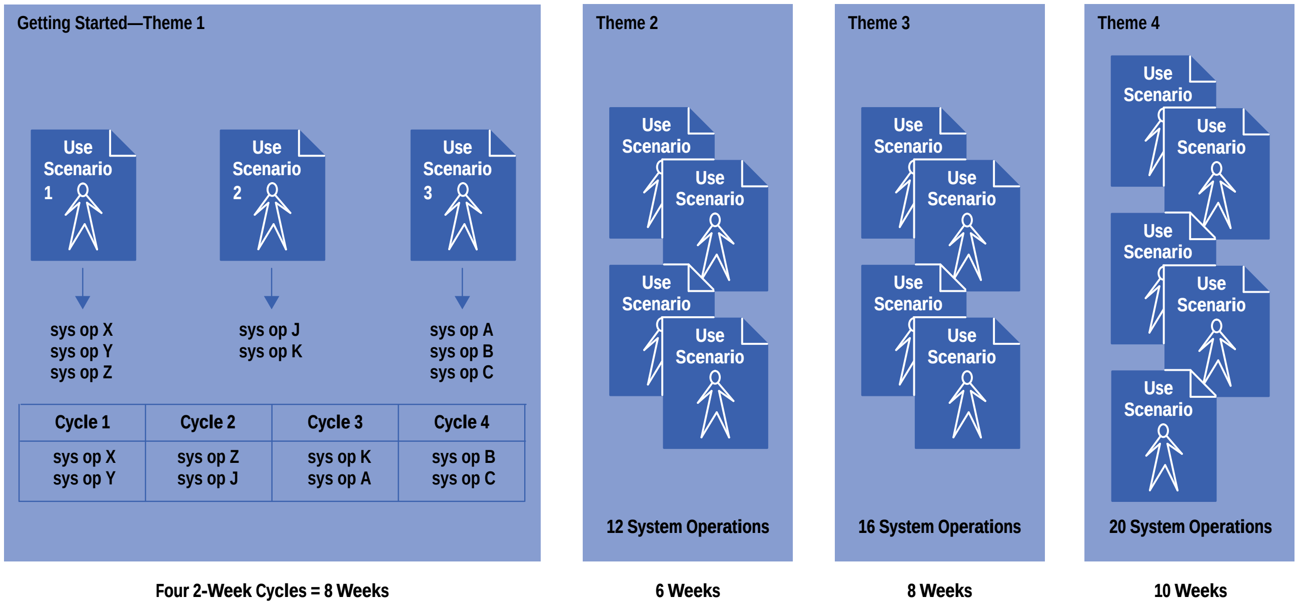
It is not necessary to provide any additional detail for the groupings of use scenarios beyond the first. It is only necessary to make sure that all functionality as it is defined at this early stage is accounted for and that an overall estimate of the effort is calculated. It is expected that experiences 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.*



Cycle 1	Cycle 2	Cycle 3	Cycle 4
sys op X sys op Y	sys op Z sys op J	sys op K sys op A	sys op B sys op C

Fig. 10. *Completed implementation plan.*



Cycle 1	Cycle 2	Cycle 3	Cycle 4
sys op X sys op Y	sys op Z sys op J	sys op K sys op A	sys op B sys op C

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.

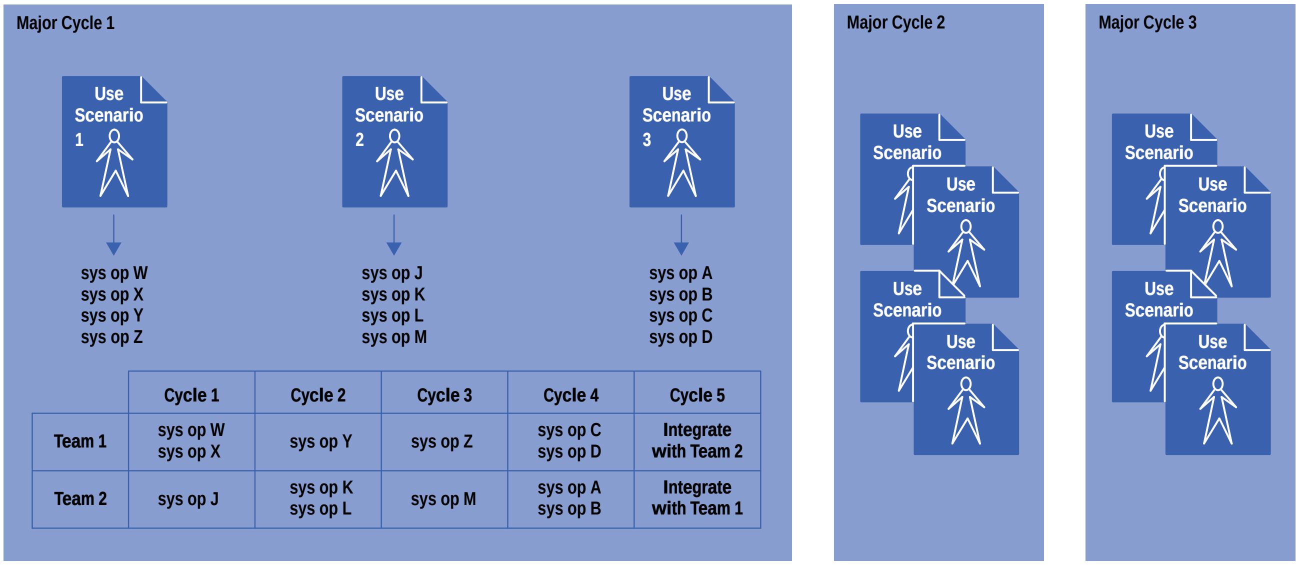
The technical lead is a key player during the refinement of the analysis models. Because they represent the overall architecture for the system, any extensions or enhancements of the models must be made without serious compromise to the integrity of the architecture. If compromises must be made, they 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.



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
Team 1	sys op W sys op X	sys op Y	sys op Z	sys op C sys op D	Integrate with Team 2
Team 2	sys op J	sys op K sys op L	sys op M	sys op A sys op B	Integrate with Team 1

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.

What is Fusion?

Fusion is a systematic software development method for object-oriented software development. Developed at Hewlett-Packard Laboratories in Bristol, England, the method integrates and extends the best features of earlier object-oriented methods. Fusion is a full-coverage method, providing a direct route from a requirements definition through analysis and design to a programming language implementation.

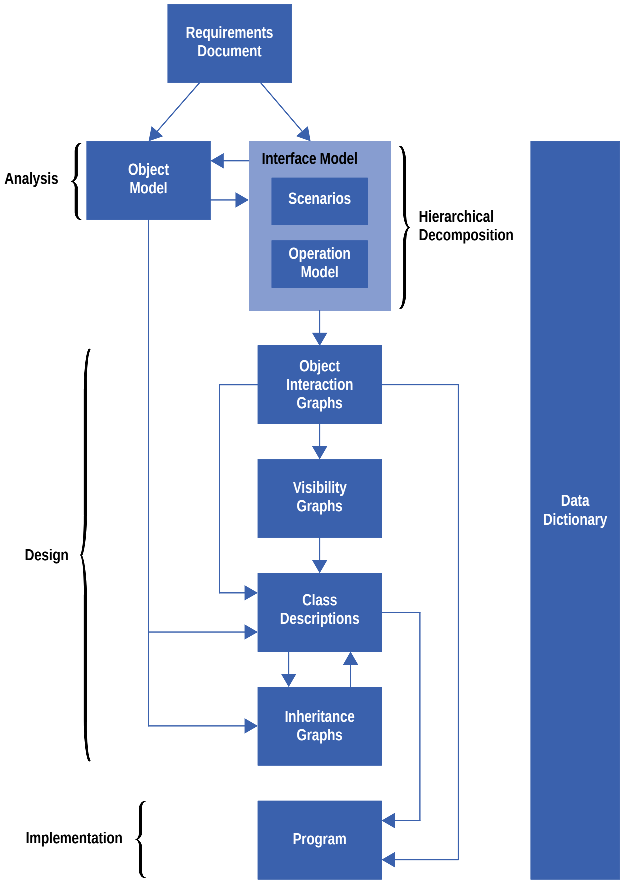
What Fusion Offers

- It provides a process for software development. It divides this process into phases and says what should be done in each phase. It gives guidance on the order in which things should be done within phases so that the developer knows how to make progress. It provides criteria that tell the developer when to move on to the next phase.
- It provides a comprehensive, simple, well-defined notation for all of its models. Because this notation is based on existing practices, it is easy to learn.
- It provides management tools for software development. The outputs of the different phases are clearly identified, and there are cross-checks to ensure consistency within and between phases. Each phase has its own techniques and addresses different aspects of translating a requirements document into executable code.
- It is adaptable. A lightweight version can be used in projects that cannot afford the effort required to use the full version, or parts of the process or notation can be used within other development processes to address their weak points.

The Process

The Fusion method structures the development process into analysis, design, and implementation phases (see Fig. 1).

Fig. 1. *The Fusion process.*



Analysis

During the analysis phase the analyst defines the intended behavior of the system. Models of the system are produced, which describe:

- The classes of objects that exist in the system
- The relationships between those classes

- The operations that can be performed on the system
- The allowable sequences of those operations.

Design

The designer chooses how the system operations are to be implemented by the run-time behavior of interacting objects. Different ways of breaking an operation into interactions can be tried. During this process, operations are attached to classes. The designer also chooses how objects refer to each other and what the appropriate inheritance relationships are between classes.

The design phase delivers models that show:

- How operations on the system are implemented by interacting objects
- How classes refer one to another and how they are related by inheritance
- The attributes of and operations on classes.

Designers may need to investigate the substructure of some classes and their operations in more detail. They do so by applying the analysis and design techniques to those classes, regarding them as a subsystem.

Implementation

The implementer must turn the design into code in a particular programming language. Fusion gives guidance on how this is done in the following ways:

- Inheritance, reference, and class attributes are implemented in programming-language classes.
- Object interactions are encoded as methods belonging to a selected class.
- The permitted sequences of operations are recognized by state machines.

Fusion also maintains a data dictionary, a place where the different entities of the system can be named and described. The data dictionary is referenced throughout the development process.

In summary, Fusion is a complete, yet lightweight development method that can be tailored to meet the different needs of software projects.

Derek Coleman
Professor of Computer Science
University of London

Fusion in the Real World

The use of Fusion has spread rapidly since its introduction in 1993. Today, it is the most widely used object-oriented analysis and design method in Hewlett-Packard. Many other companies worldwide are also employing the method on a wide variety of applications and products.

Fusion is a living method that is being extended and evolved based on lessons learned in the real world. Todd Cotton's work on Evolutionary Fusion, described in the accompanying article, is an exemplary illustration of how Fusion benefits from the collaboration of a broad community of users, consultants, and researchers. The book listed in reference 1 was created to provide a forum for articulating and disseminating such contributions to the method. It does this by collecting together reports from the field that describe the practical lessons that have been learned from projects using Fusion. Todd Cotton and other contributors combine their expertise to give the most comprehensive look yet at how Fusion is changing the world of object-oriented development. Throughout the book the emphasis is on practicality and lessons learned. The main themes of the book include:

- An introductory overview of Fusion together with full reference documentation
- Detailed experience reports of industrial projects discussing how to introduce Fusion to a project and how to succeed using it
- An account of how to reduce risk by integrating Fusion into an evolutionary life cyle
- A report on metrics and defect tracking in a Fusion project
- Lessons learned from a wide variety of applications and backgrounds, including product development organizations, research laboratories, academia, software houses, and consultancies.

Ruth Malan
Software Engineer
Hewlett-Packard Laboratories

Reed P. Letsinger
Project Manager
Hewlett-Packard Laboratories

Reference

1. R. Malan, R. Letsinger, and D. Coleman, Editors, *Object-Oriented Development at Work: Fusion in the Real World*, Prentice Hall/HP Press, 1996.

The Evolutionary Development Model for Software

The traditional waterfall life cycle has been the mainstay for software developers for many years. For software products that do not change very much once they are specified, the waterfall model is still viable. However, for software products that have their feature sets redefined during development because of user feedback and other factors, the traditional waterfall model is no longer appropriate.

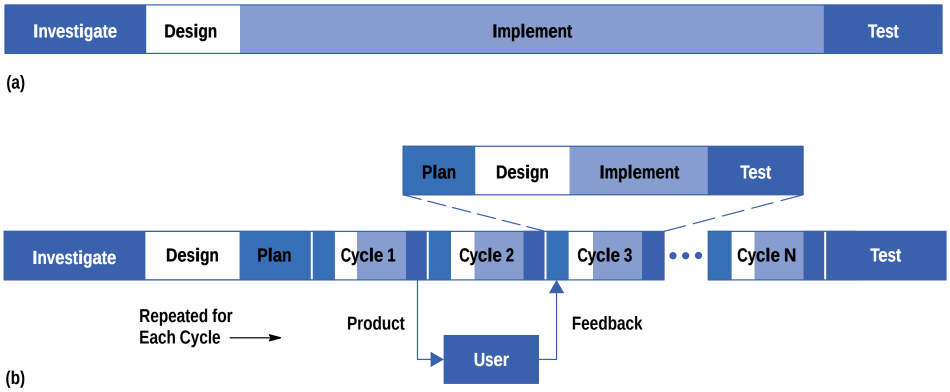
by Elaine L. May and Barbara A. Zimmer

Hewlett-Packard, like other organizations developing software products, is always looking for ways to improve its software development processes. One software development method that has become quite popular at HP is called *Evolutionary Development*, or EVO (see reference 1 and ***Article 3***). EVO uses small, incremental product releases, frequent delivery to users, and dynamic plans and processes. Although EVO is relatively simple in concept, its implementation at HP has included both significant challenges and notable benefits. This paper begins with a brief discussion about the EVO method and its benefits, then describes software projects at three HP divisions that have used EVO, and finally discusses critical success factors and key lessons about EVO.

The EVO Method

Fig. 1 shows the difference between the traditional waterfall life cycle and the EVO life cycle. The EVO development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle. The users provide feedback on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plans, or process. These incremental cycles are typically two to four weeks in duration and continue until the product is shipped.

Fig. 1. *Software development life cycles. (a) Traditional waterfall model. (b) Evolutionary (EVO) development model.*



At Hewlett-Packard, we have found that it is possible to relax some of our original ideas regarding EVO. In particular, it isn't absolutely necessary to deliver the product to external customers with customer-ready documentation, training, and support to benefit from EVO.

Benefits of EVO

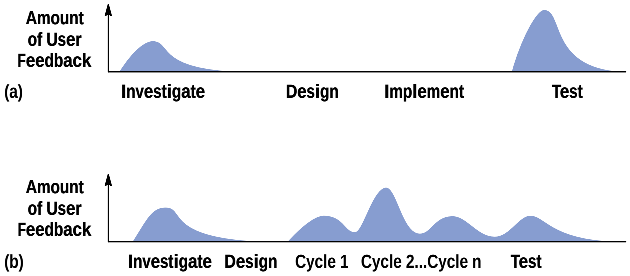
Successful use of EVO can benefit not only business results but marketing and internal operations as well. From a business perspective, the biggest benefit of EVO is a significant reduction in risk for software projects. This risk might be associated with any of the many ways a software project can go awry, including missing scheduled deadlines, unusable products, wrong feature sets, or poor quality. By breaking the project into smaller, more manageable pieces and by increasing the visibility of the management team in the project, these risks can be addressed and managed.

Because some design issues are cheaper to resolve through experimentation than through analysis, EVO can reduce costs by providing a structured, disciplined avenue for experimentation. Finally, the inevitable change in expectations when users begin using the software system is addressed by EVO's early and ongoing involvement of the user in the development process. This can result in a product that better fits user needs and market requirements.

EVO allows the marketing department access to early deliveries, facilitating development of documentation and demonstrations. Although this access must be given judiciously, in some markets it is absolutely necessary to start the sales cycle well before product release. The ability of developers to respond to market changes is increased in EVO because the software is continuously evolving and the development team is thus better positioned to change a feature set or release it earlier.

Short, frequent EVO cycles have some distinct advantages for internal processes and people considerations. First, continuous process improvement becomes a more realistic possibility with one-to-four-week cycles. Second, the opportunity to show their work to customers and hear customer responses tends to increase the motivation of software developers and consequently encourages a more customer-focused orientation. In traditional software projects, that customer-response payoff may only come every few years and may be so filtered by marketing and management that it is meaningless. Fig. 2 illustrates the difference between the traditional life cycle and EVO in terms of how much user feedback can be expected during product development.

Fig. 2. *Amount of user feedback during (a) the traditional waterfall development process and (b) the evolutionary development process (EVO).*



Finally, the cooperation and flexibility required by EVO of each developer results in greater teamwork. Since scheduling and dependency analysis are more rigorous, less dead time is spent waiting on other people to complete their work.

While the benefits can be substantial, implementation of evolutionary development can hold significant challenges. It requires a fundamental shift in the way one thinks about managing projects and definitely requires more management effort than traditional software development methods. The next section examines how EVO was applied in three different HP divisions and what we learned from the experience.

Evolutionary Development in Practice

Some form of EVO has been used in at least eight Hewlett-Packard divisions in over ten major projects. Much of this has been done drawing on expertise from HP's Corporate Engineering software initiative, which is a central service group of consultants in software engineering and management (see ***Sidebar***). The software initiative group is currently leveraging existing experience and promoting the use of EVO at HP.

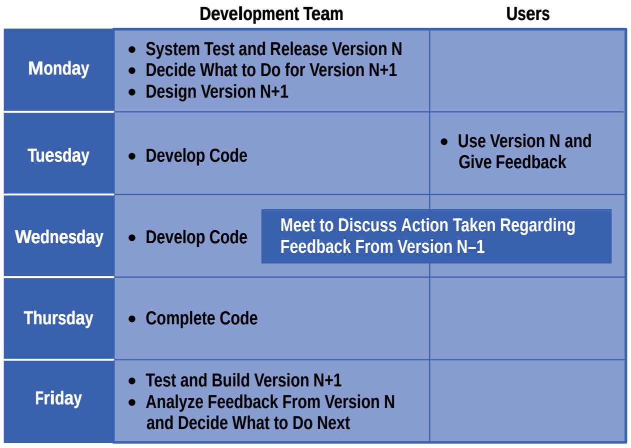
The three divisions described below are in three entirely different businesses. While all the product names used in this paper are fictitious, the case descriptions are real.

First Attempts

The first project was undertaken at HP's Manufacturing Test Division. The project (called project A here) consumed the time of four software developers for a year and a half and eventually was made up of over 120,000 lines of C and C++ code. Over 30 versions were produced during the eleven-month implementation phase which occurred in one- and two-week delivery cycles (see Fig. 3). The primary goals in using EVO were to reduce the number of late changes to the user interface and to reduce the number of defects found during system testing.

Project A adapted Gilb's EVO methods.[1] One departure was the use of surrogate users. The Manufacturing Test Division produces testers that are used in manufacturing environments. If the tester goes down, the manufacturer cannot ship products. Beta sites, even when customers agree to them, are carefully isolated from production use, so the beta software is rarely, if ever, exercised. Fortunately, the project had access to a group of surrogate users: application engineers in marketing and test engineers in their own manufacturing department. The use of surrogates did not appear to have any negative impact.

Fig. 3. An example of a typical one-week EVO cycle at the Manufacturing Test Division during project A.



	Development Team	Users
Monday	• System Test and Release Version N • Decide What to Do for Version N+1 • Design Version N+1	
Tuesday	• Develop Code	• Use Version N and Give Feedback
Wednesday	• Develop Code	Meet to Discuss Action Taken Regarding Feedback From Version N–1
Thursday	• Complete Code	
Friday	• Test and Build Version N+1 • Analyze Feedback From Version N and Decide What to Do Next	

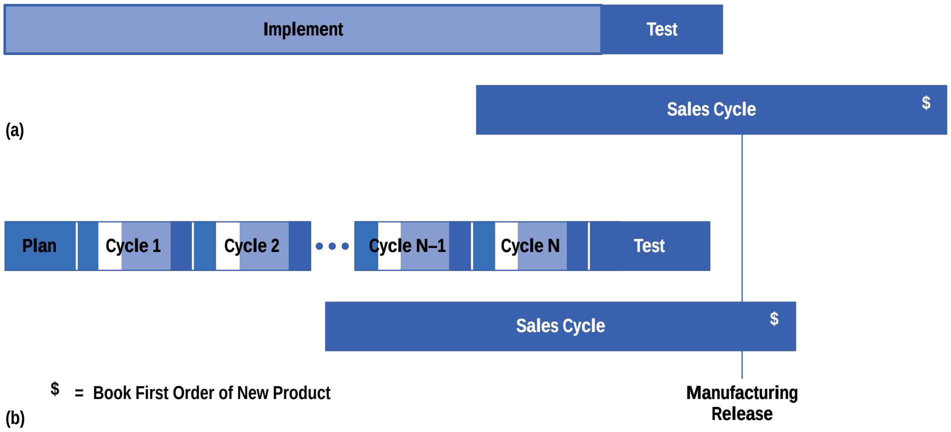
About two thirds of the way through the project, the rigorous testing and defect fixing that had been done during the EVO cycles was discontinued because of schedule pressures. The cost of this decision was quality. With all efforts focused on finishing, developers began adding code at a rate double that of previous months, and over half of the critical and serious defects were introduced into the code in the last third of the project schedule.

Even though EVO was not used to complete the project, the product was successful and the team attributed several positive results to having used the EVO method for the majority of the project. First, EVO contributed to creating better teamwork with users and more time to think of alternative solutions. Second, the project still had significantly fewer critical and serious defects during system testing. Third, the team was surprised to see an increase in productivity (measured in KNCSS per engineer-month). The project manager attributes this higher productivity primarily to increased focus on project goals.

Despite having abandoned the EVO method in project A, many in the division felt that because of the benefits derived from the method, they should give it another try. Project B, the second project to use the EVO method, involved creating custom hardware with a team of three project managers and 20 engineers. The project required significant changes to over 1.5 million lines of code. One project manager coordinated the efforts of three development teams.

The primary reason for using EVO for this project was to demonstrate the feasibility of the product's new test technique through the use of beta sites. Internal users were used early in the cycle and external customers became involved with later versions. This decision resulted in actually selling systems to beta-site customers. Further, the traditionally long startup time for the division's sales cycle was shortened significantly by the use of EVO and validation of the product by users (see Fig. 4). This created a major business impact since it typically takes nine to fifteen months before the market believes a product of this type can live up to its claims. Even more time passes before customers will actually buy the product. Because EVO encourages early exposure of a product to users, sales started even before the product shipped!

Fig. 4. An accelerated sales cycle in (a) the traditional waterfall life cycle and (b) the EVO cycle.



Start Small

The experience of HP's Personal Software Division with EVO also began with some startup problems, followed by remarkable success. The first project to use EVO was chosen because it was felt that EVO would help to prioritize new features, respond quickly to customer needs, and, because of EVO's many release cycles, enable the release of the software product at any time in response to competition.

The six to eight project managers and approximately 60 engineers on this project were all new to the EVO method. The plan was to do a complete release, including customerready documentation and support, every month. Unfortunately, the first release consisted of paper prototypes and the users were not able to provide good feedback. The second release used real code, took six weeks rather than the four weeks scheduled, and EVO was generally thought not to be worth the integration and logistical effort. For this reason, the division decided to abandon EVO.

Although EVO had a bad reputation at the division after the first project, in a smaller follow-on project, one project manager and eight engineers decided to try their own variation of EVO, calling it "phased development." During the first one-month phase, 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 showed a demonstration of its prototype. The excellent feedback from these focus groups had a large impact on the quality of the product.

After the second cycle of focus groups, the feature set was frozen and the product definition complete. Implementation consisted of four-to-six-week cycles, with software delivered for beta use at the end of each cycle. The entire release took 10 months from definition to manufacturing release. Implementation lasted 4.5 months. The result was a world-class product that has won many awards and has been easy to support.

The success of phased development for this second product led to the use of a similar process in the second release. The project manager concluded that the phased development process was the best approach for projects with an aggressive, user-driven schedule. Team experience and confidence were definite contributing factors to the product's success, and a compelling product vision proved to be absolutely necessary.

Several potential issues arose during the project. EVO can add overhead, particularly in small one- or two-person components. This is mainly because of the need for rapid context switching between various activities. Another potential problem is the amount of time consumed by evaluation. The team is investigating how to make evaluation feedback more timely. A third issue is the need to schedule enough time for front-end activities like design and inspections. Scheduling longer evaluation cycles at the beginning of a new release could accommodate this, as could setting aside intermediate cycles for design, inspections, and code cleanup.

The project postmortem listed a number of benefits from using EVO. The team particularly liked seeing the results of their work often. Other benefits included:

- Long-term vision broken into short-term steps
- Prioritized implementation within component teams
- Cross-functional, empowered component teams (decision making pushed down to the project engineers)
- Early results—good communication tool inside and outside the division
- External customer feedback
- Six-week planning at the system level
- Excellent for incremental improvements to existing products
- Early realism about how much can be done.

A New Platform

The last project described in this article involved one project manager and eight engineers from HP's Microwave Instrument Division. It was a firmware project to build a platform for developing new products.

Since a reusable platform design was a new way of working for the division, the EVO method was expected to get more visibility (via frequent delivery dates) for construction of the prototype. The platform team expected to get good feedback from the teams developing the follow-on projects. The project manager had strong reservations about using EVO and about being able to produce a verifiable slice of the platform architecture in six weeks. The project manager decided to give EVO a try even if it seemed that it would take ten weeks to complete the feasibility demonstration work. The team completed the bulk of the feasibility work six weeks into the project and finished it all by the ninth week.

Some engineers initially struggled with breaking their work down into two-week chunks. Eventually, they not only learned to do it but saw some real value in doing it, such as getting better estimates so they could meet their commitments and handle coordination and linkages with the rest of the team.

The team members also benefited from using EVO to develop their software development environment. Because of the improved infrastructure, the project team was able to train new engineers quickly on the new platform development paradigm. The project manager reported much greater insight into the progress of the team and felt better able to manage the project. EVO helped to uncover key issues early and focus attention on the right things. One-third of the way through the project, the team was able to verify and meet their first performance goals. Traditionally, this didn't happen until at least halfway through a project.

Unfortunately, after completing more than half of the planned cycles, the project was cancelled because of a shift in the division's short-term R&D strategy. The lab currently plans to use EVO in two new projects.

Critical Success Factors

Based on accumulated HP experience in evolutionary development, including the three division experiences described above, we have compiled a list of critical success factors. Since not every project is suited for evolutionary development, the following success factors provide some indicators for deciding if a project is a good candidate for EVO.

Clear Vision

Perhaps the most critical success factor in using EVO is having a clear and compelling vision of the product. The perceived vision or value of the product is the reason why someone would buy a given product rather than another, or buy no product at all. Whether adding incremental functionality to an existing product or developing major new components or functionality, the project team needs to understand and accept this vision.

This vision will help guide prioritizing and decision making and will make it easier for users and developers to understand why some changes are approved and others are not. The project manager at the Personal Software Division noted that the lack of clear focus for the second version of the product made the project much more difficult to manage than the first release. A clear vision is critical to convergence on a releasable product.

Project Planning

Three factors need special consideration in planning EVO projects. First, managing an evolutionary development project requires substantially more effort than managing a traditional waterfall development project. The contents of each delivery should be planned so that no developer goes more than two releases without input to a release. The goal is to get everyone on the project team developing incrementally. Although it is difficult and time-consuming, the work breakdown structure and dependency information must be done and done correctly.

In addition to more management effort, EVO also requires a fundamental shift in how we think about software development. Traditionally, the first third of a project is spent getting the infrastructure in place before developing any customer-visible capability. This is a problem for an EVO project because EVO requires earlier development of customer-visible functionality to elicit customer response. Delaying customer interaction with a product until the second third of the project is incompatible with this objective.

The solution typically lies in finding some existing code to leverage. Although there is almost always resistance to using this approach, it is usually possible to find something to leverage. If this is not possible, then think about implementing the infrastructure in an evolutionary manner. In any case, the first delivery should be released in no more than two EVO cycles (see Fig. 1) from the start of implementation. The first reason for this is that many of the concerns developers have with evolutionary development are best addressed by doing EVO. Second, if the start of EVO deliveries is delayed too long, the risk that delivery will never happen (until manufacturing release) is increased.

The final planning recommendation is to create a standard development plan that can be used for each cycle. Having the same activities occur at the same time within each cycle helps team members get organized and makes process improvement easier. The activities of three groups should be planned: developers, users, and the group who will make decisions about user feedback. Keeping the cycles short helps keep the developers motivated to make changes in response to customer feedback.

Fill Key Organizational Roles

The unique needs of EVO projects require two additional key project roles to be filled: technical manager and user liaison. Because of the extra management burden imposed by EVO, it is useful to have one of the engineers act as a technical manager. This person is responsible for developing the EVO plan (with the project manager) and keeping track of the progress of the group. The technical manager is key to moving the project forward, resolving daily tactical issues, and handling most coordination activities, with the management team as backup. Typically, technical managers also have development responsibilities, so they tend to understand the dependencies between tasks and can relate more freely to the other engineers.

The other key position, user liaison, trains users, collects their feedback, and communicates changes in the status of the project. The user liaison is the single point of contact for all the users and developers. This person should be the first one to use each delivery to see what has changed and if there are any problems in the code. The liaison's job is to make it easy for users to be involved in the project. A strong user advocate in this role can also contribute to management decisions. Without this role, communication between the developers and users can be haphazard, inefficient, and a major energy drain.

Manage the Developers

Although evolutionary development may seem intuitively obvious, implementing it in a traditional software life cycle environment should not be undertaken lightly. Much of the challenge has to do with managing people. The following steps have also contributed to success at HP:

- Establish a credible business reason for using EVO
- Discuss the method and the rationale for using EVO with the development team
- Ask for feedback
- Develop an initial plan that addresses as many concerns as possible
- Ask the development team to try EVO for a couple of releases and then evaluate future use.

Two major concerns arise in managing developers. The first is a concern that the development effort will degenerate into hacking. To prevent this, the software architecture must be well-partitioned and loosely enough coupled to enable easy modification. This is why object-oriented programming techniques are particularly well-suited to evolutionary development. In addition, one or more persons must be assigned to maintain architectural integrity, and if substantial redesign is required, time must be scheduled. This should be a major consideration in determining if a project is appropriate for EVO methods.

A second concern is that it will be too difficult to make so many releases. If it is difficult to make one release every 9 to 18 months, how much more difficult will it be to release every two weeks? The answer is that when you make frequent releases, you get better at it (if this is not the case, EVO becomes too inefficient). Further, the small chunks in each cycle keep things to a manageable size.

Be aware of a few potential problems that could make managing developers difficult in the implementation phase if not addressed properly. First, users tend to focus on what they don't like, not what they do like. To keep this from being discouraging for the developers, it might help to provide a standard feedback form that elicits "Things I liked" followed by "Things I didn't like." Because many more project management decisions need to be made in EVO, handling decisions can also become a problem. If the decisions are not timely or cause dissension, progress can be delayed. Participatory decision-making techniques have been one solution at HP. Finally, developer overwork or burnout is a potential hazard. Most developers overestimate the amount of software they can write in one or two weeks. While working long hours may seem attractive for a short period, it will ultimately be destructive.

Select and Manage Users

Evolutionary development requires users to exercise each delivery. Many potential users have been alienated in the past by the inability of developers to respond to their feedback in a timely manner. Additionally, users are usually being asked to do a task above and beyond their regular jobs. Consequently, the selection, care, and treatment of the user base is a key issue for an EVO project manager.

The source of the user base is the first issue to address. External customers (through field organizations), internal customers, marketing or field people, and temporary workers have all been used successfully to test products. The closer the project team gets to external customers, the more accurate the feedback tends to be, but the more difficult the customer-relations situation becomes. Several projects satisfactorily used internal surrogate users for early releases and then shifted to external customers.

The user group should have a mix of customers that are representative of the target market. The group must be big enough so that one person doesn't skew the results, yet not so big that managing users overwhelms the project team. Among the user expectations that need to be set are:

- Time commitments to use the product and give feedback
- The possibility of critical problems with the software
- The possibility that the software may or may not change substantially during the project
- Prohibition against discussing the software with anyone outside the project.

If the user is an external customer, the field organization must also be comfortable with their involvement.

In addition to setting expectations correctly, keeping users satisfied during the development process is the other main challenge of managing users. An obvious way to keep users happy is to give them code that works reasonably well. If the code keeps failing, they will get frustrated and tend to stop using it. A second key to customer satisfaction is to take their comments seriously and let them know what changes resulted from their feedback. If a suggestion can't be implemented, explain why to them or, better yet, have one or more of the users involved in the decision-making process. Finally, streamline the software distribution and feedback collection process. Find out what mechanisms customers like to use for installing the software and providing feedback. Then accommodate those desires as much as possible.

Shift Management Focus

Traditional software project management focuses 95% of the team effort on shipping code. With EVO, it is important to focus attention equally on all three components of the process, as shown in Table I.

Table I
Management Focus during Traditional and EVO Life Cycles

Activities	Traditional	EVO
Shipping Code	95%	33%
Getting Feedback	2.5%	33%
Making Decisions	2.5%	33%

Because of the need to radically shift the focus of all involved, getting feedback and making decisions in the early part of the project should be emphasized. Putting a lot of structure around those two activities by doing such things as scheduling regular meetings to review feedback and make decisions will help ensure that they get done. These two activities are prerequisite to getting real value from EVO.

Manage Builds

To do evolutionary development, a project team must have the ability to construct the product frequently. If the product will be released every two weeks, developers should be able to do a minimum of one build per week, and preferably a build every other night. The engineers must be able to integrate their work and test it, or they can't release it. Code that is checked into the configuration management system must be clean, and the build process itself must run in 48 hours or less. Identifying a build engineer or integrator can help the process.

Focus on Key Objectives

While there are many reasons to use evolutionary development on a project, focusing on one or two critical benefits will help optimize efforts. These goals will guide later decisions such as how to structure user involvement, how to change plans in response to user feedback, and how to organize the project. Regardless of what goals are focused on, it is critical to communicate the reasons for strategic decisions to both management and the development team.

Evolutionary development is a different way of thinking about managing software projects. Most groups will probably experience some of the pain that usually accompanies change, so start with a small pilot project first and then try a larger project.

Conclusion

The evolutionary development methodology has become a significant asset for Hewlett-Packard software developers. Its most salient, consistent benefits have been the ability to get early, accurate, well-formed feedback from users and the ability to respond to that feedback. Additional advantages have come from the ability to:

- Better fit the product to user needs and market requirements
- Manage project risk with definition of early cycle content
- Uncover key issues early and focus attention appropriately
- Increase the opportunity to hit market windows
- Accelerate sales cycles with early customer exposure
- Increase management visibility of project progress
- Increase product team productivity and motivation.

The EVO method consists of a few essential steps: early and frequent iteration, breaking work into small release chunks, planning short cycle times, and getting ongoing user feedback. Other components can be modified to accommodate the needs of specific projects, products, or environments. Examples where situation judgments are appropriate include selection of users and length of cycles.

Additional activities, like establishing a clear product vision, identifying a technical manager and user liaison, creating a standard development plan, and setting correct user expectations, will help optimize the benefits of using EVO.
The challenges in using EVO successfully are mostly, but not exclusively, human resource issues. These include the shift in thinking about a new project structure paradigm and perceptions that EVO requires more planning, more tasks to track, more decisions to make, more cross-functional acceptance and coordination, and more difficulty coordinating software and firmware development with hardware.

As noted earlier, many of these perceptions are valid but have extremely advantageous cost-benefit trade-offs. Since many software developers are no longer primary users of their products, they now need to be able to understand the primary users' needs, skill levels, and motivations. Finally, major changes in the customer-developer relationship can result in customer demand for more input and involvement in product definition and design.

HP is continuously improving the EVO process, building on our experience at different divisions. The software initiative team now offers a workshop and consulting expertise on the EVO method. Experience with the value of using EVO to develop the infrastructure and the need for management focus have framed recent implementation efforts.

The key lessons to remember when first attempting EVO are to start small, keep good records, and be diligent about doing the essentials.

Acknowledgments

The authors would like to thank Mike Teska at Manufacturing Test Division for his support of software development method improvement, the board consultant development team consisting of Sharon LaTourrette, John McDermid, Bob Illick, Kathy Withers-Miklos, and Charles Zeng, and the users of the board consultant EVO releases, especially Kent Dinkel. We'd also like to thank Todd Cotton and Beatrice Lam for discussing with us their EVO experiences on some HP projects, and Tony Dicolen and other members of his team at the Microwave Instrument Division for sharing their EVO experiences with us. Finally, we'd like to thank the other development teams within HP that have shared their EVO experiences with us and the software initiative management for sponsoring this work.

Reference

1. T. Gilb, *Principles of Software Engineering Management,* Addison Wesley Publishing Company, 1988.

The Software Initiative Program

HP's Corporate Engineering software initiative (SWI) is a major corporate effort to make software development a core competence at HP. Drawing on the expertise of its members in software engineering, management, and business methods, the SWI partners with product development organizations, delivering knowledge and expertise in key software competence areas to get more products to market faster, with lower costs and higher quality.

SWI's mission is to foster sustainable breakthroughs in product generation capabilities for HP's business goals as related to software and solutions. To achieve this, SWI consultants' work is driven by customer-directed outcomes. In practice, determining what these outcomes are is often in itself a significant contribution to the customer organization and is the first step in any SWI engagement. The SWI team works with multiple levels of group and division management to understand the entity business goals and explore ways of gaining competitive advantage. These goals, along with challenges or obstacles that need to be considered, drive the creation of specific improvement plans.

SWI's value is in its ability to accelerate software development and reduce the risk of having to make fundamental changes to existing software development and management practices. SWI is currently partnering with product development organizations in all of HP's business sectors. This includes significant efforts focused on software reuse, platform development, and testing.

HP Domain Analysis: Producing Useful Models for Reusable Software

Early software reuse efforts focused on libraries of general-purpose routines or functions. These fine-grained assets did not produce the hoped-for quality and productivity improvements. Recent software reuse efforts have shown that architecture-based, domain-specific reuse can yield greater quality and productivity improvements.

by Patricia Collins Cornwell

A software domain is a set of systems or applications that share some common functionality. This common functionality is typically embodied in various software components.* Domain analysis is a software engineering process that produces a characterization of a software domain to support the reuse of the software components. The HP domain analysis method produces a set of models that guide the design of reusable software.

While a few papers and books have been published on aspects of domain analysis,[1,2,3,4,5] very little has been published on practical domain analysis methods. HP has developed and refined a practical domain analysis method which has been used in several reuse projects. The method has proven to be an effective and efficient way to get the information needed for the design of domain-specific software. The focus of each domain analysis is guided by the business priorities and anticipated uses of the domain models.

This article describes a reuse process framework and the essential activities, deliverables, and typical uses of the results from the HP domain analysis method. Because the HP domain analysis method is designed to be tailored to the strategies of HP's business organizations, sections of this article will describe the business contexts for reuse strategies and special reuse roles in the organization.

Reuse Process Framework

Early reuse efforts focused on libraries of general-purpose routines or functions. These small-grained assets did not produce the productivity and quality improvements hoped for because so much engineering effort had to go into integrating these assets to produce a useful product. More recent efforts have shown that architecture-based, domain-specific reuse with larger assets can provide significant productivity and quality improvements. For the past five years, practical engineering experience in adapting reuse to meet HP's business needs has confirmed that the biggest return on investment comes from reuse that is based on domain-specific components that work in a flexible, but well-defined architecture. Reuse-oriented engineering addresses how these software assets are produced, supported, and used.

Because each organization has its own variations on processes and often quite distinct choices of specific methods, the U.S. Department of Defense's Software Technology for Adaptable, Reliable Systems (STARS) program sponsored a project to develop a conceptual framework for reuse processes.[3] This conceptual framework helps organizations understand the relationships among their software asset production, support, and utilization processes. HP participated heavily in the definition of the reuse process framework and provided some of the earliest experiences in its application. The usefulness of this reuse process was validated with organizations adopting reuse-oriented software engineering practices. This article's discussion of major reuse processes blends the knowledge gained from the STARS Conceptual Framework for Reuse Processes (CFRP)[6] with subsequent experience in reuse adoption at HP.

Fig. 1 shows the fundamental relationships among the reuse engineering processes: produce assets, support assets, and utilize assets. All three processes are guided by the results of one or more domain analyses performed in the Analyze Domain process, which analyzes and models a domain for architecture-based, domain-specific reuse efforts. Essential assets of software reuse-oriented engineering include a domain architecture and reusable software components.** Domain analysis produces domain models and other domain information used by the other three reuse-oriented engineering processes. The domain models and information are valuable assets, capturing the organization's knowledge about its product line capabilities and how those capabilities can work together in a range of competitive products.

* A complete software component includes both object code and all related information needed to use it. This related information includes parameterization information, source code if not proprietary, test information, design information, evaluation results, and other descriptive information.

** Some reusable software assets include software generators rather than components that are based on reusable code.

Fig. 1. *The domain analysis software engineering process.*

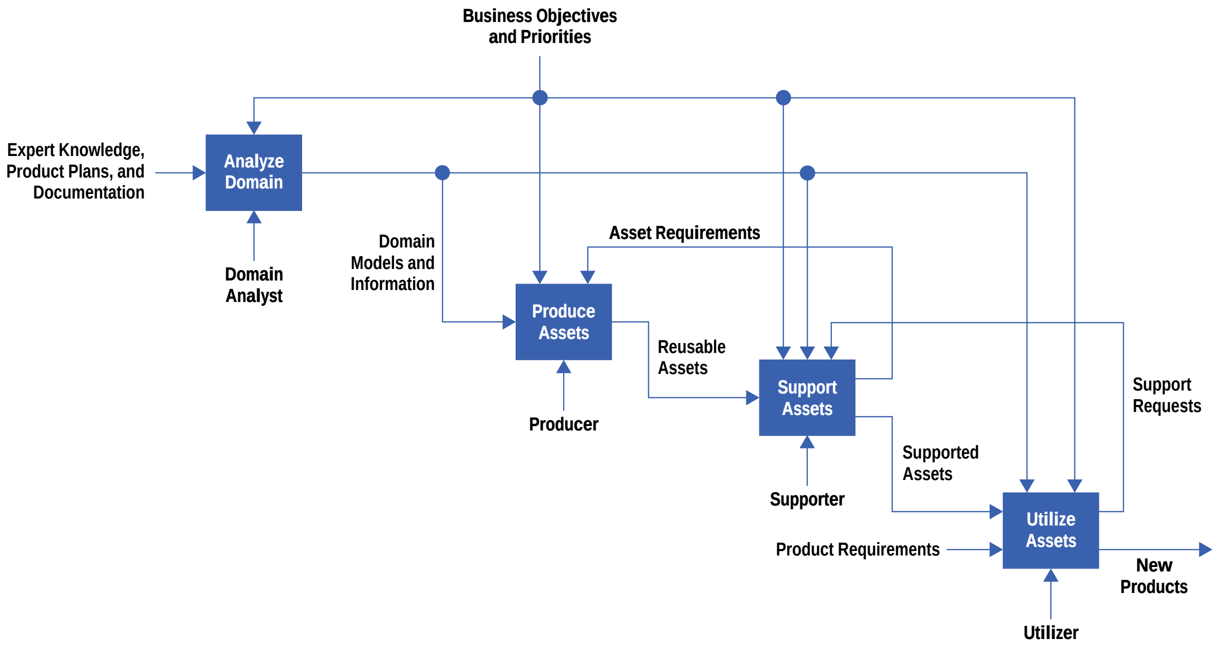
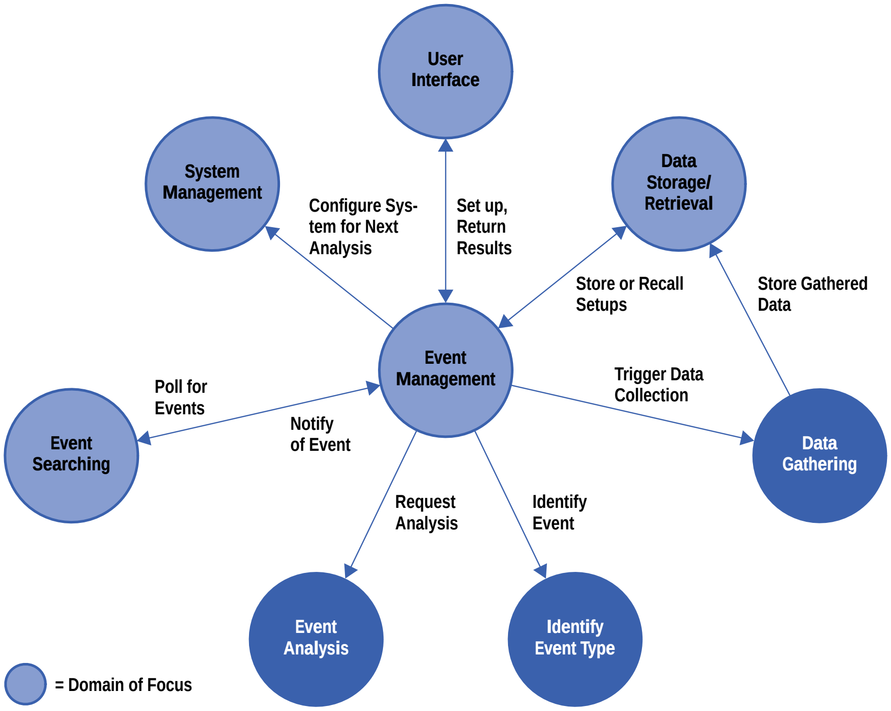


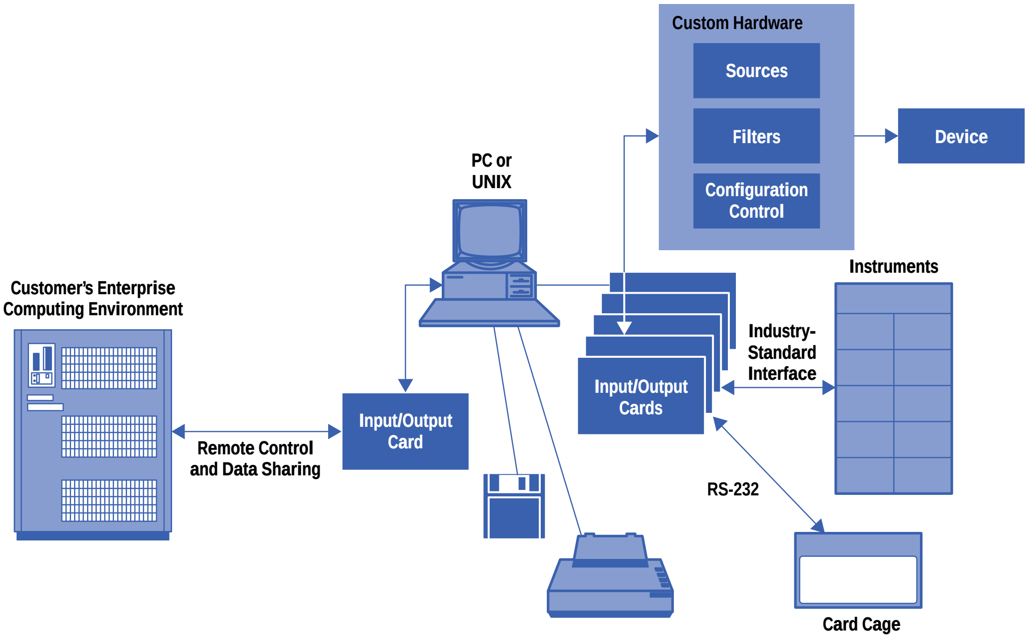
Fig. 2 shows a conceptual model for a data analysis domain, and Fig. 3 shows a physical environment model for a device measurement domain.

Fig. 2. *Conceptual model for a data analysis domain.*



Domain analysis is an essential part of any reuse effort. However, the domain analysis methods being used in industry range from an informal and quick expert prediction of what the domain should cover to a highly structured, exhaustive analysis and modeling effort with hundreds of pages of documentation. For different business situations, either of these extremes or more moderate alternatives may be appropriate. Across software engineering businesses, it is not possible to define one domain analysis method that meets all needs. In fact, even the higher-level descriptions of an overall reuse process may differ significantly from organization to organization. Rather than defining a domain analysis method that would work in only one business context, the HP domain analysis method is adaptable to a wide range of businesses that generate products that include software or firmware.

Fig. 3. Physical environment model for a device measurement domain.



The Produce Assets process shown in Fig. 1 uses domain models to develop reusable assets for use in portfolios of products within the domain. The Support Assets process includes managing the collection of assets and assisting the users in understanding how to take best advantage of the assets. Asset support also includes serving as a users' advocate with producers, integrating the needs of many user groups and assessing the relative benefits of producing or reengineering particular assets. The Utilize Assets process constructs new products with supported assets.

Business Contexts for Reuse Strategies

The first step in any domain analysis is to understand the business priorities and constraints where reuse is going to be used. There are numerous business circumstances that demand an improvement in the way software is developed and maintained. Reuse-oriented software engineering is often used to address business pressures for reducing product cycle time, increasing product quality, escalating the rate of introduction of new product features, and improving employee job satisfaction. However, software reuse is not always an appropriate way to accomplish business goals. Like so many software engineering methods, software reuse has become the goal for some organizations, rather than a means to accomplishing an organization's business goals. To ensure that reuse serves the business needs, we recommend that the management of a reuse effort begin by explicitly identifying the business priorities and analyzing what kind of reuse strategy (if any) will best support achieving those priorities.

Some organizations have launched reuse initiatives only to find that their products do not lend themselves to productive, cost-effective use of the software assets they develop. The two aspects of business context that most influence the decision to employ a reuse approach are the business goals and the product portfolio characteristics.

Businesses need to be more productive than ever to be competitive. Software reuse is rarely a short-term solution for meeting these increased productivity pressures. However, with a managed investment in adopting reuse, the benefits can be measured within the first few uses of the software assets.

Product Cycle Time. In many commercial businesses, the strongest competitors are those who dramatically reduce the time between introduction of a product and the introduction of its successor. To determine the potential impact of software reuse in meeting shortened time-to-market goals, we first assess whether software engineering (development and quality assurance) has critical path activities for the overall product development and release process. Managers who are being encouraged to reduce product cycle time need to focus on engineering activities that produce more and do it faster, possibly with a smaller team. They cannot afford to make organizational or process changes to noncritical path efforts if those changes won't substantially improve the product cycle time. Adopting software reuse involves an up-front investment in new skills and, often, in additional engineering effort. Therefore, investing in software reuse is appropriate when there is a predictable long-term benefit from the up-front investment, and the short-term costs of the investment are tolerable.

Product Quality. To determine the potential impact of software reuse in meeting product quality goals, we first assess how much of a product's quality is determined by the software or firmware. Software reuse is valuable in improving product quality when a significant amount of the functionality is consistent from product to product, so that as that software is tested and in use, more defects are found and fixed. For HP products, the question is often not so much of ensuring the highest quality (which is a must) as it is of providing that quality with less testing time. Nevertheless, as the reusable software “ages” and fewer defects are found in successive uses, customers tend to experience higher quality in the products.

Rate of Innovation. Many businesses make ongoing trade-offs between the rate of introduction of new products and the number of innovations that are new to each successive product. The rate of innovation in products can be increased when the product is based on a stable software platform. A reusable software or firmware platform provides that base functionality without the effort to design and implement the same functionality for each product, freeing the product team to focus on innovations for each new product.

Employee Job Satisfaction. Improved employee job satisfaction has become a very important business goal in some organizations, especially those where intense pressures for meeting deadlines have resulted in employee burnout. We have worked with organizations where the move to reuse was motivated as much by the desire to provide a better work environment for sustainable employee job satisfaction and work-life balance as it was to meet marketplace pressures. For an organization that already has the market share it needs and has a competitive product line, the goal may be to release those future products with a team that is energized by the software engineering effort rather than exhausted by it.

Note that software reuse often makes it imperative for an organization to adopt architecture, design, implementation, and quality practices that would be a significant benefit to their software engineering projects even if reuse were not accomplished. Many software engineers welcome the move to software engineering methods that make them more productive, which contributes to their job satisfaction.

Product Portfolio Characteristics. An increasing number of HP divisions have business plans for a portfolio of products. The products may be tailored to address different market segments, from personal uses to enterprise uses, or tailored to meet specific industry needs, like automotive or banking businesses. We can look at this portfolio as a snapshot-in-time of the set of products a business wants to deliver. In addition, the product portfolio must be managed over time, with the introduction of additional features in successive versions of products. The business’s vintage chart anticipates the desired product evolution and provides essential information for assessing the potential for reuse.

The characteristics of a product portfolio that can improve the prospects for software reuse rely mostly on the stability of the feature set in the product portfolio. There must be some significant set of base functionality that the set of products have in common to make it profitable to invest in reuse. This common functionality may constitute as little as 10% of a product’s software and still be worth implementing with reuse in mind.

To reduce the risk of adopting reuse (any change involves risk, as does no change), those chartered with producing the reusable software rely on access to experts in the kinds of functionality for the products that will be produced. These people are often referred to as domain experts. These experts must be made available to the domain analysis effort as part of management support for the reuse strategy.

We use a rule of thumb in which the asset designers must get access to at least three existing examples of products that have the kind of functionality they want to provide in a reusable form. They also need characterizations of at least three intended future products that would also have that kind of functionality. The three examples give concrete information about what functionality is common (an existence proof). The three projected uses suggest that the investment will be amortized adequately to realize the benefits of designing, developing, and maintaining the software assets. The future uses also suggest the range of variation the assets must support.

HP Domain Analysis

The HP domain analysis method was developed by HP’s software initiative (see ***Sidebar*** in Article 4). The HP domain analysis method supports analysis and modeling of capabilities (functionality or services) provided by domains such as microwave frequency measurement modules, crosscorrelation analysis algorithms, or report generation routines. The method explicitly addresses gathering the constraints and requirements of producers, supporters, and anticipated users of the domain analysis and related assets. A reuse strategy for software engineering is most successful when producers, supporters, and utilizers are full, active contributors to the domain analyses they will later use. The roles of the producers, supporters, and utilizers are described in the sidebar “***Reuse Roles***”.

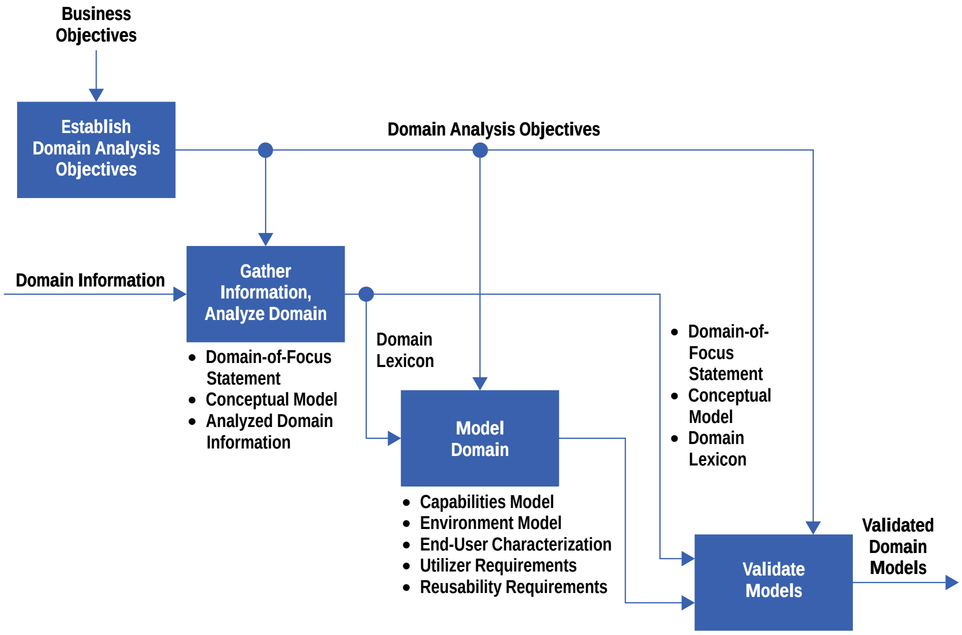
The Analyze Domain process shown in Fig. 1 produces a kind of reusable asset in the HP domain analysis method and is, therefore, conceptually a kind of Produce Assets process. Nevertheless, Fig. 1 shows the Analyze Domain process separately to emphasize that HP domain analysis guides and is guided by all three fundamental reuse engineering processes.

Basics of HP Domain Analysis

The HP domain analysis method includes domain analysis and modeling. The analysis identifies capabilities of systems in a domain of focus (i.e., the set of systems being analyzed), and classifies the common capabilities and the range of variation across systems that are anticipated in the future. The modeling captures the relationships among critical capabilities in the domain and creates models of the capabilities and their relationships without imposing a particular implementation solution.

In HP domain analysis, the term “domain” refers to any set of implementations (systems or subsystems, for example) in which the implementations have some common capabilities. Most of the time we define the domain by the set of common

Fig. 4. *The activities and deliverables that make up the domain analysis process.*


Business Objectives
Establish Domain Analysis Objectives
Domain Analysis Objectives
Domain Information
Gather Information, Analyze Domain
• Domain-of-Focus Statement
• Conceptual Model
• Analyzed Domain Information
Domain Lexicon
Model Domain
• Capabilities Model
• Environment Model
• End-User Characterization
• Utilizer Requirements
• Reusability Requirements
• Domain-of-Focus Statement
• Conceptual Model
• Domain Lexicon
Validate Models
Validated Domain Models

capabilities, rather than by listing all the potential implementations that could fall in the domain. For example, for a domain like a microwave measurement test system, there are endless possible products. However, most implementations of microwave measurement test systems include capabilities like test management, data management, report generation, signal measurement, and so on.

There are no common rules about what makes a set of capabilities the right size and complexity to be called a domain. If the domain of focus represents a consistent set of capabilities in a larger context (for example, in the context of a product portfolio), the most useful scope for such a domain is one in which there is a high degree of cohesiveness among the capabilities within the domain, and a limited coupling to other domains with which the domain of focus might be combined to produce products. For example, a domain like a graphics editor has significant complexity within it. However, in a larger context like document publishing, the graphics editor might have connections to other domains like text editing and document printing, which have a well-defined and comparatively simple interface.

Intuitively, domain assets are much more likely to be reusable if they provide a coherent “chunk” of desired capabilities and if the assets are easy to integrate into a complete solution. Typically, ease of integration is accomplished with a simple interface between the assets and the rest of the product software or firmware.

The Method

The HP domain analysis method usually involves three cycles through a set of well-defined activities. Fig. 4 shows these activities and the deliverables they produce, and Fig. 5 shows a typical level of effort expended on each deliverable during each of the three cycles through the domain analysis process. At each step through these cycles the analysis and models are refined and deepened and go through the same basic activities. The first cycle usually needs to focus on the context in which the domain will function because the team is still analyzing just what part of the overall product portfolio the domain must cover. The second and third cycles refine the scope of the domain and fill in details on domain capabilities and their relationships.

The following sections describe the activities shown in Fig. 4 and the deliverables produced by each activity.

Establish Domain Analysis Objectives. Use business goals and constraints to produce a clear statement of the purpose for the domain analysis. Identify those who have a stake in the domain analysis. Typically stakeholders include managers in the product development organization, those who will design and implement the domain assets, those who will support those assets, and those who are targeted to utilize the assets. Get agreement from these people on the objectives.

The deliverables from this activity include a statement of desired objectives for this domain analysis and a set of metrics that will determine the progress and success of the analysis. The documentation will also show how the domain analysis objectives are aligned with the business goals. Fig. 6 shows an example of some of the objectives and metrics for a domain analysis project.

The value of this activity is that it ensures that the domain analysis meets business needs, enables the domain analysis team to manage its investment of time and effort, and establishes a partnership with stakeholders to ensure that the domain analysis results meet their needs.

Fig. 5. The level of development effort expended during each cycle of the domain analysis process.

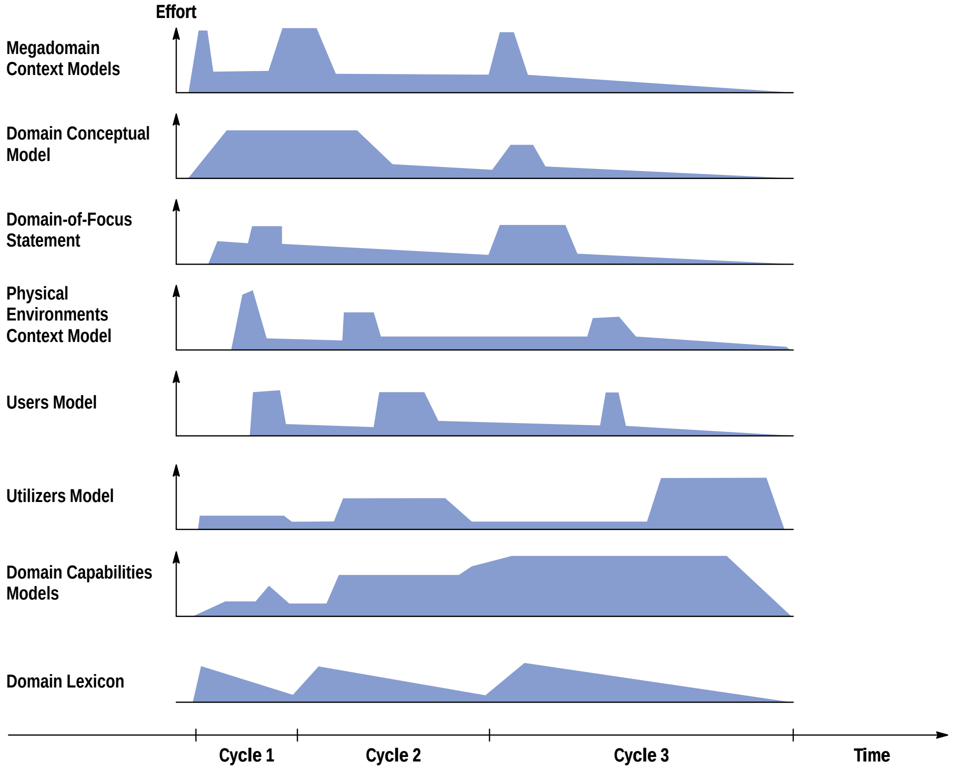


Fig. 6. An example of some of the objectives and metrics for a domain analysis project.

Objective: The domain analysis will assess the value of building products based on reusable software, where that value is based on potential competitive advantage.

Metrics:
- Count the number of competitive capabilities in the domain capability models.
- Name the clusters of capabilities that marketing identifies as a competitive advantage.

Objective: Determine what software would be highly reusable as a software platform in our products targeted for the years 1997 to 2001.

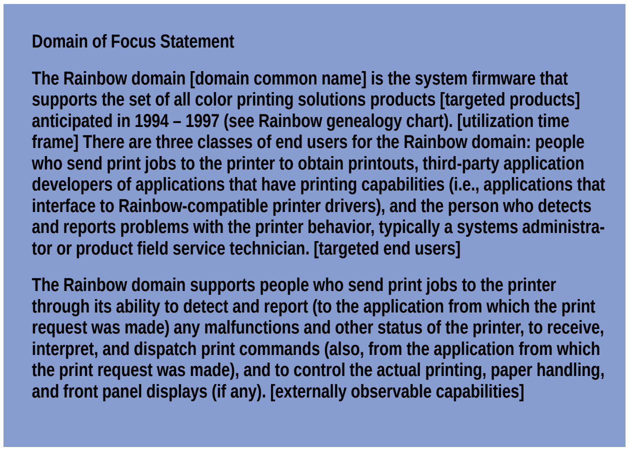
Metrics:
- Does the domain context model clearly identify which parts of the domain constitute the platform domain?
- Does the platform domain's conceptual model capture the capabilities needed for targeted products in the years 1997 to 2001? Are those capabilities highly similar across the targeted products?

Develop a Domain Context Model. Identify examples of existing systems that include capabilities of the domain you have identified. Use the list of targeted uses of the domain from the domain-of-focus statement to identify examples of future systems that include the domain. Develop a top-level model of the major elements of the systems. This is called the *megadomain.* Normally this model contains five to eight elements, with labeled relationships shown on the interconnections between the elements. Fig. 2 shows a conceptual model of a megadomain. This model serves as the organization's domain context model, since the parts of the megadomain that are in the domain-of-focus statement can be highlighted and the relationship to the overall megadomain can be identified.

This activity identifies a top-level interface between the domain and the rest of a system. The focus of the activity is in identifying what parts of a system are within the domain. The domain boundary decision is refined over the three cycles of the domain analysis.

Identify the Domain of Focus. Determine what domain will be the focus of the domain analysis and produce a *domain-of-focus statement*. The domain-of-focus statement is a descriptive statement (about three to five pages long) of what characterizes the domain, in terms of its anticipated uses and the scope of its capabilities. This statement complements the conceptual model (described below) and provides a basis for shared understanding about the domain, without needing to understand the detailed relationships of elements in the domain. Because it describes the scope of the domain, it guides the more detailed modeling efforts. Fig. 7 shows an excerpt from a domain-of-focus statement for a printer command handling domain.

Fig. 7. *A portion of a domain-of-focus statement for a printer command handling domain.*



Domain of Focus Statement

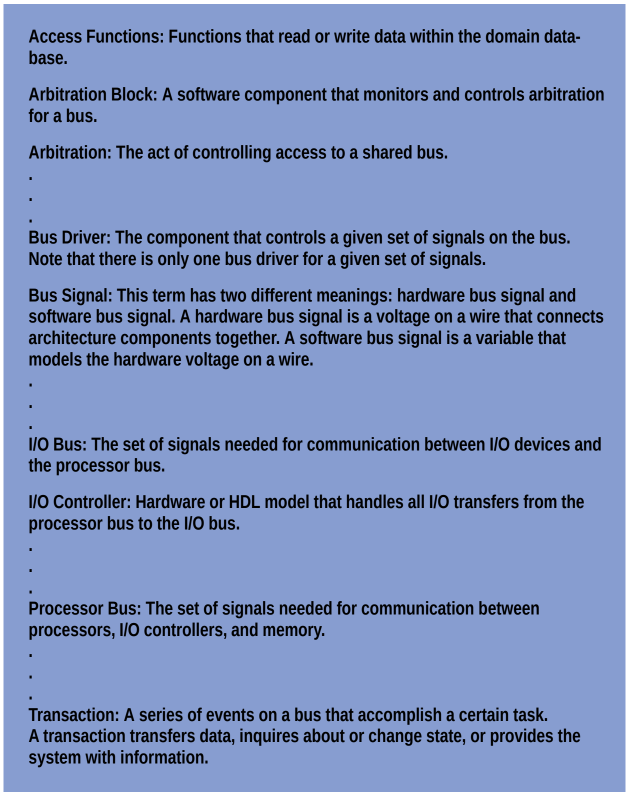
The Rainbow domain [domain common name] is the system firmware that supports the set of all color printing solutions products [targeted products] anticipated in 1994 – 1997 (see Rainbow genealogy chart). [utilization time frame] There are three classes of end users for the Rainbow domain: people who send print jobs to the printer to obtain printouts, third-party application developers of applications that have printing capabilities (i.e., applications that interface to Rainbow-compatible printer drivers), and the person who detects and reports problems with the printer behavior, typically a systems administrator or product field service technician. [targeted end users]

The Rainbow domain supports people who send print jobs to the printer through its ability to detect and report (to the application from which the print request was made) any malfunctions and other status of the printer, to receive, interpret, and dispatch print commands (also, from the application from which the print request was made), and to control the actual printing, paper handling, and front panel displays (if any). [externally observable capabilities]

Be sure the terminology used is documented in the *domain lexicon* (described below). As the description of the domain evolves, be sure to keep the domain-of-focus statement consistent with the domain models, especially the conceptual model. Fig. 8 shows a portion of the lexicon for an I/O bus project.

This activity provides a succinct statement of the essential characteristics of the domain for use with stakeholders. It also serves as a reference for decisions about the scope of the domain.

Fig. 8. *An example of definitions that might appear in a domain lexicon.*



Access Functions: Functions that read or write data within the domain database.

Arbitration Block: A software component that monitors and controls arbitration for a bus.

Arbitration: The act of controlling access to a shared bus.
.
.
.
Bus Driver: The component that controls a given set of signals on the bus. Note that there is only one bus driver for a given set of signals.

Bus Signal: This term has two different meanings: hardware bus signal and software bus signal. A hardware bus signal is a voltage on a wire that connects architecture components together. A software bus signal is a variable that models the hardware voltage on a wire.
.
.
.
I/O Bus: The set of signals needed for communication between I/O devices and the processor bus.

I/O Controller: Hardware or HDL model that handles all I/O transfers from the processor bus to the I/O bus.
.
.
.
Processor Bus: The set of signals needed for communication between processors, I/O controllers, and memory.
.
.
.
Transaction: A series of events on a bus that accomplish a certain task.
A transaction transfers data, inquires about or change state, or provides the system with information.

Gather Domain Information. Look at existing systems that include the domain. Also look at information on future products that might be able to use reusable designs and implementations in the same domain. Interview domain experts for their knowledge about trends in technology or capabilities associated with the domain. Identify the externally observable capabilities of the domain.

This activity provides the technical information for analyzing and modeling the domain, and ensures that the domain characterization is accurate and adequately complete.

Develop a Conceptual Model. Create a graphical depiction of the domain's primary elements and their relationships. This model may not be a technically accurate, top-level representation of the eventual design, but rather is an easy way to understand the "big picture" of the domain. It complements the domain-of-focus statement. Often the domain-of-focus statement and the conceptual model are used in management briefings and as part of the support documentation for engineers who need a general understanding of the domain.

There is no need to be fancy with the conceptual model. In fact, an intuitive model is preferred. Arcs between elements of the model must be labeled and the intended interpretation of the connections between those elements must be documented. The conceptual model should use the terminology defined in the lexicon and must be consistent with the domain description in the domain-of-focus statement. Usability of the conceptual model is enhanced by using a single modeling paradigm such as process flow, data flow, or entity-relationship diagrams.

Define the Domain Lexicon. This is an ongoing activity in which terms used to discuss the domain are defined, with pointers to related terms. The lexicon is used as a reference document for the terminology of choice for communicating about the domain. The domain lexicon includes every term in the domain models. It is invaluable in saving time and minimizing misunderstandings among the asset producers or between asset producers, supporters, and utilizers because it enables them to interpret the domain models consistently. The domain lexicon also allows a new member of a team to study the domain models and gain an initial understanding of the models without needing full-time assistance.

This effort will create a common understanding of concepts related to the domain, capture decisions about preferred terminology, and support efficient learning about the domain.

Model Domain Capabilities. Develop a model or set of models that capture the relationships among externally observable capabilities of the domain. This model is based on the gathered domain information, uses the terminology of the lexicon, and is consistent with the domain-of-focus statement and the conceptual model. The capabilities model is the primary reference used by the domain architect and asset designers. Thus, it must contain a characterization of all essential, externally observable capabilities of the domain. Also, because the capabilities model is concerned with the externally observable capabilities of the domain, it can be a valuable document for those needing to understand how to maintain or use the domain.

Later, the architecture and designers (who are members of the producer team) will transform the capabilities model into a chosen engineering solution. For organizations that have not done formal architecture and design, the capabilities model may serve as the design, supplemented by the other domain models. As the organization's skill in software design increases, the capabilities model will be the primary source of information for guiding the more detailed and formal software design. The domain analyst documents links between gathered domain information and decisions about capabilities and their relationships.

The most practical approach to capabilities modeling is to use the same paradigm as will be used in the architecture and design of the domain assets. Feature-based models, entity-relationship-attribute models, object-services models, and logic-rules models can be employed. However, the domain models are not meant to reflect implementation decisions. Furthermore, the capabilities models show only what is observable to utilizers and end users of the domain capabilities.

Most capabilities modeling requires a combination of aggregation, abstraction, and decomposition approaches to identify the top two or three layers of externally observable capabilities. Since these layers may not be strictly hierarchical, the models must capture the kinds of relationships that exist among the capabilities. Capability models identify each of the capabilities as required, or optionally, identify sets of alternative capabilities and note other capability interdependencies.

The most practical way to capture the range of variation across intended uses of the domain is through use scenarios. One very useful kind of model shows stimulus-response relationships among capabilities (i.e., what services or actions transpire as a result of what events) for different scenarios.

Modeling the domain capabilities provides domain architects and asset designers with a characterization of the domain's externally observable capabilities in sufficient detail for architecture and external design needs.

Model Physical Environments. Characterize the physical environments in which the software for the domain needs to work. This may mean describing the processors in an instrument where firmware runs, or the various heterogeneous enterprise environments where application software will run. Also, if there are diverse interface standards to be met, those are captured in this model. Fig. 3 shows an example of a simplified physical environment context model.

This activity ensures that asset designers have the information needed to accommodate computing environment constraints, like parallel or distributed processing, and refines the scope of the intended use of the domain assets.

Model End Users' Needs. Capture the characteristics of the end users that could influence the design or the implementation of domain assets. Each targeted development project may provide a characterization of their end users that includes skill level, understanding of how the product works, expectations, and a mental model of the user interface. The models of end users translate the end-users' usability requirements into a model of the capabilities the domain provides to meet those requirements.

This activity ensures end-user usability of domain assets and often influences the set of capabilities provided.

Model Utilizer Needs. Use identified utilizer needs for usability of the domain assets. This list usually includes the utilizer's constraints with respect to development environment (programming tools, version control and configuration management expectations, etc.), usage support requirements, and skill level. This information may influence the reusability requirement decisions and the domain architecture or asset design. The utilizer model translates the utilizer's usability requirements into a model of the capabilities the domain's components provide to meet those requirements. This model is typically quite different from the end-user model because it shows what will need to be provided in the product development phase, rather than the functionality needed in the delivered product.

Reusability Requirements. Develop a clear statement of how the domain will interpret such reusability characteristics as portability, modularity, scalability, extensibility, tailorability, interoperability, plug compatibility, and standards conformance. This statement defines how the team will know when they have achieved adequate reusability in their domain design and implementation.

Validate Models. Ensure consistency, completeness, and usability of the domain analysis and models. This activity is best supported with regular and explicit quality assurance activities, like minireviews or checkoffs against objectives and measures defined in establishing domain analysis objectives. This activity will ensure quality and provide an assessment of when the domain analysis is adequately complete.

How Much

The knowledge captured during a domain analysis is essential for success in reuse. Therefore, domain analysis is not overhead but rather a low-risk, efficient way of gathering and developing domain knowledge in forms that are readily accessible to those in the organization who:

- Develop reusable software or firmware
- Develop products that use the reusable assets
- Support the assets.

The larger the domain, the more people are typically involved in the domain analysis. Nevertheless, planning for it to take about six weeks of full-time effort from the start of the analysis and modeling until the stakeholders have what they need to delve into architecture and asset design is reasonable. Normally, the stakeholders are the first people to use the domain analysis results. For a team new to domain analysis, productivity will be greatly enhanced by having an experienced domain analyst (even one from a very different domain) available to guide the team through the method.

As mentioned earlier, the HP domain analysis method has three cycles of well-defined activities. For a typical domain, the first cycle generally takes about a week of focused effort for the domain analyst, who is leading the domain analysis effort, plus 10 to 30 hours for each of the domain experts providing information and assisting in the analysis. The second cycle takes about two weeks of information gathering, analysis, and modeling for the domain analyst and those who will be involved in the design and implementation of the domain assets. The users are consulted during this cycle, but the time commitment may only need to be a few hours for review and suggestions. The final cycle takes about three weeks for refining and validating the analysis and models. The domain analyst is involved full time, while the domain architect and domain asset designers may begin sketching out their first asset design ideas in parallel with the third domain analysis cycle. This parallel approach can ensure that the refining and validating activities meet the designers' needs. Domain analysis typically ends as an explicit activity when the design team has the information it needs to develop or reengineer an architecture and reusable software. However, it is important to maintain consistency between requirements, domain models, domain architecture, and asset design.

Using HP Domain Analysis Results

The deliverables from an HP domain analysis are designed to be useful in other, specific reuse activities. In architecting the domain and designing the reusable assets (Produce Assets in Fig. 1), all of the deliverables are used. The capability models and domain characterizations are primarily targeted to be used to guide these activities.

In supporting asset utilization, the lexicon is indispensable. The conceptual model and domain-of-focus statement are especially useful in acquainting developers with the domain assets (a form of asset support). The capability models and domain characterizations provide useful details for the utilizer, who is trying to understand how the assets might best provide the capabilities needed in the product (another form of asset support). The lexicon and capability model also support asset management through library classification, asset access, and configuration management.

In using assets, the utilizer will likely reference most of the domain analysis deliverables, initially relying on the conceptual model and domain-of-focus statement. Part of using assets is taking the initiative to identify asset requirements to the producers, who will translate requirements requests into refinements of the impacted deliverables. For example, the need for a new capability is reflected in the capability models and lexicon.

Generally, managers rely most heavily on the conceptual model and domain-of-focus statement, with the lexicon as background material. See the sidebar: ”***Management***”.

Conclusion

The HP domain analysis method provides a simple and effective way of getting information needed to be successful in domain-specific, architecture-based reuse. By providing a method with a clear set of deliverables that have well-defined uses, we improve the efficiency and effectiveness of the domain analysis. Following the HP domain analysis method can substantially reduce the risks in reuse-oriented software engineering, risks that arise when the assets produced and supported do not adequately meet utilizers’ or end-users’ needs.

In most cases, we find the best results are obtained working with an experienced domain analyst the first time a team goes through the cycles to do their first domain analysis. Over time, that team’s experience in domain analysis can increase to a level of domain analysis expertise that can be spread throughout the organization.

Acknowledgments

The HP domain analysis method was initially captured as a domain analysis workbook. Its primary author was Mark Simos of Organon Motives, who was a consultant to HP’s software initiative program. Modifications to the workbook and the underlying process model were influenced by the work of Ruben Prieto-Diaz and Guillermo Arango.[4,5] A completely new workbook was developed after the modified workbook was used with HP product development teams in the computer peripherals, computer systems, medical, and test and measurement businesses. Mike Ogush and Tom Van Slack from the software initiative program have provided valuable suggestions for making the method match the needs of HP software development organizations.

References

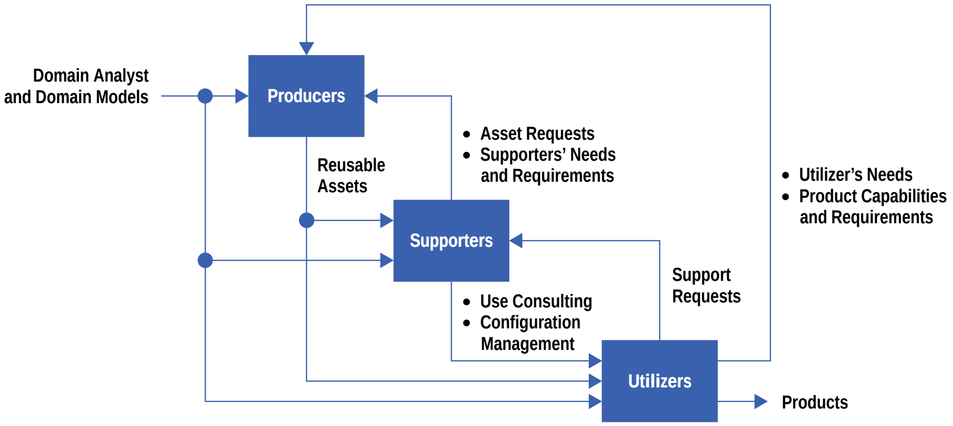
1. J. W. Hooper and R. O. Chester, *Software Reuse: Guidelines and Methods*, Plenum Press, New York,1991, pp. 51-66.
2. *Symposium on Software Reusability*, ACM SIGSOFT, Seattle, Washington, April 28-30, 1995.
3. T. J. Biggerstaff and A. J. Perlis, *Software Reusability*, Vols. 1 and 2, ACM Press, New York, 1989.
4. G, Arango, “Domain Analysis : From Art Form to Engineering Discipline,” *Proceedings of the Fifth International Workshop on Software Specifications and Design*, 1989, pp. 152-159.
5. R. Prieto-Diaz, “Domain Analysis of Reusability,” *IEEE Proceedings of COMPSAC ’87*, 1987, pp. 23-29.
6. *STARS Conceptual Framework for Reuse Processes (CFRP), Volume I: Definition*, STARS-VC-A018/001/00, September 30, 1993, and *STARS Conceptual Framework for Reuse Processes (CFRP), Volume II: Application*, STARS-VC-A018/002/00, September 30, 1993.

Reuse Roles: Producers, Supporters, and Utilizers

In the early stages of moving to reuse-oriented product development, software engineers take on the roles of being responsible for developing their software to be reusable (producers), learning how to use software developed by others (utilizers), and supporting their software for use by others (supporters). As reuse becomes more systematic, it is common for organizations to evolve so that individuals take on the specific roles of producer, supporter, or utilizer for the duration of a product development cycle.

Fig. 1 shows the primary relationships among software engineering roles in a reuse-oriented organization, and the following sections describe the responsibilities for each of the roles.

Fig. 1. *Relationships among the software engineering roles in a reuse-oriented organization.*



Domain Analyst and Domain Models
Producers
Reusable Assets
• Asset Requests
• Supporters' Needs and Requirements
• Utilizer's Needs
• Product Capabilities and Requirements
Supporters
Support Requests
• Use Consulting
• Configuration Management
Utilizers
Products

Domain Analyst

- Analyze the common feature set and the range of feature variation across the projected uses of the assets.
- Characterize capabilities the domain must provide to support the users of products built with domain assets by the product developers. Capture the characterization in models that can be used to design and develop domain assets and guide the use of those assets.
- Produce conceptual models that are readily understandable by managers, new project managers, and engineers who will produce, utilize, or support the domain assets.
- Extract domain information from diverse sources such as past designs, interviews with experts, product data sheets, and trade press articles.
- Use consistent, unambiguous terminology captured in the domain lexicon to communicate about the domain.
- Develop and maintain a working partnership with producers, supporters, utilizers, managers, and key technical contributors.

Producers

- Include utilizers' requirements and needs as part of the design. Consider the utilizers' assessment of product requirements and what it takes for them to be able to tailor and integrate the assets easily to build products.
- Include supporters' requirements and needs as part of the design. Consider the supporters' ability to maintain the assets, to manage the asset base's evolution, and to provide assistance to utilizers.
- Develop an architecture for the product portfolio that clearly defines the common elements and the range of variation across the uses of those elements. Design the architecture's evolution to meet delivery requirements.
- Design the assets to support critical abilities like portability, supportability, extensibility, scalability, and tailorability and to meet functionality and performance requirements.
- Develop and maintain a working partnership with the domain analysts, supporters, and utilizers, including managers and key technical contributors.

Supporters

- Develop and maintain a configuration management process and environment that support the producers and the various teams of utilizers, as well as making it easy to configure and distribute releases.
- Provide asset use consulting to utilizers.

- Join with producers throughout the producers' development effort to ensure that the assets will be easy to understand, maintain, and port.
- Contribute to the prioritization of asset development and support plans and consider the overall business priorities and needs.
- Develop and maintain a working partnership with domain analysts, producers, utilizers, managers, and key technical contributors.

Utilizer

- Use the architecture and available software assets to guide every phase of the product development life cycle. This includes everything from determining product requirements to quality assurance.
- Design the product to take advantage of new combinations of features that could provide a market advantage.
- Join with producers throughout the producer's development effort to influence their design and implementation of assets so that they will meet the utilizer's product needs.
- Join with the domain analyst to influence the scope of the domain and the domain utilizer's model.
- Contribute to the prioritization of asset development and support plans, considering overall business priorities and needs.
- Develop and maintain working partnerships with domain analysts, producers, supporters, managers, and key technical contributors.

Management

Successful reuse depends on a different kind of relationship among engineering teams. The teams behave as true partners, each concerned for and respectful of the legitimate needs and constraints of the other. Because this partnership is so critical to successful reuse, managers whose responsibilities span the producer and utilizer organizations need to understand the issues involved, support the reuse effort with computing and training resources, allow time for regular communication and joint decision-making by producers and utilizers, and acknowledge the value of the reuse investment.

This is no easy task when the next product's release is in danger of slipping and the return on the reuse investment will not be felt until products after the current product are in development. Nevertheless, without informed senior management involvement and support, producer and utilizer organizations find it difficult to behave as true partners, and typically fall back into the old ways of engineering software, which are the very ways they had decided to abandon because they would not meet future business needs.

In one HP organization, a lab manager endorsed a pilot reuse project for reengineering existing software into a more easily reused software platform. The lab manager's responsibilities spanned the platform engineering team and numerous product development teams that would use the platform. The lab manager's leadership and active involvement in understanding the issues, providing resources, and rewarding reusable results transformed the way software and firmware components for an entire product line were produced. The transformation did not take place overnight. However, the business is now producing four to six products per year rather than the one per year that it was producing four years ago when the reuse effort began. This has been accomplished with a modest increase in staffing and a great deal of software and firmware reuse or leverage.

In another business sector, a group of businesses agreed to cooperate on the development of a reusable software platform that could be used in numerous product lines. The senior management (R&D group managers) invested in the project and encouraged the individual businesses to invest senior technical contributors in the asset production effort. These senior technical contributors served as producers in the early months of the reuse effort. However, they were also responsible for representing the needs of their individual organizations as potential utilizers of the software platform. As a serendipitous result, the resulting platform was used to bootstrap a new business that had many of the same product capabilities that had been analyzed and designed into the reusable software, allowing HP to get to market substantially faster with a highly competitive product.

Unfortunately, there are also examples where talented engineers have developed solid technical solutions for reuse, but were unable to engage their targeted utilizers or were unable to get senior management involvement and active support. Eventually, each of these investments was abandoned with a return to the short-term product development methods that were not meeting business needs when the reuse effort began. There is a very strong correlation between engaging producers, utilizers, and managers and succeeding with reuse.

A Model for Platform Development

For many software and firmware products today, creating the entire architecture and design and all the software modules from the ground up is no longer feasible, especially from the point of view of product quality, ease of implementation, and short product development schedules. Therefore, the trend is to create new product versions by intentionally reusing the architecture, design, and code from an established software platform.

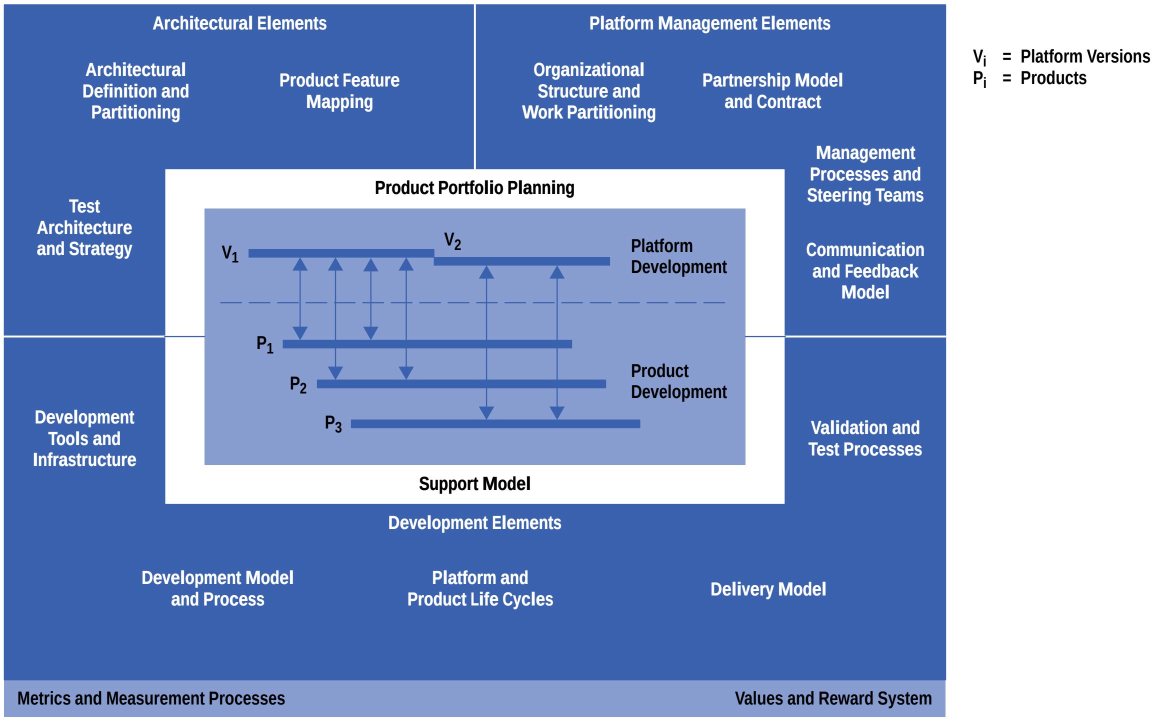
by Emil Jandourek

HP's software initiative program has been working in partnership with product development organizations in Hewlett-Packard for almost five years. Its goal is to help take software and firmware development off the critical path of new product introductions and transform HP's software and firmware development capability into a competitive advantage. Through our work we have observed and participated in the application of many different strategies, all aimed at raising an R&D team's collective ability to build software and firmware that meets the overall market requirements, including functionality, usability, reliability, performance, supportability, and time-to-market goals.

Several patterns have emerged that many HP organizations are successfully using to elevate their software and firmware development capability. One pattern corresponds to a set of operational practices that we call the *platform development paradigm*. The software initiative program has created a conceptual model for platform development (see Fig. 1) which builds upon HP's product development experience and integrates many of HP's best practices in software development. The individual elements of the model are closely tied to the technical and management systems used in the company and have been validated through actual team experiences in developing new products.

Since the platform development model is conceptual, it is used as a framework for determining the elements that an organization needs to invest in to attain a competency in platform development. The software initiative program works with product development organizations to identify the areas of the model that are applicable to a given organization's situation and works with the organization to customize the model accordingly. The resulting instantiation of the model yields processes tuned to the specific needs and requirements of the particular development organization, leading to a new level of development capability. Organizations within HP that have established a competency in platform development have

Fig. 1. *Major elements of the platform development model.*


Architectural Elements
Architectural Definition and Partitioning
Product Feature Mapping
Platform Management Elements
Organizational Structure and Work Partitioning
Partnership Model and Contract
Management Processes and Steering Teams
Communication and Feedback Model
Test Architecture and Strategy
Product Portfolio Planning
V1
V2
Platform Development
P1
P2
P3
Product Development
Support Model
Development Tools and Infrastructure
Validation and Test Processes
Development Elements
Development Model and Process
Platform and Product Life Cycles
Delivery Model
Metrics and Measurement Processes
Values and Reward System
Vi = Platform Versions
Pi = Products

significantly reduced their time to market, improved operational efficiency, and become more responsive to the needs of their customers. These gains are accompanied by improved business results.

The following are brief descriptions of the elements of the platform development model shown in Fig. 1.

- Product Portfolio Planning. This element defines the strategic relationship between the platform and all product versions to be released over a multiyear period. It identifies the key business drivers and sets the overall goals, direction, priorities, and parameters of the platform strategy.
- Architecture. This group of elements includes:
  - Architectural definition and partitioning of the major functional and technology subsystems.
  - Product feature mapping, which identifies appropriate subsystems and component modules used in the implementation of each feature (i.e., translation of customer needs to product features to specific platform or product modules)
  - Test architecture and strategy, which define the overall structure and methods for verification and validation to ensure necessary quality levels in the final product.
- Platform Management. This group of elements includes:
  - Organizational structure and work partitioning, which defines the organization's operating model at an abstract level (e.g., reporting relationships and team organization)
  - Partnership model and contract, which provides the generic framework for instantiating the operating model between platform and product teams (e.g., interdependence between teams and expectations for their working relationships)
  - Management processes and steering teams, which define how the product portfolio plan is created and how its execution is managed
  - Communication and feedback model, which defines the timing and content of the information that flows between teams.
- Development. This group of elements includes:
  - Platform and product life cycles, which define the major phases, with goals, activities, and deliverables for both the platform and products
  - Development model and process, which specify the processes followed for the creation and enhancement of a module through its integration into the final product
  - Delivery model, which defines how platform components and subsystems are delivered for use within products
  - Validation and test processes, which define the specific quality criteria and test procedures used throughout the product and platform life cycles
  - Development tools and infrastructure, which provide a common development environment and processes for platform and product work (e.g., procedures and tools for creating, storing, finding, building, and testing components).
- Support Model. This element defines the mechanics and logistics of how individuals and teams get help when using platform components.
- Metrics and Measurement Processes. This element defines the means by which progress and results for each of the other elements are monitored to ensure achievement of business goals.
- Values and Reward System. This element integrates and aligns the organization's values and culture with its performance evaluation and reward mechanisms to support the other elements of the model and thereby achieve platform, product, and business goals.

The remainder of this article describes the key elements of the model in greater detail, including the deployment and use of the elements, anecdotes about their implementation, and finally, HP's experiences with the model. The use of the word "software" throughout this article refers to both software and firmware.

Definitions and Background

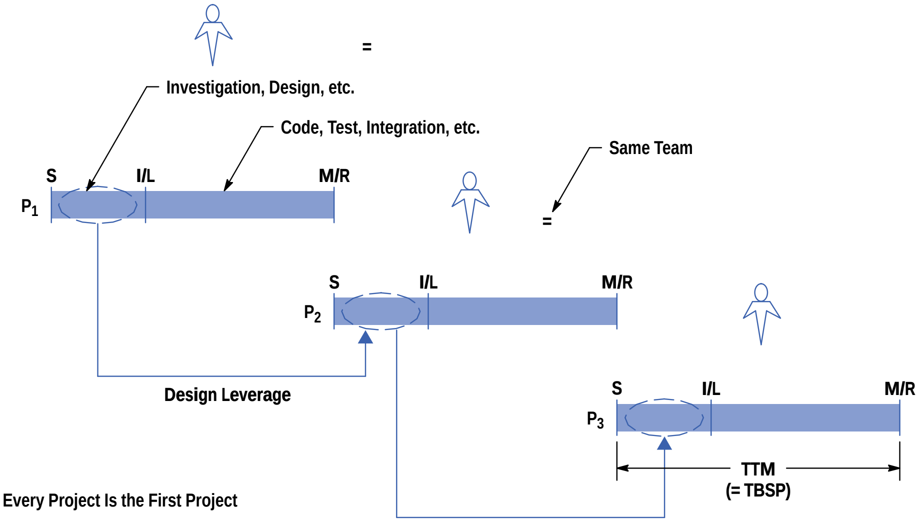
For HP and many other high-technology businesses, the evolution of product development organizations parallels that of a company's business. The character of a business changes as its products evolve, mature, and expand their market penetration beyond the innovators and early adopters.

This technology adoption life cycle has implications for how an organization develops its products.[1] An organization's first product for a new, emerging market is often an experiment aimed at validating a product concept and getting feedback to help shape its evolution. Consequently, the first product is often incomplete and may in fact be a cleaned-up prototype. Successful market introduction and subsequent demand for the product inevitably lead to plans for follow-on products.

Paradigm I: Serial Development Projects

Development during the early stage of a new product's life cycle is characterized by a series of independent projects (see Fig. 2). A "just build it" mentality often drives the first few products because of uncertainty about the market acceptance of the product. From a software development perspective, the product's architecture and design are often implicit and poorly

Fig. 2. Paradigm I, serial development projects.



Every Project Is the First Project

Characteristics:

- Serial Development
- Project Independence
- "Just Build It" Philosophy
- Design Is Implicit
- Design Is not Necessarily Documented
- Lots of Leverage
- Same Team
- Little Structure, Low Overhead, and Little Bureaucracy

Challenge:

- Make design explicit and do enough of it to accelerate the schedule.

P_i	=	Different Products in the Same Product Line
S	=	Start of Project
I/L	=	Transition Checkpoint from Investigation to Development
M/R	=	Manufacturing Release
TTM	=	Time to Market
TBSP	=	Time Between Successive Products

documented. Little structure and formality work reasonably well for small development teams as long as there is continuity between the initial product team and the teams that develop subsequent products. In fact, very often the initial team and the teams for follow-on products are the same. This continuity of individuals and teams enables both design and code leverage between projects.

In paradigm I, the time to market (TTM) is defined as the time between the start or initial staffing of the project and its release to customers. Organizational learning and leverage between any two successive projects can reduce the TTM for the latter project. Thus, if a similar amount of functionality is contained in both projects one expects the TTM for project N + 1 to be less than the TTM for project N. Ideally, the bulk of the effort invested in a latter project is directed at those value-added, differentiating features that are visible to customers.

An organization can choose any number of different development methodologies or life cycles for its development effort. Within HP, many organizations are adopting an evolutionary delivery approach as opposed to a waterfall model (see ***Article 5***, ***Article 3*** and reference 2). In fact, even those using a waterfall model have modified it to support increased concurrency and reduce the impact of a reset at any stage.

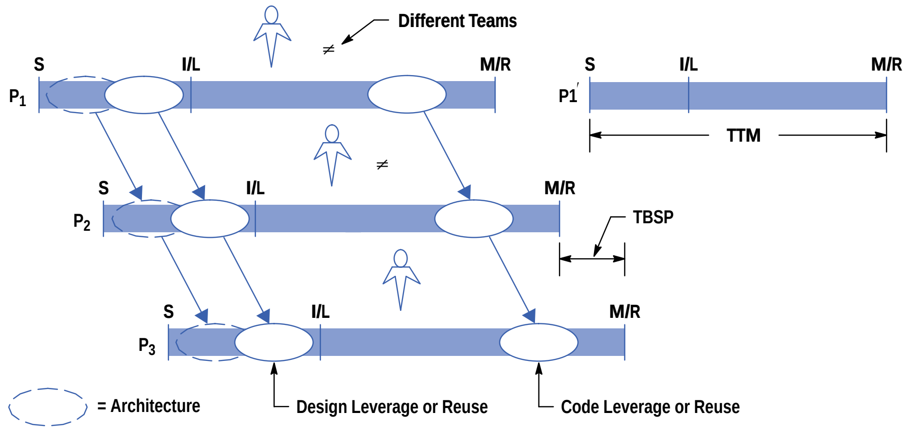
The time between successive products (TBSP) is calculated by subtracting the release date of the earlier product from that of the later product. This measure is referred to as time to market prime (TTM′) by some organizations. In the case of serial independent projects TBSP is equivalent to the TTM of the last product.

Paradigm II: Multiple Parallel Projects

When a business finds increased market acceptance of its products and has market penetration beyond the innovators and early adopters, it is common for customers to demand follow-on products with shorter intervals between them. The pressure to do faster product releases and thereby cut TBSP almost invariably results in a shift within the product development organization to multiple independent projects. We call this paradigm II (see Fig. 3). This situation usually results in different teams working in parallel to build closely related products.

In paradigm II, the TTM does not necessarily change, but the TBSP shrinks because of the overlap between projects. Customer demands for frequent product releases and consistency within a product family put pressure on the product development organization to achieve an appropriate level of consistency across products and shrink both TTM and TBSP. The potential to reduce the TTM for follow-on products exists if there is a high degree of leverage from previous products.

Fig. 3. *Paradigm II, multiple parallel projects.*



Only One First Product per Portfolio (Customer View)

Characteristics:

- Parallel Development
- Project Independence
- Local Optimization
- Products Look Similar
- Design Is Probably not Explicit
- Different Teams, Some Leverage
- Architecture Benefits Accrue to Follow-on Projects

Challenge:

- Make leverage or reuse of both design and code happen predictably.

P_i	=	Different Products in the Same Product Line
S	=	Start of Project
I/L	=	Transition Checkpoint from Investigation to Development
M/R	=	Manufacturing Release
TTM	=	Time to Market
TBSP	=	Time Between Successive Products

Since leverage fundamentally looks into the past for existing assets to draw upon, there is no guarantee that the software found will not require require extensive modifications to work for the new product. Thus, the benefit of leverage is subject to an inherent limitation and in the worst case may be negative (i.e., when the cost of leverage exceeds that of a new implementation).

A much greater reduction in TTM can be achieved if extensive reuse is possible. Reuse fundamentally looks to the future and orients development around what follow-on products will require. Since reusable software components are designed with the future in mind, they can be plugged into new products without any modifications. This is known as black-box reuse because the component user's primary concern is with the external behavior and interfaces and not with the internal details of the component. The practices of leverage and reuse form two ends of a continuum in terms of benefit to recipient project teams. In general, the benefits for project teams that reuse components are greater than those that leverage components. However, components that are not specifically designed for reuse typically need to be modified and hence end up being leveraged. There are significant differences in the development processes used to build reusable components.

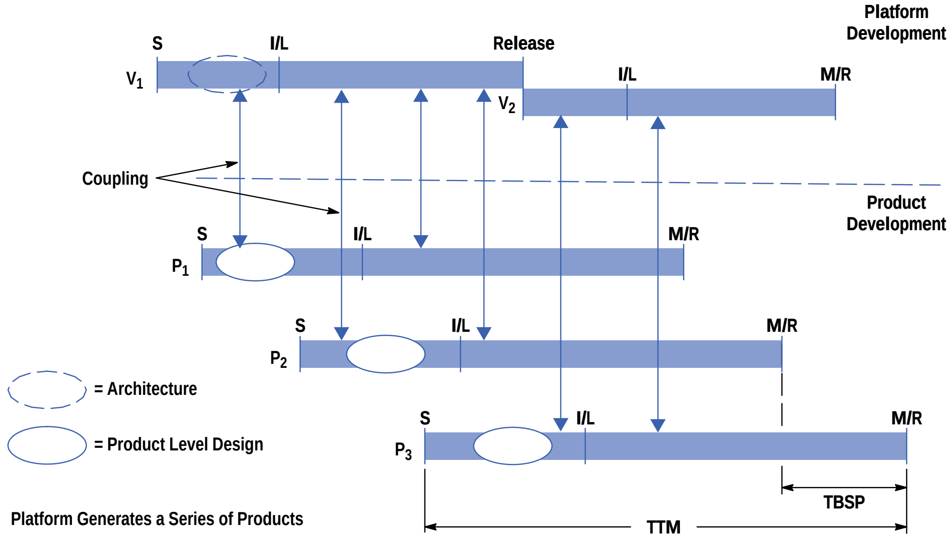
The challenge for organizations doing multiple independent but related projects in parallel is to make the practices of leverage and reuse happen predictably across projects. These practices can happen at multiple levels, from the sharing of architecture and high-level designs to object code and test vectors. The larger the granularity of work shared, the greater the impact on reducing project TTM. For example, reusing a complete error-handling subsystem will reduce project effort more than reusing just a few selected error handling routines.

In paradigm II, individual teams usually have dedicated project managers and architects for each product. This configuration provides each team with a large degree of independence and autonomy. At the same time it can also make it difficult to coordinate the sharing of work between teams. Many organizations find that their predominant mode of operation closely follows paradigm II. For businesses with different product lines there may be several sets of independent projects underway at any given time.

Paradigm III: Platform Development

Businesses that have firmly established a presence for their products in the marketplace, have moved beyond the early adopters, and have achieved a deep customer and product understanding may consider moving to platform development, paradigm III (see Fig. 4). Platform development is essentially an extension of paradigm II, where the common elements within a product family are factored out and developed once. The essence of paradigm III is to pull out those product elements, features, and subsystems that are stable and well-understood, and that provide a basis for value-added, differentiating features.

Fig. 4. *Paradigm III, platform development.*



Characteristics:
- Lots of Parallel Development
- Producer/Consumer Model (Interproject Dependencies, Global Trade-offs)
- Separation of Platform Efforts from Product Efforts
- Support and Evolution of Platform
- Architecture and Design Must Be Explicit (Common Artifacts, Clear Hand-offs)

Challenge:
- Manage the Complexity
- Manage the Timing and Resources

P_i	=	Different Products in the Same Product Line
V_i	=	Platform Version
S	=	Start of Project
I/L	=	Transition Checkpoint from Investigation to Development
M/R	=	Manufacturing Release
TTM	=	Time to Market
TBSP	=	Time Between Successive Products

A platform is different from a reuse library in that it has a cohesive, underlying architecture. The exact composition of the platform for any given product family can range from a complete product framework to a collection of subsystems to sets of individual components. The platform's contribution to individual products can vary from 10% to nearly 90%, either in terms of code or development effort. The exact amount and form of the contribution depend on the specific needs of each product family. Products developed using the features and pervasive structures (e.g., error handling and GUI standards) resident within the platform have a much shorter TTM.

The shift to platform development takes the effort an organization normally puts into product basics and reduces it through reuse. Although new functionality and features can be provided by either platform or product software, in cases involving a large degree of uncertainty new features are usually implemented as part of the product. Once the new product functionality stabilizes and is accepted by the marketplace, it can be migrated into the platform. Thus, it becomes available to subsequent product development efforts.

The net result of implementing paradigm III is a reduction in the TTM for individual projects. This reduction coupled with the parallel development inherent in paradigm II allows organizations to shrink their TBSP. This also enables better market responsiveness, and not surprisingly, in mature businesses a whole series of platforms may be developed to support different product families.

Examples

The following two examples serve to illustrate the power of platform development. In the consumer electronics world, Sony Electronics Inc. is a large producer of handheld portable, radio and cassette players. In fact, Sony makes over twenty different Walkman® stereo radio and cassette players that it sells in the United States. Close examination of these products reveals that only a few underlying cassette mechanisms and cases are used for the entire product family. These mechanisms and packaging constitute Sony's platforms, and they enable Sony to generate an assortment of products targeted at a broad spectrum of customer needs extremely rapidly. The incremental investment needed for Sony to bring out a new model is small because of the large amount of reuse offered through its platforms. There are numerous other examples like this in the consumer electronics markets.

Within the computer networking market, HP offers a product called HP OpenView, a software product used for managing complex networks. In HP OpenView's case, parts of the software form a platform that HP's customers use to build network management applications. In addition to offering HP OpenView to other network management vendors, HP also markets its own suite of HP OpenView-based network management applications. The HP division responsible for OpenView produces additional network management applications in under half the time required for a grounds-up implementation. This represents a TTM reduction of over 50%. Third parties working with HP OpenView also experience similar levels of effort and time savings. Unlike HP OpenView, which is sold and used external to HP, most development labs are producing and using platforms internally to provide the foundation for individual product lines.

Changing an organization's development paradigm to platform development is nontrivial and requires a significant investment. Richer, more robust, and more competitive products along with TTM reductions of one third to one half are not uncommon. Coupling the use of a platform with doing multiple products in parallel results in significant reductions in TBSP. The lower limit for TBSP is the rate at which the market can absorb new products.

Platform Competency Model

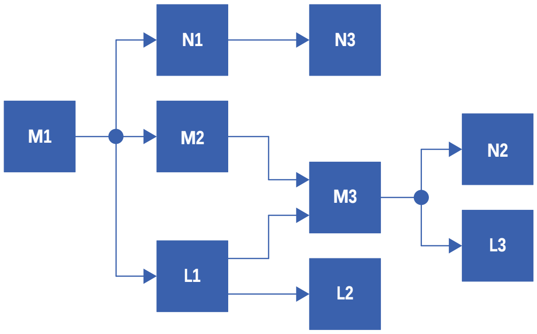
The purpose of the platform competency model is to depict the core elements that make up an organization's software development system (see Fig. 1). Each individual element of the model addresses a particular aspect of how an organization's development system works. The model collectively represents the overall operating model for software development within an organization. At the same time the model is holographic since each of the elements contains references to aspects of the other elements. As a result of this, the model is not amenable to hierarchical decomposition. This will become apparent as each element is reviewed.

Product Portfolio Planning

The heart of the platform competency model centers around what the business requires of the teams developing software. A product development organization is constrained by the dynamics of the marketplace and the competitive environment within which it participates. The combined market and competitive forces determine accepted operating ranges for investment, time-to-market goals, product specifications, ongoing support, and so on. These constraints put limits on development organizations and establish acceptable bounds for TTM, TBSP, and R&D resource efficiency (e.g., engineering years/product).

The result of integrating these constraints and high-level business goals is articulated as part of a business plan. The business plan includes a product vintage chart, which shows target release dates for individual products over a multiyear period (see Fig. 5). Apart from their use in business plans, product vintage charts are often supplemented by a set of product lineage charts showing the hereditary relationships between products and their constituent components. Fig. 6 illustrates the structure used for a traditional lineage chart and Fig. 7 shows a platform-based lineage chart. Separate lineage charts are often constructed to highlight different components of a product (e.g., electrical circuits, software and firmware, industrial design, mechanical assembly, etc.). Lineage charts are also used outside of development labs to depict the evolution of marketing, training, and support materials.

Fig. 6. *An example of a lineage chart that might be used for the software in different versions of the workstation products shown in Fig. 5.*



Notes:
1. M, N, and L are the same products as shown in Fig. 5
2. Typically, the arrows are annotated with software subsystems or component names
3. Branches generate multiple versions of a software component.

The key in planning a product portfolio is deliberate, systematic attention focused on developing a product lineage chart that effectively addresses the organization's product needs. The software version of the lineage chart sets forth what can be accomplished as a result of the organization's platform strategy. Together the product vintage chart and lineage chart provide the framing for the product portfolio plan (see Fig. 8). The plan establishes the targets for the amount of contribution that a platform makes to individual products. This is frequently expressed as both a target effort savings and a target reuse percentage (i.e., the platform will reduce product investment by X engineering months and will provide Y% of the final product code). It also sets forth the goals for TTM (time to market) and TBSP (time between successive products) for an entire product family.

Fig. 5. An example of a product vintage chart that might be used for workstation products.

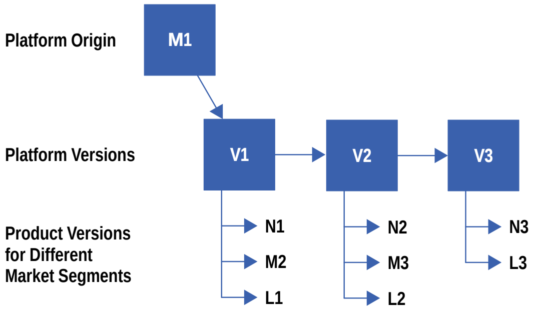
Target Market Segment*	Products							
High-End (e.g., High-Performance Servers)			N1			N2		N3
Midrange (e.g., Servers and Workstations)		M1	M2		M3			
Low-End (e.g., Low-End Workstations)				L1		L2		L3
	Spring	Fall	Spring	Fall	Spring	Fall	Spring	Fall
	1992		1993		1994		1995	

Product Release Dates

* The market segment also determines the product's price.

The product portfolio plan also includes a statement of overall goals, direction, and priorities. Details regarding product definition and customer needs are incorporated by references to individual product plans (see Fig. 8). Thus, the product portfolio plan provides the link between an organization's platform strategy and underlying business goals. As such, it acts to align and unify both platform and development teams and to set the context for individuals in the organization who are charged with working out the implementation details for the platform strategy.

Fig. 7. Platform version of the software lineage chart given in Fig. 6.



Within HP, the portfolio plan tends to be a collection of slides built around an annotated product lineage chart. The product lineage chart is modified to show the flow of software from the platform to individual products and includes a development time line. This package of materials is augmented with details on the goals and objectives for the platform and product teams. The loose structure and informal nature of HP's portfolio plans work well as long as there is constant communication to reinforce the underlying message about the organization's chosen development paradigm and the link between this paradigm and the organization's business goals.

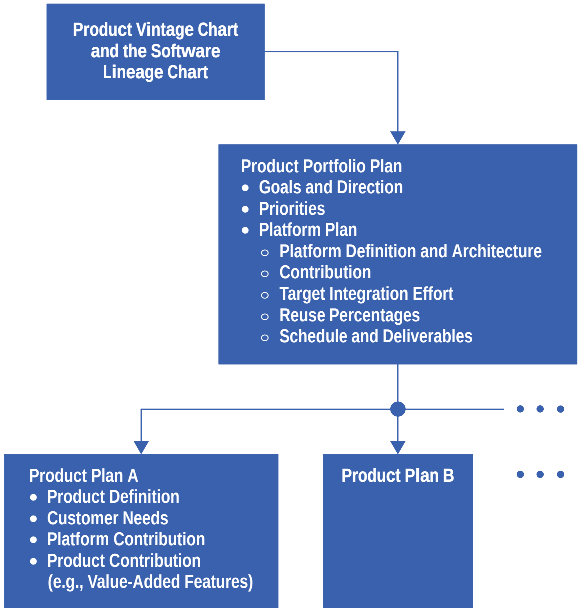
It is essential that all players understand why an organization has chosen the platform development paradigm and what it hopes to achieve as a result of this choice. HP's experience reveals that the rationale and expected benefits leading an organization to adopt the platform development paradigm must constantly be reaffirmed by management. HP divisions that have aligned their development labs around the platform strategy, engendered confidence throughout their organizations, and provided ongoing direction and support during the transition to the new ways of working have adopted platform development more rapidly and achieved better success than divisions lacking active management sponsorship.

Architecture Definition and Partitioning

Since platform development involves separating out the common elements contained within a product family, having a clear and explicit platform architecture becomes very important. This is a key shift compared to traditional product development in which a product's architecture is often implicit and may not even be written down. In fact, often the architectural knowledge of a product or set of products is contained within the head of a single architect or small group of architects. Implicit and informal architectures work for serial independent projects and even for parallel product development when team sizes are small, the level of complexity is low, and the amount of concurrent development is minimal (see Figs. 2 and 3).

These conditions make it possible for the architect or architects to explain the product architecture informally and to assist other engineers as they develop their code. However, increasing complexity and simultaneous pull from multiple teams

Fig. 8. *The components of a portfolio plan.*


Product Vintage Chart and the Software Lineage Chart
Product Portfolio Plan
• Goals and Direction
• Priorities
• Platform Plan
○ Platform Definition and Architecture
○ Contribution
○ Target Integration Effort
○ Reuse Percentages
○ Schedule and Deliverables
Product Plan A
• Product Definition
• Customer Needs
• Platform Contribution
• Product Contribution (e.g., Value-Added Features)
Product Plan B

often forces architects to spend most of their time communicating and helping others with the product architecture. While this would be an extreme in the case of independent products, it is virtually certain to happen in the case of a platform. In addition to supporting others, architects need to have time to focus on evolving and extending the architecture.

The solution to this situation is to have architects document and make explicit the platform and product architecture. Formal architecture documents (diagrams and text) make it possible for engineers to access the architectural knowledge that they need to complete their design and implementation tasks without having to refer to the architects constantly. A documented architecture also provides a means through which development teams can provide feedback to the architects so that they can tune and evolve the platform and product architecture. This not only directly benefits the architects, but also helps to ensure that a set of high-quality and better-integrated products result. Having an explicit architecture also makes it possible to quantify trade-offs between the platform and the product in a systematic way and feeds the management planning processes for current and future products.

Another key distinction of the platform architecture is that it subsumes the partitioning between the platform and platform-based products. In this respect it differs from traditional product architectures which usually do not differentiate between individual products within a product family. Like all architectures, the platform architecture typically includes major functional or technology subsystems and the interfaces between them. The chosen partitioning of architectural responsibility between platform and product teams determines the degrees of freedom that product teams have in their work. The shared challenge for platform and product architects is to determine where to draw the platform and product boundary. The boundary needs to be drawn so that it balances the foundation and the infrastructure provided by the platform with the amount of flexibility needed to support value-added product features.

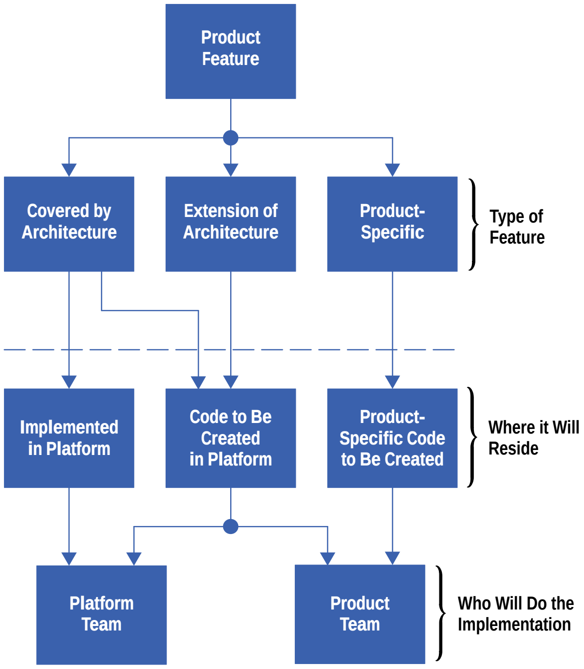
HP organizations struggle in their selection of the initial boundary. In practice their initial partitioning is adjusted over time to achieve the proper balance between platform contribution and product flexibility. Another lesson learned is the need to delineate clearly which interfaces are jointly owned by platform and product teams and which are separately owned. Translating architectural constructs down to the level of interface, module, and code ownership helps avoid conflicting and uncoordinated changes to the platform. It also makes it easier to trade off changes explicitly since it provides a means of linking to all directly impacted teams.

Product Feature Mapping

A closely related area of the platform development model is product feature mapping. This involves the translation of customer needs into product features and ultimately into code implementation. The process of going from customer needs to product feature definition is unchanged from traditional methods. A key shift occurs in the step where product engineers figure out how to map their features to the architecture. Product engineers may no longer have full control in cases where this mapping logically places part of a feature within the product and another part within the platform.

Fig. 9 illustrates the decision-making process that engineers go through to map their features to the platform and product architecture. The ultimate goal of this process is to get engineers to understand what they need to implement and where it falls within the code base. The productivity of product engineers is largely determined by how well they can apply the architecture and move on to implementation. In the case of one HP division with a newly created object-oriented architecture, product engineers were not able to use the architecture at first. The engineers' lack of experience with

Fig. 9. *The processes involved in mapping product features to platform and product architectures.*



object-oriented concepts and the fact that the architecture was separated into logical, functional layers that did not directly correspond to product features made it extremely difficult for them to understand how to use the architecture. As a result, no product development progress was made.

The underlying lesson learned from this experience was the need for the platform architecture to be explained from a product feature perspective, the way in which the product engineers thought about it. This division's dilemma was solved when the platform architect made explicit the process for mapping features to the architecture and taught this to the product teams. What the architect and the rest of the platform team did was to walk through the process and provide direct coaching to help product engineers work through mapping their features to the platform architecture.

An equally critical dimension of product feature mapping is the understanding by product project managers. Since project managers are responsible for the schedule and resource plan for the product, they need to know what work their team will do and what specifically will be provided by the platform. Consequently, the managers must understand the partitioning of work across the platform-product boundary and also the overhead costs of incorporating reusable components from the platform. This information in turn enables them to create an appropriate work breakdown and arrive at a schedule for their project. The underlying shift here is to a new way of product planning. Within HP we have found it extremely beneficial to provide training and coaching to product project managers to help them with their planning tasks.

Test Architecture and Strategy

The platform architecture not only helps define the dividing line between the platform and product, but also drives changes in test strategies and implementation. Since the platform provides components, modules, and subsystems to product teams before final system integration, some degree of testing of platform work products is necessary before they are delivered to the product teams. Ideally, the platform team fully tests its work products so that product teams can focus on their specific extensions to the platform. This division of test effort results in an overall reduction of the effort, defects, and schedule once the first platform-based product is released.

Unfortunately, several factors complicate this ideal. It may be difficult to fully test platform functionality without additional product-specific code, and in many cases platform functionality is developed concurrently with product functionality rather than sequentially. Both result in lower code quality and potentially increase the testing burden placed on product teams. Product teams usually cannot afford to be the de facto systems test organization for the platform because doing so compromises their own goals, and in particular, their schedule. If product teams get overwhelmed with defects or integration problems passed on by the platform team, conflicts in schedule, priorities, and even team relationships arise. Furthermore, since platform and product teams often sit in close proximity, problem solving gets driven by personal priorities and urgency rather than an objective, organized approach. As a result the organization's cumulative testing effort may actually increase, negating any potential savings.

Thus, a key factor of successfully using the platform development approach is the creation of a shared test architecture and strategy that ensures the delivery of high-quality platform components, thereby enabling product teams to focus their development and testing efforts on product-specific features. The goal of the test strategy is not to outline exhaustively the details of how the appropriate level of quality is built into deliverables, but rather to describe the overall risk management and test approach. The test strategy makes reference to specific product milestones, checkpoints, and activities. As expected, code drops correspond to key points of synchronization between platform and product teams. At each code drop, there are specific outcomes, questions, and measures that describe both product and platform goals. Based upon these expectations, testing and risk management activities can be determined. The test strategy specifies what these activities are and when they occur, but not the details about their execution. For instance, the test strategy may call for design reviews and inspections at particular points along the life cycle. The specific kinds of reviews or inspections, to what degree, and of what documents, will depend on the types of risks that need to be mitigated.

The test architecture focuses and streamlines the multilayer, multicycle test process. Each element in the test architecture is linked to the product architecture at the component, subsystem, interface (integration), system, or solution level. The individual elements also serve to set the scope and purpose for test suites. For instance, multiple suites may exist to test subsystems for functionality, usability, performance, or reliability. Once each test architecture element is defined, it gets mapped to the test strategy and reconciled with development and milestone dependencies.

Without a test architecture to define the scope and purpose of a test suite, cycles in the test process will often be redundant, increasing the time and resources used in each product team. A product and test architecture can also facilitate the development of a platform regression test strategy. As a platform is used in more projects, the platform team will want a method of ensuring that changes made to the platform don’t inadvertently impact the functionality of one product over another.

Organizational Structure and Work Partitioning

In addition to the many architectural implications for developing products under the platform development paradigm, there are many management issues that need to be addressed. The spectrum of management concerns is quite broad and includes defining the context within which teams are configured, deliverables specified, and conflicts resolved, and defining how teams communicate with one another. One pivotal management activity is the definition of the organization’s structure and its processes for partitioning work.

Crucial differences in individual roles between platform development and traditional development paradigms are at the heart of structural and work-assignment issues. Platform development does not result in the creation of new roles within an organization, rather it causes existing roles to become explicit and more formal. In traditional development projects, one or more individuals fulfill the roles of program management, project management, people management, product architecture, and process architecture. A brief definition of each of these roles is contained in Table I.

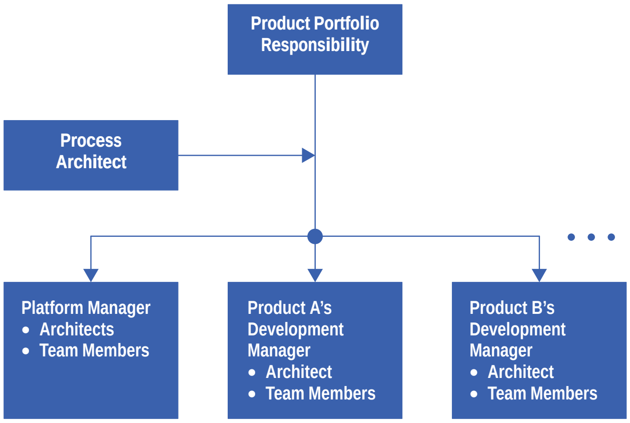
Table I
Product Development Roles

Role	Responsibilities
Program Management	Integrate and coordinate all the functions involved in developing the product, including marketing, development, learning products, and field support.
Project Management	Develop and maintain the project budget and schedule, allocate resources, and manage work assignments.
People Management	Perform employee training, skill development, performance evaluation, salary administration, and other administrative and legal tasks.
Product Architecture	Develop and evolve the product architecture including coaching and mentoring others on its application and use.
Process Architecture	Define and integrate the various processes used during product development, including how work is done, extensive communication, and process training.

Until fairly recently within HP, the roles of project management, people management, product architecture, and process architecture were filled by a single project manager. This is still common in some HP divisions. As a result of increasing product complexity, many divisions have created explicit positions for product architects and have thereby removed this responsibility from their project managers. Some divisions have gone farther and pulled process architecture responsibilities from their project manager positions and added new process architect positions. The separation of responsibilities is especially important for platform development since multiple teams are working in parallel on products within a single family. The larger scale of this endeavor makes it difficult for any one individual to juggle combinations of these roles while simultaneously attending to the broad scope that accompanies each role. As a result, individual jobs tend to be more specialized and better defined in this paradigm.

The key shift in organizational structure for platform development occurs when separate platform and product teams are created (see Fig. 10). The separation of platform and product teams becomes a necessity when there is more than one

Fig. 10. *The organizational structure for the platform development paradigm.*



product under development at any given time. HP's experience has shown that having one team simultaneously develop a platform and a product while another team works on a different product is unworkable because of a perceived lack of trust between teams. The underlying issue here is perceived favoritism by the platform manager for the platform team's own product effort. This perception is hard to counter and requires an explicit way to address conflict of interest issues. Our solution is to separate the product responsibility from the platform to ensure that the platform effort is equally shared between the various products. This solution takes advantage of an organization's reporting structure and relationships.

Just picking an appropriate organizational structure is not sufficient for ensuring that work gets done smoothly. Individual work assignments need to be aligned with the reporting structure. Otherwise, individuals within the organization can wind up spending a lot of their time trying to figure out who does what for whom. In HP's experience, formalizing individual roles and responsibilities helps a lot. It simplifies the tracking of individual accountability and provides the context for optimization of work assignments. In addition, management leadership and direction are needed to help people cope with ambiguity and to address coordination of activities across team boundaries.

Partnership Model and Contract

Management, through its own style and working relations, sets the tone and context for how teams work together. As a result, management behavior largely determines the kind of partnerships that will exist within the the organization. The purpose of having a partnership model and contract is to make explicit how teams are to work together.

The reuse of software components fundamentally involves a producer-consumer relationship in which one or more teams produce software assets that other teams use. In the case of platform development the platform team is the producer and the product teams are the consumers. How the teams work together determines the overall success of their combined effort. Team perceptions of autonomy, accountability, and control all weigh heavily in setting the context and boundaries for how teams can work together.

The starting point for establishing a solid working relationship is given by the organization's existing social and cultural norms. Within HP we have found that if either team seeks to gain sole control of the relationship, the platform development system will fail. The key shift that is needed is for all teams to think and act as full partners. The notion of working together on a collaborative effort creates a win-win situation and avoids the inherent conflict in one team being superior to another. It also blurs the distinction between platform and product team roles and thus provides a greater degree of flexibility in work assignments (refer back to Fig. 9).

While the partnership model defines the preferred working mode between teams, the partnership contract is the ultimate agreement that is reached between product and platform teams with respect to their mutual deliverables. As such, the partnership contract incorporates and builds upon the partnership model. It contains the specifics of what gets delivered, by whom it is delivered, and when it is delivered. It also includes the explicit communication channels that will be used between teams as well as the mechanisms and protocols for handling changes and exceptions. The partnership contract is not a legally binding document, rather it results from discussions between platform and product teams.

Within HP the process of negotiating a partnership contract is more important than the contract itself. For each new product, the corresponding platform and product team members sit down and work through a series of questions to arrive at a mutual agreement that supports the organization's product portfolio plan. The final answers to the questions can then be incorporated into the platform and product team plans as appropriate (see Fig. 8). This negotiation process serves as a mechanism for joint planning and lays the foundation for the ongoing relationship between the platform and product teams.

Management Processes and Steering Teams

Management processes provide mechanisms for managing ongoing team relationships and for addressing changes in both internal and external requirements. The processes include mechanisms for interproject planning and resource prioritization, tracking and controlling progress, and recognizing, escalating, and resolving issues. At an aggregate level these processes need to be aligned with the product portfolio plan. Management is responsible for achieving alignment and for providing their people with the means to work toward the overall strategy.

A key shift in platform development is the creation of standing teams to deal with ongoing issues. In particular, management typically charters a management steering team and an architecture steering team. The management steering team is made up of the portfolio manager, the platform manager, the product team managers, and the process architect. The team may also include the manager of a separate quality or testing team. The team is responsible for monitoring progress, maintaining resources and schedule synchronization, and resolving daily operational issues. The existence of the team is not meant to replace ongoing, one-on-one work. Rather, team meetings serve as a forum for surfacing issues and establishing linkages for issue resolution.

Architecture steering teams are staffed by senior designers, architects, and technical people. They also include an organization's process architects. Their charter is to focus on managing the overlap between the platform and product architecture. The team is responsible for ensuring that the architecture can be used by product teams effectively and for managing the evolution of the architecture so that overall architectural integrity is preserved.

Within HP, management and architectural steering teams are used at multiple levels. The number of teams and their structure depends on the complexity of a division's business, the number of product lines, and the number of distinct platforms within those product lines. Generally, one steering team is created per platform. In our experience steering teams work well when they are staffed with key stakeholders and have clear, well-articulated charters. Management needs to set appropriate team expectations and model new desired team behaviors, especially if they differ from the organization's existing norms.

Communication and Feedback Model

The success of an organization's platform development system is largely a function of the strength of its communication and feedback paths. Good communication between platform and product teams is essential for reducing unexpected surprises and supporting rapid decision making. For communication to be effective, the right information must reach appropriate individuals in the organization at the proper time. Incorrect, inappropriate, or out-of-date information has little value, and in fact, can be counterproductive.

The attributes of a good communication and feedback model are that it:

- Specifies communication roles and responsibilities
- Enumerates the taxonomy of information types
- Identifies explicit communication links, channels, and pathways between teams
- Establishes triggers for certain types of informational exchanges
- Creates a context for continual organizational learning.

The communication and feedback model can be thought of as an architecture for the movement of information within an organization. As such it plays a major role in helping to ensure that the right information gets to the right place at the right time.

The key shift for most organizations is coping with the need for wider dissemination of information. As the number of interdependent teams increases, the number of stakeholders with interest in a given piece of information increases. Putting together a communication model in the form of a data flow diagram helps teams identify who needs to know about plans, assignments, issues, status, best practices, and successes. However, getting information to flow is not enough because the recipients of the information need to be able to respond and act on the information, if appropriate. The challenge for individuals transmitting information is to gather feedback on the effectiveness of their communication and to tune future information exchanges. Having individuals automatically check their communication effectiveness serves to build and promote organizational learning.

Within HP we have seen significant gains in organizational performance as a result of eliminating communication slippage and optimizing its efficiency. Pleasant, clear communication lowers organizational stress and makes it easier for people to work together. It also enables faster and higher-quality decision making. Many divisions are using e-mail and World-Wide Web publishing to make platform information more readily available to product teams.

Support Model

It goes without saying that for product teams to be effective, they must be able to understand, incorporate, and successfully use the platform. This includes the platform architecture, its components, development tools, and the underlying process infrastructure. The support model addresses how product teams get assistance as they work to incorporate platform components into their products. It provides the means by which product teams get help in the following situations:

- Achieving understanding and resolving "how to" questions (e.g., How do you do X? How does Y work?)
- Sorting out instances of something not working as expected or not meeting a product's needs (e.g., There appears to be a bug in X. Can Y be adapted to support product feature Z?)

The creation of a support model starts with the identification of detailed support requirements and establishes the mechanics and logistics for how platform and product teams work together. It includes both initial training and ongoing support during implementation. The support model identifies the types of support provided, the mechanisms through which it is delivered, and the overall service level expectations (e.g., typical turnaround or response time). The model covers activities such as providing documentation, delivering training, answering questions, making defect repairs, and releasing enhancements. It also sets forth support roles and responsibilities—to whom to go for assistance. Finally, it includes an escalation path for resolving impasses.

The concept of a support model is not new. In fact, most organizations have an explicit model for supporting their customers. The key shift in platform development is the creation of an explicit support model for in-house product development work. The need for putting formal structures in place increases with the number of product teams working in parallel. In HP's experience the most effective platform support models are developed jointly by platform and product teams. We have found that including a set of agreed-upon performance measures serves to calibrate and set individual expectations. Typical measures include support availability, response time, and limits on the maximum number of hand-offs. Finally, the support model's scope extends beyond product construction and needs to include product planning and testing.

Platform and Product Life Cycles

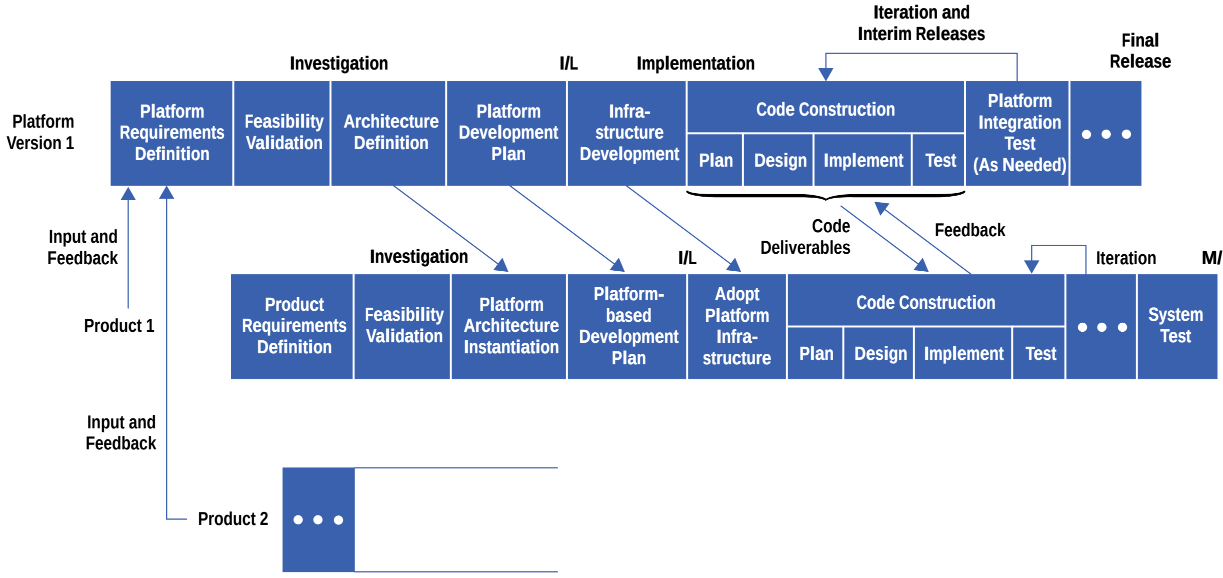
In addition to the need for more formalized support, platform development results in many changes to an organization's development practices. The development process changes are incremental in nature and generally reflect a formalization and refinement of existing practices. An organization's platform and product life cycles provide the structure and context within which individual processes fit. Like other life cycles, they are characterized by major phases with associated goals, activities, deliverables, and checkpoints.

Fig. 11 shows the underlying structures of the platform and product life cycles. The major phases of the two life cycles are very similar. Since the platform architecture and components flow from the platform into products, there is a dependency between the two life cycles. The close coupling of the two life cycles represents a key shift compared to autonomous development projects.

The output of the platform investigation phase is a definition of the overall system architecture and answers to the following questions:

- What is the platform?
- How is it intended to be used?
- How will it be delivered to product teams?
- How will it be supported?
- How will it be extended and evolved over time?

Fig. 11. *The platform and product development life cycles.*



As part of this phase, the platform team identifies necessary changes to existing development practices needed to support platform development. The proposed ways of doing things form the basis for what is called the *platform way*. The deliverables from the investigation phase of the platform need to be essentially complete when product teams move from requirements definition and feasibility validation to instantiating the platform architecture and building upon the platform plan. There is a similar dependency in the implementation phase between infrastructure development and use. The infrastructure consists of the processes and tools that support the platform way.

The implementation phase of the platform life cycle often overlaps with product implementation work. This arrangement requires coordination between iterations of platform and product code construction. The use of an evolutionary development methodology, which subdivides code construction into a series of short plan, design, implementation, and test cycles, provides one way to achieve the needed coordination (see ***Article 3***). In addition to continuous feedback and integration of platform and product components during implementation, continued support for platform use is essential. As mentioned earlier, the test architecture and strategy will determine the overall approach to verification and validation by the platform and product teams.

The platform life cycle does not end when the platform components for all currently active products are finished. Rather it wraps around and a new investigation phase begins. In subsequent iterations the platform architecture, processes, and components are modified and extended based on new requirements. The boundary between the platform and products may change as certain features migrate into the platform so that they can be used by subsequent products.

HP's cumulative experience underscores the need to separate platform and product life cycles. Furthermore, life cycles must be linked to existing hardware life cycles, or software work may not begin until after the hardware prototype is done, thereby delaying product introduction. We have also learned that securing future product input during the development of the overall system architecture and definition of the platform way is essential to building strong interteam relationships. One particularly effective way to engage future product teams is to involve them in signing off on platform checkpoints.

Development Model and Development Process

The development model and the development process describe how new functionality is created. They cover the processes used in the creation and enhancement of a platform module through its integration into products. They are analogous to a module or component life cycle in that they catalog the development steps spanning from construction to final use. Since platform modules and components are ultimately used by product teams, the development process really includes two different perspectives: software asset development and software asset utilization. The key shift in platform development is formalization of these perspectives.

In cases where the boundary between platform and products is diffuse, a single process incorporating both perspectives can be used. The common process can be applied by both teams regardless of whether they were building platform or product functionality. On the other hand, a sharp boundary has the advantage of providing product and platform teams with greater autonomy over their work, since each can use different processes. For example, if the platform-product interface is confined to linking a set of library modules, then the platform might be designed and built using object-oriented methods and tools, while product teams might use traditional structured programming methods and tools. This decoupling is not without cost because the use of different methods and approaches makes it harder for teams to communicate—consider the difference between object-oriented programming in C++ and functional programming in C. Furthermore, its very existence raises the cost of modifying the platform and product boundary and makes it significantly more expensive to migrate functionality across the boundary. It also reduces resource flexibility by making it more difficult to move engineers between the platform and product teams.

Experience within HP has shown that platform development proceeds smoothly when platform and product teams follow highly complementary development processes. By agreeing to use common methods and tools, platform and product teams make it easier for one another to cross the boundary between their work domains. This provides flexibility so that resource use can be optimized. It also allows for evolution of the platform through incremental redefinition of the platform-product boundary. HP division's have reaped significant benefits from having documented development processes since these reduce the support burden for bringing new engineers up to speed on how things are done.

Delivery Model

The delivery model defines how platform modules, components, and subsystems are passed on to product teams. Platform deliveries are essentially a microcosm of the product release process since they cover the depth and breadth of what a development organization delivers at the manufacturing release (M/R) of a product. At M/R, a final production build is delivered to manufacturing along with a set of release notes, documentation, and other supporting material.

Final release may be proceeded by multiple iterations and intermediate releases, especially in the case of complex systems products. Furthermore, each constituent part of the whole product can be delivered in a different way. For example, while physical hardware and firmware go to manufacturing along with detailed assembly, calibration, and packaging instructions, the product's software drivers may simply be given to a third party for replication.

Although most organizations have long had internal hand-offs among teams, the key shift in platform development is making these handoffs and deliveries explicit. While the platform life cycle outlines the types of artifacts that get delivered and when they are delivered, the platform delivery model provides specifics on what is delivered and how it is delivered. If a delivery is analogous to passing a package from one team to another, then the delivery model corresponds to a packing list or bill of materials.

The delivery model also specifies how the package is delivered to the product teams. The delivery mechanism typically falls somewhere along the continuum from push to pull. In the push case, the platform team delivers packages when they are ready. In the pull case, product teams request packages when they want them. There are pros and cons to each approach. The push approach can force product teams to use something before they are ready, while the pull approach can cause the platform team to support multiple versions of the same component. Within HP most divisions have adopted a hybrid model in which both push and pull approaches are used depending on the type of each deliverable.

Although it might be tempting to rely on an ad hoc delivery model, HP's experience shows that having an explicit, formal delivery model is essential to ensure that product teams can successfully use the platform. This is particularly true when multiple product teams are simultaneously using the platform. Furthermore, the delivery model provides additional means of supporting product team members.

Validation and Test Processes

Another important aspect of platform and product development is the accompanying validation and test processes. These processes correspond to the tactics used to implement the overall system test strategy and architecture, including the selection, implementation, and execution of appropriate testing methods and procedures.

In traditional product development, testing is often relegated to the back end of the life cycle and almost becomes a certification step intended to ensure that the product meets certain quality and reliability requirements. In contrast, progressive development organizations view testing as a process of verification and validation that can be applied throughout the product life cycle. These organizations conscientiously perform verification and validation activities as early as possible in their life cycles to catch defects early and thus reduce overall development effort and shorten back-end cycle time.

Organizations that have successfully made the shift to early defect detection see testing for platform development as a variant of their current processes. For others, the move to platform development forces a shift in testing emphasis to architected, early defect detection. When this shift is not made in time, platform development projects become very painful and can even fail.

The key shift needed is for the platform team to certify its work products before they are delivered to product teams. Many testing techniques can be used to ensure appropriate levels of quality throughout the development life cycle. Techniques such as formal design techniques, reviews, and inspections can be used to catch errors before moving into implementation. Prototypes can be used to demonstrate feasibility and validate certain design constructs. Coding standards help to ensure code portability and bounds checking is accomplished via assertions in the code. Downstream activities include white box and black box testing, unit testing, and regression testing. The stability of code modules or product components is verified by doing frequent, regular builds. Finally, integration testing is used to ensure proper cohesion between platform and product components. This list of possible verification methods is by no means exhaustive and many additional types of tests and quality methods exist. It is incumbent on the teams to pick those methods that best address their particular project risks.

Once the platform team has developed processes that meet the delivery criteria of the product teams, the second key shift is for the product teams to leverage and complement the validation and verification already accomplished by the platform team. This shift involves making changes in the product test process to leverage the test architecture and focus test cycles on product-specific contributions and their integration with the platform. If product teams find themselves testing platform components, then something is wrong. This is analogous to having teams that use C compilers do testing on the accompanying C libraries.

Experience within HP indicates that when the product test strategy is not revised, unnecessary redundant testing occurs. In this case the platform is fully tested as part of every product. As the number of simultaneous products under development increases, so does the burden placed on those doing system testing. The end result is repeated testing of parts of the platform, at high cost, and with little added risk reduction.

Development Tools and Infrastructure

The final element of how engineers undertake their work is made up of the development tools and infrastructure that they work with. The tools needed for platform development are no different from those used in other development activities. However, platform development often demands more from the development tools and infrastructure than traditional product development. Since platform development is usually accompanied by a higher degree of complexity, tool flexibility and robustness often become issues. Key shifts in this area have to do with (1) formal tracking of project issues, (2) architecture, design, and process documentation, and (3) robust configuration management.

Ad hoc issue management breaks down in the case of platform development because of the large number of stakeholders external to the platform team. The use of a process and tool to maintain a database of issues and their resolutions not only supports the ongoing issue management, but also provides a means of capturing historical information about key project decisions.

Design documentation and automation tools help with the generation of documentation that is essential for successfully supporting product teams. Standard design tools also help to ensure consistency across the work of the members of the platform team by providing common formats for individual work products.

Finally, tools for simple version control are replaced by a full-fledged source configuration management system that provides the needed horsepower to address concurrent platform and product development needs. In HP's experience moving to a robust source configuration management system leads to new ways of working. The transition to more sophisticated processes for version, build, and workspace management is nontrivial.

Metrics and Measurement Processes

Metrics and measurement processes cut across all the other elements of the platform development model. These processes are used to know how things are going and to flag potential exceptions. Although any organization can collect metrics, the real question is whether or not the organization's metrics are driving effective decision making. A good metrics program measures the right things and then provides a means for acting on the data collected.

Since metrics communicate what is important and focus organizational energy, they provide a mechanism for shaping and institutionalizing the platform way of doing things. Consequently, they provide a means of reinforcing the new behaviors that are required for platform development to be successful. A set of metrics can be defined for the elements of the platform model in such as way that the metrics correspond to how well each element is working for the organization. Defining this set of metrics, rolling them out, and following through correspond to the key shifts in this area.

The management processes and steering teams and the platform and product life cycle directly link many of the metrics to decision making within an organization. An organization's management processes and steering teams provide the context for ongoing review of metrics and for taking appropriate follow-up actions, particularly those related to planning, management, and support issues. With respect to the architecture and development elements of the platform competency model, metrics are included as part of the platform and product life cycles. The metrics for individual elements are tied to specific life cycle checkpoints (refer back to Fig. 10).

The metrics that an organization collects and uses largely depend on its business requirements and level of maturity. A technique that has been particularly effective within HP is the goal/question/metric paradigm (GQM).[2] The principles behind GQM are rather simple. It begins with a goal or set of goals, defines a set of questions that provide visibility into how well the organization is doing at meeting the goals, and identifies a set of measures that provide answers to the questions (see Fig. 12). Since GQM is not necessarily linear, it can be used as a means of exploring and clarifying the meaning of a stated goal. This makes it a powerful technique for defining the metrics for the individual elements of the platform model.

Values and Reward System

In most organizations individuals act based on their beliefs, values, and understanding of the consequences of their actions. Although people do extraordinary things when presented with dire consequences, in most cases an extraordinary level of performance is not sustainable. High turnover and burnout in development organizations are often the result of continually demanding individual and team heroics. The norms for individual and team behaviors depend on an organization's culture which in turn is shaped by the collective beliefs and values of individuals within the organization.

The importance of shaping organizational culture to support platform development cannot be understated. The first step in making the transition to new behaviors for platform development is to identify the vision for how things will be working when platform development is fully functioning within the organization. The vision needs to be vivid, rich, and descriptive. Having a vision is not sufficient to get an organization to its desired state. The organization's culture, values, and rewards must be aligned with the vision.

Fig. 12. *An example of the goal/question/metric paradigm (GQM).*

Goal: Reduce software integration time.

Question1:	How many builds are there in the integration time?
Metric:	Number of builds.
Question 2:	How long does an average build take?
Metric:	Calendar time, engineering effort.
Question 3:	How many unplanned builds are there?
Metric:	Number of planned builds versus unplanned builds.
Question 4:	How many failed builds occur?
Metric:	Number of failed builds versus successful builds.

The organization's recognition and rewards systems play a major role in reinforcing the desired new behaviors that are part of the platform way. It is extremely important to align performance evaluation, recognition, and reward mechanisms with behaviors that simultaneously support achieving platform, product, and business goals. A common failure is to assume that all of these mechanisms and systems are aligned. The key shift is to link the desired state explicitly to specific roles and activities within the organization and to make sure that those roles, activities, and behaviors are reinforced.

Within HP we achieve alignment by working issues both top-down and bottom-up. As a result we integrate business thinking with the logistical, operational, and personal attributes of vision in such a way that individuals throughout the organization know how they fit in. We have also found it necessary to revise existing performance evaluation criteria to tie them to an organization's platform development strategy.

Results

Various divisions within HP have realized productivity and efficiency gains as a result of adopting the platform development paradigm. The degree of improvement varies between divisions and further gains are expected since many divisions are still in the midst of their transition.

One division has doubled its product generation capability from an average of two to four new products per year with no increase in development staff. Another division has cut its time between product releases (TBSP) from twelve to six months and has slowed staffing growth despite an exponential increase in the number of new product releases per year. This division's staff grew linearly while the number of product releases went up almost tenfold over a three-year period. The experience at another division included increased product consistency, improved similarity between products within a product family, and better overall quality. These benefits ultimately enabled this division to offer a better integrated solution to its customers. Furthermore, this division cut its time to market for new products and has reduced the time required to bring new staff on board and make them fully productive.

Fortunately, organizations do not have to wait until they complete the transition to platform development before they start reaping benefits. Since platform development mandates a higher level of organizational maturity and individual skill, it helps to institutionalize solid engineering practices. Typically, there is an incremental productivity gain associated with each improvement in existing engineering practices. Thus, an organization's transition to platform development effectively compounds multiple productivity improvements. Benefits can be reaped all along the journey and not just at the end. In fact, for those HP divisions that have had to cancel or delay their platform efforts, most are better off than before their migration to the platform development paradigm. The reason for this is quite simple: they continue to use better engineering practices on new and existing projects.

In comparison to the business impact of platform development, the results from using the platform competency model in development are not quite as dramatic. The model has primarily been used as a tool for exploring the many facets of platform development. It has proved to be an excellent vehicle for building a shared understanding of what it will take to institutionalize platform development within an organization. The model has also been used as a diagnostic tool to identify areas requiring further attention, thus providing the basis for developing an overall investment plan.

Significant differences exist between the investment plans of different organizations. Differences in the magnitude and sequencing of investments in the elements of the competency model can be traced to differences in an organization's respective level of software expertise, the maturity of its operating practices, and its business constraints. Since each organization has a different profile, its roadmap for establishing a platform capability is uniquely its own. In our experience, when organizations stumble in their adoption of the platform development paradigm, the root cause of their difficulty can be traced to one of the model's elements.

The feedback that we have received as a result of using the competency model with HP divisions has only served to validate the model. There have been a few minor additions to the model, but no significant changes. The unanimous consensus within HP is that each and every element of the model is critical. All sixteen elements of the model are necessary for successful deployment of platform development.

Conclusion

Although numerous product development strategies and paradigms are in use throughout HP, platform development is becoming the paradigm of choice within HP. The platform competency model discussed in this article captures the essence of HP's cumulative experience in platform development. The model is being used by HP product development organizations to understand the requirements and implications of platform development, to guide the creation of investment plans, and to assist in the customization and tailoring of the model's elements to fit each organization's particular situation.

Engineering Perspective

From an engineering and technical perspective, the platform development paradigm is an enabler for technologically superior products. The use of a platform as the base for new products allows engineers and architects to focus their efforts on the key, new technical contributions. The use of a platform also results in more solid products that are higher in quality, better integrated, and more consistent. This degree of robustness results from the continual evolution and improvement of the platform.

Management Perspective

The platform development paradigm results in a number of changes in the way development teams are managed. Since new products are developed using both platform and product component streams, management attention shifts from an intense product focus to a more integrated and global perspective. Managers find themselves actively involved in charting their organization's future, in setting appropriate goals and objectives, and in establishing the necessary supporting infrastructure. Managers through their own actions shape how their organization transitions into new ways of working within the platform development paradigm.

In platform development, managers are principal actors in understanding the trade-offs between product and platform team needs, communicating and making those trade-offs explicit, linking them to short-term and long-term project and business goals, and finally, taking appropriate action. Ultimate success is rooted in the ability of the entire management team to align its thinking and actions around its instantiation of the platform development paradigm. Active participation by multiple levels of management, appropriate acceptance, and support are necessary but not sufficient conditions for successful implementation of the platform development paradigm.

Platform development enables an organization to deliver innovative, feature-rich, high-quality, and consistent products on short development schedules. It does so through increased reuse, reductions in per-product testing, and ever increasing product quality. Simultaneously, it raises overall engineering efficiency, makes it easy to assimilate new engineers, and improves schedule predictability. These gains ultimately translate into reduced development cycle times and shorter time to market.

Organizational and Business Perspective

From a systemic viewpoint, the platform development paradigm is just one of many product development strategies. Increased development efficiency and cycle-time reduction can be achieved in incremental steps by gradually evolving an organization's development competencies. Transitioning to platform development is nontrivial and is in many ways analogous to reengineering the product development process. So even though changing development paradigms is a business imperative for some HP divisions, the shift away from largely independent and autonomous development projects to collections of interrelated projects is somewhat countercultural. Successful adoption of the platform development paradigm requires aligning the organizational culture, values, and rewards to support new ways of working.

There are numerous approaches to adopting platform development. Within HP most development organizations tend to follow an incremental, evolutionary approach as opposed to a complete, ground-up, or reengineering approach. The former supports ongoing, new product development efforts and gradually improves an organization's development capability. As such it reduces the amount of change the organization must assimilate at any given time and stretches the change investment out over a longer period of time.

Regardless of the approach that an organization selects, there are many common issues and challenges that it will face and yet each organization inevitably faces some unique issues and challenges. How an organization goes about implementing the capabilities underlying the model's individual elements will vary depending on its situation. Each organization needs to work out an implementation plan that fits the parameters of its business context.

Acknowledgments

I would like to thank the many individuals within the software initiative who have been working in partnership with HP divisions to implement platform development. I believe that they are truly making a difference for HP by helping divisions reach a new level of competitiveness in their software development effort. The platform competency model is the result of their hard work and the many learnings from actual applications of the model. I would like to extend special thanks to Todd Cotton, Ron Crough, Mark Davey, Joni Ohta, and Mike Ogush for working with me on the definition, application, and evolution of the platform competency model.

References

1. G. A. Moore, *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers,* Harper Business, 1991.
2. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.
3. R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement,* P T R Prentice-Hall Inc., 1992.

A Decision Support System for Integrated Circuit Package Selection

The package provides signal and power distribution, heat dissipation, and environmental protection for an integrated circuit (IC). The process of selecting a package is complicated by the large number of packaging alternatives with overlapping capabilities. To handle these difficulties, a decision support system was developed. The Package Selection System (PASS) combines expert system tools and multiple-attribute decision making techniques. The expert system provides a list of technically feasible alternatives. The multiple-attribute decision making modules are used to rank the alternatives based on nontechnical criteria.

by Craig J. Tanner

Two current trends in the electronic industry greatly increase the need for tools that assist in the selection of IC packages. The first trend is the emergence of a new field of electronics known as "manufacturingless manufacturers" (MLMs). The second trend involves the introduction of competing and overlapping technologies from subcontract packaging services. This trend complicates the package selection process because it creates many technically feasible packages for the decision maker to choose from.

MLMs are only concerned with the design phase of IC manufacturing, and in some cases final package test. All of the support activities associated with ICs, such as wafer fabrication, test, and packaging of the IC are subcontracted outside of the company to foundry services. The finished product is then sold by the MLM to the end user. MLMs have several advantages over full-service semiconductor companies. They can concentrate on doing one thing (design) well and they have no overhead or R&D costs associated with maintaining and developing IC manufacturing processes. Because the engineers who work for MLMs are typically focused on the design phase and are not experts in packaging, an IC package selection system can be a valuable tool.

The introduction of competing technologies from packaging subcontractors has made the selection process more difficult. In the past, there was usually one dominating technology that filled a particular niche. As United States and Asian subcontractors have become more involved in the research and development of new technologies, sometimes several packages that have similar technical attributes have been made available to users almost simultaneously. For example, the packages shown in Table I were released within 18 months of each other and all were aimed at users who need thermally enhanced surface mount packages.

Multiple-attribute decision making (MADM) techniques are incorporated into the decision support system to provide the decision maker with a method for selecting an IC package from among technically feasible alternatives. The MADM modules allow the decision maker to rank the feasible alternatives using nontechnical criteria. This technique can effectively address the emerging trend of simultaneous introduction of competing technologies from different suppliers.

Table I
Competing Thermally Enhanced Surface Mount Technologies

Technology	Company
Metal Quad Flat Pack	OIT
Micro Cool	Motorola
Enhanced Dissipative Quad Flat Pack	ASAT
Power Quad I & II	Anam

System Overview

The Package Selection System (PASS) contains all of the subsystems that are typical components of a decision support system:[1]

- Database management system (DBMS)
- Model-base management system (MBMS)
- Dialog generation and management system (DGMS).

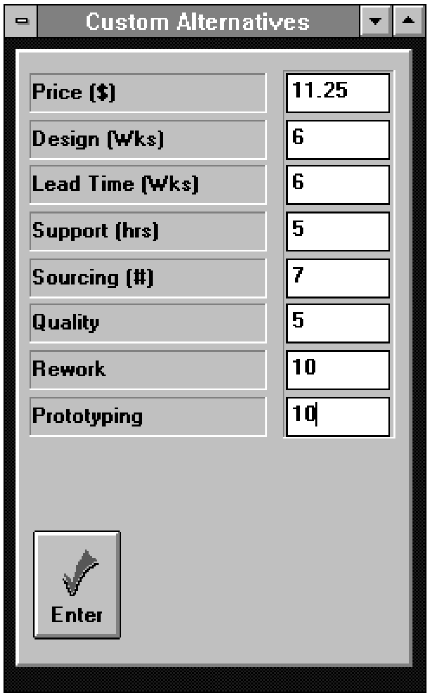
The DBMS is composed of a database and a management system. A database is a file or set of files containing information needed, generated, or manipulated by a computer program. The management system provides the method for creating, accessing, and maintaining the database. The PASS database is an *individual records model.* This model consists of a set of records in which each record contains a set of fields. Each record represents a type of package. Attributes of that package, such as price, are contained in the fields. Because the database is an ASCII file, a separate management system is not required. The database can be updated and maintained through the use of any text editor or word processor that is capable of reading and writing ASCII files.

The MBMS consists of three models. The first model is a knowledge-based expert system. The expert system is used to determine which IC packages are technically feasible. The second and third models use the multiple-attribute decision making techniques known as PROMETHEE and AHP. These models allow the decision maker to rank the technically feasible alternatives based on pricing and other nontechnical attributes.

The DGMS is classified as a *graphical user interface.* A graphical user interface (GUI) is a means of visually communicating between the user and the application. The attributes of visual communication include the graphical techniques used to communicate a concept, a task, a message, the contents of a screen, or any other interface component. The overall look and feel of a GUI is established through the use of screens, windows, controls, panels, icons, menus, animation, and sometimes sound (while sound is not visual, it can be a visual enhancer).

A sample screen from PASS is shown in Fig. 1. This screen is used for entering custom alternatives. Output from PASS sessions is presented in two ways: graphically and as an ASCII text file. The DGMS also contains an online help facility.

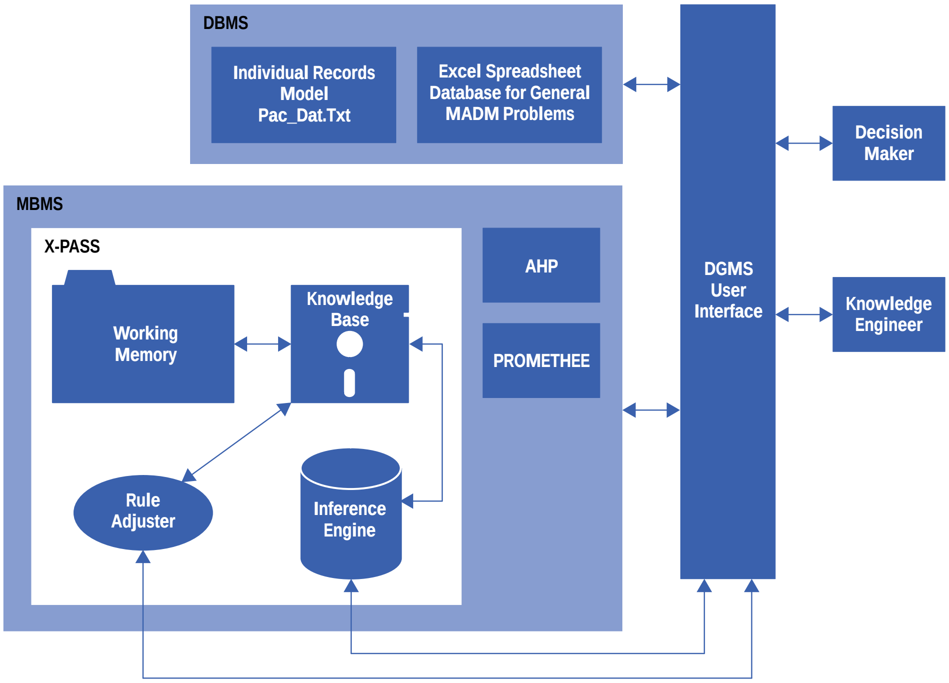
Fig. 1. *Package Selection System (PASS) screen used for entering custom alternatives.*



Price ($)	11.25
Design (Wks)	6
Lead Time (Wks)	6
Support (hrs)	5
Sourcing (#)	7
Quality	5
Rework	10
Prototyping	10

The three subsystems described above must interface with each other and the decision maker to form a complete decision support system. A block diagram of the PASS components is shown in Fig. 2. The MBMS contains analytical tools and heuristic tools. The AHP and PROMETHEE modules are mathematical models while X-PASS employs expert heuristics in the form of a knowledge base and an inference engine. Decision support systems that use both of these problem-solving methods are frequently called *hybrid decision support systems.*[2]

Fig. 2. *PASS block diagram.*



Multiple-Attribute Decision Making

Multiple-attribute decision making (MADM) is the study of techniques that can be used by a decision maker to select a good alternative from a finite number of alternatives when faced with conflicting objectives. MADM techniques are necessary in PASS because X-PASS typically generates multiple alternatives and a method is needed to evaluate these alternatives. X-PASS only makes recommendations based on the technical aspects of electronic packaging. Technical attributes tend to overlap packaging technologies.

Nontechnical attributes such as price and delivery have varying degrees of importance based on the application and the decision maker's objectives for a particular integrated circuit. MADM techniques offer powerful ways of dealing with the decision maker's preferences and for ranking alternatives. For this reason multiple-attribute decision making has been included in PASS.

PASS contains two modules for performing multiple-attribute decision making. The first module uses PROMETHEE (Preference Ranking Organization Methods for Enrichment Evaluation). The second module uses AHP (Analytic Hierarchy Process).

Including both PROMETHEE and AHP in the decision support system allows those users who subscribe to either the French or the American schools of thought to take advantage of that preference. Offering both techniques is also an advantage for those users who have no preference or are not familiar with MADM techniques. These users benefit by examining both methods and selecting the technique with which they are most comfortable. The following sections contain a brief discussion of the PROMETHEE and AHP methods.

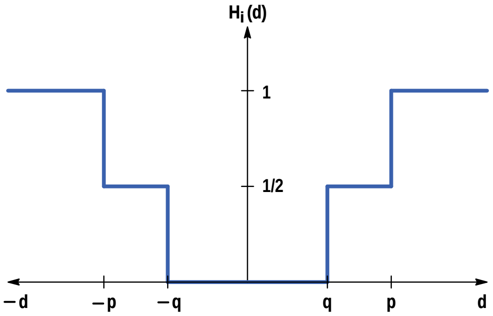
PROMETHEE

The PROMETHEE technique was presented by Brans and Vincke in 1985.[3] PROMETHEE is an outranking method and can result in the partial preordering of alternatives (PROMETHEE I) or the complete preordering of alternatives (PROMETHEE II). The PROMETHEE methods consist of three steps: construction of generalized criteria or preference functions, calculation of the multicriteria preference index, and determination and evaluation of an outranking relation to give an answer to the multiple-attribute problem of interest.

During the construction of generalized criteria, the decision maker must assign a preference function, $P_h(a,b)$, to each criterion f_h, where $h = 1,2,...,k$. The preference function gives the degree of preference of the decision maker for selecting action a rather than action b. Four meanings are given to the preference function:

- $P_h(a,b) = 0$ No preference for a over b, $f_h(a)$ and $f_h(b)$ indifferent.
- $P_h(a,b) \backsim 0$ Weak preference for a over b, $f_h(a) > f_h(b)$).
- $P_h(a,b) \backsim 1$ Strong preference for a over b, $f_h(a) >> f_h(b)$.
- $P_h(a,b) = 1$ Strict preference for a over b, $f_h(a) >>> f_h(b)$.

Fig. 3. *Graph of the generalized criterion* $\boldsymbol{H}_i(d)$*. The parameters p, q, and d are defined in the article.*



For example, if in comparing a $5.00 package a to a $7.00 package b, the decision maker feels that the $2.00 difference between $f_h(a) = \$5.00$ and $f_h(b) = \$7.00$ is insignificant, then there is no preference for a over b and $P_h(a,b) = 0$.

The difference between the two evaluations, d, is equal to $f_h(a) - f_h(b)$. $H_h(d)$ is then defined as:

$$\begin{aligned} H_h(d) &= P_h(a,b), \quad d \geq 0 \\ &= P_h(b,a), \quad d \leq 0. \end{aligned}$$

The function $H_h(d)$ combined with the criterion f_h, that is, $\mathbf{H}_h(d) = \{H_h(d),f_h\}$, is called the *generalized criterion* associated with f_h. Brans, Vinke, and Mareschal[4] have developed six possible types of generalized criteria. While other generalized criteria can be defined, these six should meet most decision makers' needs. They require that the decision maker define only a few parameters.

One of these generalized criteria is defined as follows (see Fig. 3):

$$\begin{aligned} \mathbf{H}_i(d) &= 0, \quad |d| \leq q \\ &= 1/2, \quad q < |d| \leq p \\ &= 1, \quad \text{otherwise,} \end{aligned}$$

where q is the indifference threshold, which represents the largest value of d below which the decision maker considers there is indifference, and p is a strict preference threshold, which represents the lowest value of d above which the decision maker considers there is strict preference.

The second step of the PROMETHEE method is to calculate the multicriteria preference index for each alternative over all criteria. The preference index is defined as:[4]

$$\pi(a,b) = \sum_{h=1}^{k} W_h P_h(a,b),$$

where

$$\sum_{h=1}^{k} W_h = 1$$

and

$$0 \leq \pi(a,b) \leq 1.$$

This index is the mean of the values of the k preference functions. W_h is a weight associated with each criterion. A weak preference of a over b is denoted by the value of $\pi(a,b)$ being close to zero. A strong preference of a over b is denoted by the value of $\pi(a,b)$ being close to one.

The third and final step of the PROMETHEE method, determination and evaluation of an outranking relation, requires that positive and negative outranking flows be determined from the multicriteria indexes. The positive and negative flows are defined as:[3]

$$\Phi^+(a) = \sum_{b \in A} \pi(a,b)$$

and

$$\Phi^-(a) = \sum_{b \in A} \pi(b,a),$$

where A is the set of possible actions.

The following rules can then be used to generate a partial preordering of the alternatives. P means "is preferred to," I means "is indifferent to," and iff means "if and only if."

- aP^+b iff $\Phi^+(a) > \Phi^+(b)$
- aI^+b iff $\Phi^+(a) = \Phi^+(b)$
- aP^-b iff $\Phi^-(a) < \Phi^-(b)$
- aI^-b iff $\Phi^-(a) = \Phi^-(b)$.

The *outranking relationship* is then constructed from these rules:

- a outranks b if aP^+b and aP^-b or aP^+b and aI^-b or aI^+b and aP^-b.
- a is indifferent to b if aI^+b and aI^-b.
- a and b are incomparable otherwise.

Analytic Hierarchy Process

AHP was developed by Thomas L. Saaty in the 1970s. It is based on a set of axioms developed by Saaty[5] and published in a paper by Harker and Vargas in 1987.[6] A good mathematical analysis of these axioms can be found in reference 7. The four axioms were paraphrased by Harker as follows.

1. Given any two alternatives i and j out of the set of alternatives A, the decision maker is able to provide a pairwise comparison a_{ij} of these alternatives under any criterion c from the set of criteria C on a ratio scale that is reciprocal, that is,

$$a_{ji} = \frac{1}{a_{ij}} \quad \text{for all } i, j \in A.$$

2. When comparing any two alternatives $i,j \in A$, the decision maker never judges one to be infinitely better than another under any criterion c, that is, $a_{ij} \neq \infty$ for all $i,j \in A$.

3. One can formulate the decision problem as a hierarchy.

4. All criteria and alternatives that impact the given decision problem are represented in the hierarchy, that is, all expectations must be represented (or excluded) in terms of criteria and alternatives in the structure and be assigned priorities that are compatible with expectations.

Using these axioms, decision applications of AHP can be carried out in two phases: hierarchy design and evaluation.

The first phase, hierarchy design, involves the estimation of weights for each criterion that is used to rank alternatives. Most decision makers are faced with two types of criteria: quantitative and qualitative. For criteria based on quantitative data such as cost or size, the weights can be estimated by normalizing or inversely normalizing the comparison factors for each column of alternatives such that the weights sum to 1:

$$w_{ij} = \frac{a_{ij}}{\sum_{k=1}^{n} a_{kj}} \quad \text{for all } i, j = 1, 2, ..., n.$$

For criteria based on qualitative data, a relative weight matrix can be constructed using Saaty's scale of measurement[6] (see Table II).

The positive reciprocal matrix constructed using a verbal scale technique may contain errors in judgment.[4] Column normalization of this type of data would produce different results depending on which column was chosen.

Saaty's eigenvector method[8] is one technique for generating weights that effectively deals with these errors. The eigenvector method results in final weights that are an average of all possible ways of comparing the alternatives. The weights from the eigenvector method are calculated by raising the matrix of alternatives $\mathbf{A} = \{a_{ij}\}$ to increasing powers of k and then normalizing the resulting system:[4]

$$w = \lim_{k \to \infty} \frac{\mathbf{A}^k \mathbf{e}}{\mathbf{e}^T \mathbf{A}^k \mathbf{e}},$$

where $\mathbf{e}$ is a column vector consisting of all 1s and $\mathbf{e}^T$ is the transpose of $\mathbf{e}$. When w converges, the process is complete and a consistency index can be calculated that is an indication of the magnitude of the errors in the matrix.

After construction of the hierarchies, the second phase of AHP is evaluating the hierarchies to make a decision. Evaluation begins with constructing a final hierarchy of the pairwise comparisons of the criteria. Because the criteria are not usually equally important or quantifiable, Saaty's scale of measurement and eigenvector approach are well-suited to developing the weights for ranking the importance of the criteria. The order of preference can then be determined by summing the relative priorities by weighting them with the overall priority of the given criterion.

Table II
Saaty's Scale of Measurement

Value	Definition
1	Equally important or preferred
3	Slightly more important or preferred
5	Strongly more important or preferred
7	Very strongly more important or preferred
9	Extremely more important or preferred
2,4,6,8	Intermediate values to reflect compromise
Reciprocals	Used to reflect dominance of the second alternative over the first

Using PASS

The Package Selection System is a Microsoft Windows®-based application. It was developed using Microsoft's Visual BASIC, version 2.0. Visual BASIC is a programming methodology that allows the developer to create programs that can take advantage of the Windows graphical user interface (GUI). Visual BASIC applications have the overall "look and feel" of professional Windows applications including pulldown menus, buttons, check boxes, scroll bars, text boxes, and icons. Visual BASIC programs can take advantage of other Windows features including multiple-document interface (MDI), dynamic data exchange (DDE), object linking and embedding (OLE), and dynamic link libraries (DLL).

A typical PASS session is run in the following manner:

1. Select technically feasible alternatives using the expert system, X-PASS.
2. Start the multiple-attribute decision making module by clicking on MADM.
3. Input the number of criteria and alternatives.
4. Select the MADM tool (AHP or PROMETHEE).
5. Input the alternatives. Usually alternatives are selected from those suggested by X-PASS but the user is free to select other alternatives or define a custom alternative.
6. Input the criteria. Several criteria are suggested by PASS for the purpose of ranking alternatives, but the user is free to define custom criteria.
7. Input the information required to describe the decision maker's preferences.
8. Tell PASS to evaluate alternatives by clicking on Eval from the PASS window menu.

Fig. 4. *Hierarchy of the major PASS modules.*

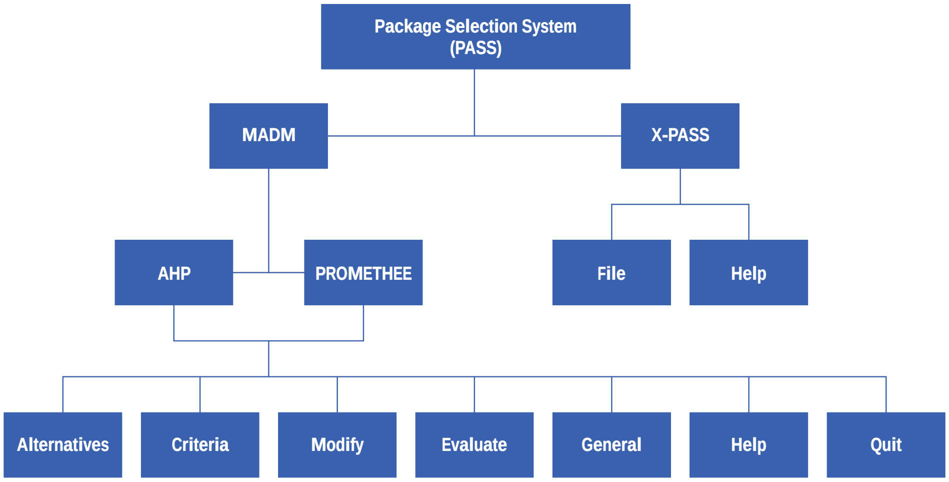
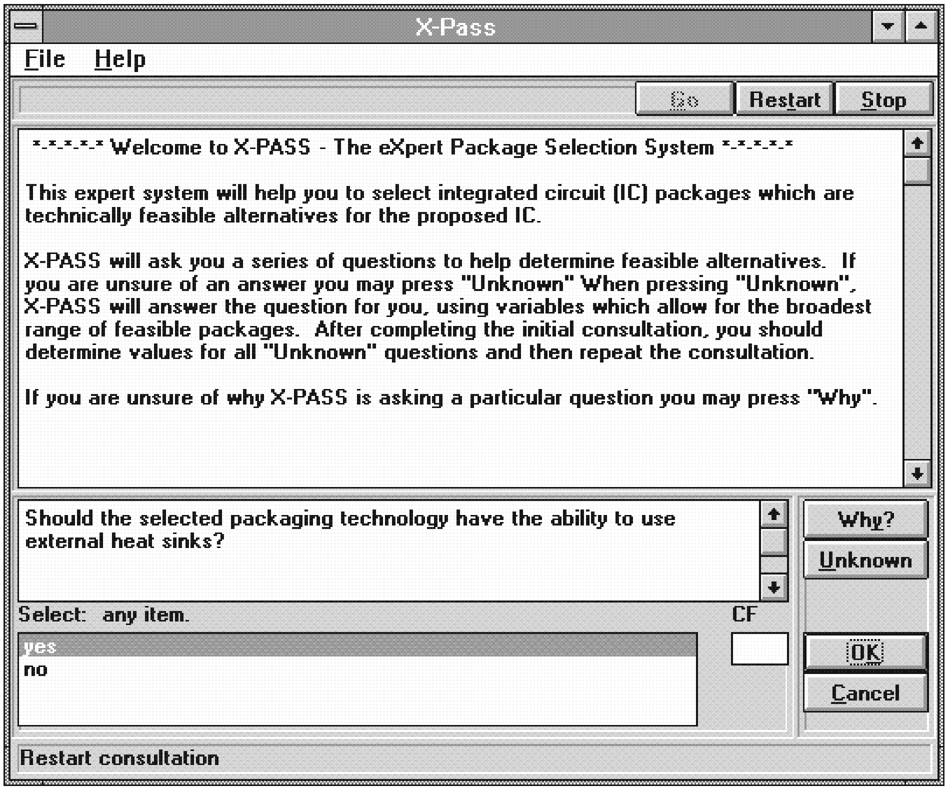


Fig. 5. *X-PASS form.*



--*-*-* Welcome to X-PASS - The eXpert Package Selection System *-*-*-*-*

This expert system will help you to select integrated circuit (IC) packages which are technically feasible alternatives for the proposed IC.

X-PASS will ask you a series of questions to help determine feasible alternatives. If you are unsure of an answer you may press "Unknown" When pressing "Unknown", X-PASS will answer the question for you, using variables which allow for the broadest range of feasible packages. After completing the initial consultation, you should determine values for all "Unknown" questions and then repeat the consultation.

If you are unsure of why X-PASS is asking a particular question you may press "Why".

Should the selected packaging technology have the ability to use external heat sinks?

Select: any item.

CF

yes
no

Restart consultation

PASS contains 18 separate modules for accomplishing this task. The modules are used for inputting data, selecting criteria and alternatives, creating custom alternatives, displaying results, and interfacing with other Windows applications. The hierarchy of the major PASS modules is shown in Fig. 4.

PASS can be started by clicking on the PASS icon or by typing in the proper path and Pass.exe in one of the Windows Run screens. During startup, a welcome screen is displayed. After a brief pause, a module selection screen is displayed. From this screen the user can choose an expert system consultation or select multiple-attribute decision making.

Selection of X-PASS causes the expert system module to start. The knowledge base for X-PASS is automatically loaded. The X-PASS screen is shown in Fig. 5. This form contains four control buttons: Go, Restart, Stop, and Cancel. The X-PASS screen also contains three response buttons: Why, Unknown, and OK.

Go is used to begin a consultation. Restart continues a consultation that was inadvertently aborted. Stop ends a consultation but preserves the current knowledge cache. Cancel ends the consultation immediately and clears the knowledge cache.

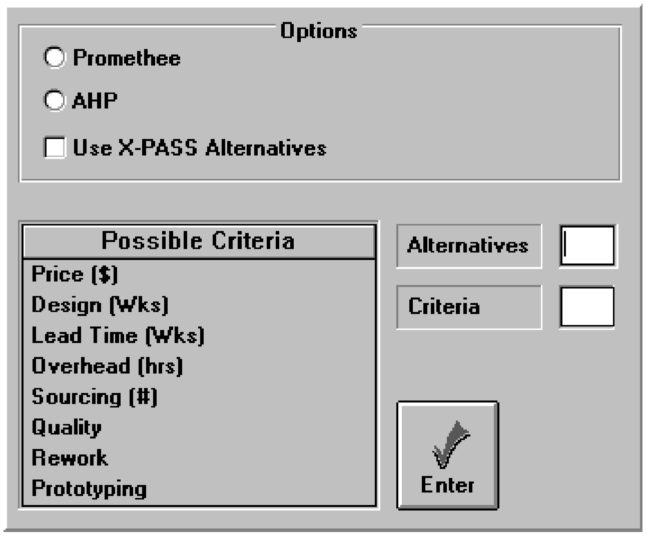
A user can respond to a question asked by X-PASS with Why, Unknown, or OK. A response of Why causes a message to appear that explains why the expert system needs that particular piece of information. Selecting Unknown forces the system to pick an alternative that is the least restrictive in terms of eliminating alternatives. It also associates a low certainty factor with that choice (a certainty factor is a number between 0 and 1 representing the decision maker's confidence in an answer given in response to a question from X-PASS). Selecting OK enters the user's highlighted response into the knowledge cache.

Also on the X-PASS form, certainty factors can be entered in the box labeled CF, and other knowledge bases can be entered into X-PASS through the use of the file menu. This powerful feature allows the X-PASS database to be modified by any user. It also allows X-PASS to be used as an expert system shell for other, nonpackaging-related problems. Feasible alternatives that are determined by a consultation with X-PASS are saved in an ASCII file for optional use by the MADM modules.

Clicking the MADM button on the module selection form initiates the process for using the multiple-attribute decision making tools. The first form in the MADM module (Fig. 6) allows the user to input the number of alternatives and criteria. The AHP or PROMETHEE technique is also selected by using this form. A list box on the form shows the decision maker all of the standard criteria supported by PASS. The user also has the option to check the box labeled Use X-PASS Alternatives. Selecting this option enters the feasible alternatives determined by the most recent X-PASS consultation into working memory. Otherwise, the decision maker can type the number of alternatives and criteria into the text boxes at the righthand side of the form. A maximum of 20 alternatives and 20 criteria can be used. This limitation is a result of the way in which Visual BASIC handles large arrays.

The remainder of the MADM forms are activated by using the menu on the main PASS form. Most of the data entry for PASS is self-explanatory and a help facility is provided to assist in data entry.

Fig. 6. *Initial MADM form.*



Entering Criteria for AHP and PROMETHEE

Criteria for AHP and PROMETHEE are entered in different ways. The criteria input modules are activated by selecting Crit from the main menu. For AHP, two forms are used to enter criteria. The first form (Fig. 7) is for alternatives selection and contains a combination list box labeled Criteria. As the name implies, it is a list of the standard criteria supported by the packaging selection system. After all criteria are entered into working memory, a second screen is activated. This screen, shown in Fig. 8, allows pairwise comparison of the criteria, based on Saaty's scale of measurement.

Comparisons are made by entering a number from Saaty's scale in the matrix in the upper lefthand corner. This causes a row to be compared with a column. The reciprocal comparison is automatically entered into the proper cell. After all comparisons have been made the weights and consistency index can be displayed by using the Eval button.

A consistency index greater than 0.1 indicates that modifications to the matrix may need to be made. The Reset button clears the weights and consistency index but leaves the matrix intact. This is useful if only slight modifications to the matrix are desired. The Clear button clears the matrix as well as the weights and consistency index. Pressing Exit enters all data into working memory and returns the system to the main menu. Saaty's scale is shown in a list box at the lower lefthand corner of the form.

Entering criteria for the PROMETHEE technique is a three-step process. The PROMETHEE form is shown in Fig. 9.

The first step is selecting a criterion from the list box in the criteria section of the form. The criterion is entered into working memory by pressing OK. A range of values for each of the criteria corresponding to the preselected alternatives is shown in the range list box.

The second step involves selecting a preference function by pressing one of the six buttons located in the upper lefthand corner of the form. When a preference function is selected, a graph of the function showing the parameters appears in the screen to the right of the buttons. For the form shown in Fig. 9, preference function four (P4) has been selected.

The third and final step is to enter the appropriate values for the parameter(s) in the text boxes located in the parameters section of the form. The listing of the range of data is useful for this step. All data is entered into working memory by clicking on Record. This process is repeated for all selected criteria.

Fig. 7. *Selecting alternatives in AHP.*

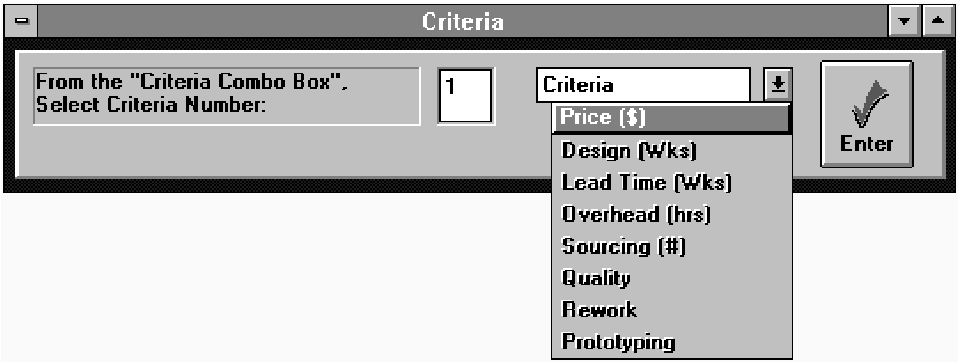
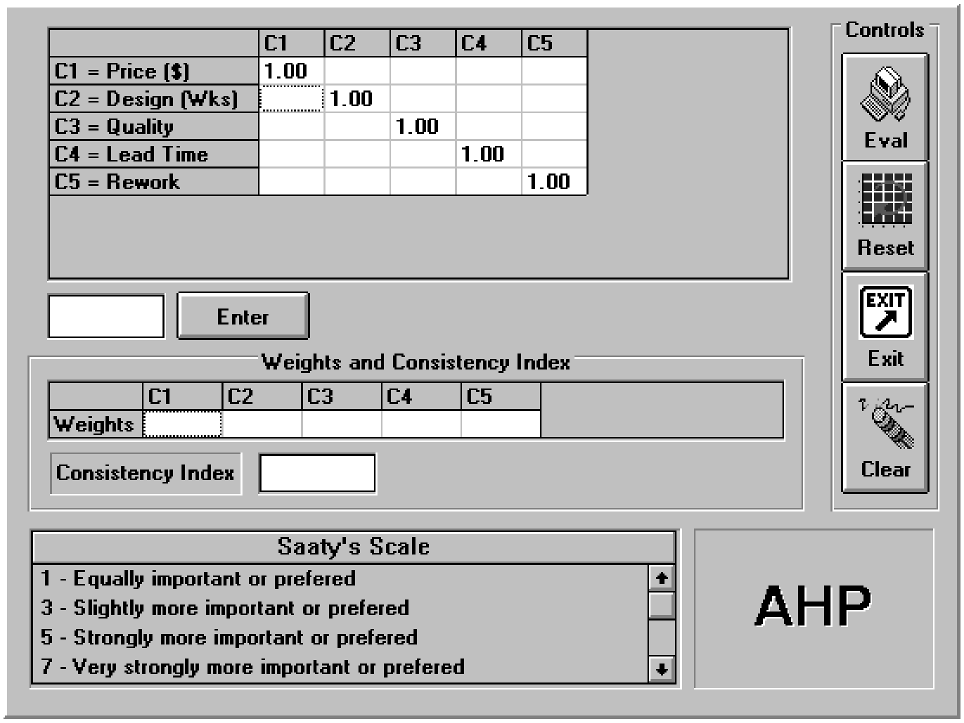

From the "Criteria Combo Box", Select Criteria Number: 1

Fig. 8. *Making pairwise comparisons in AHP.*



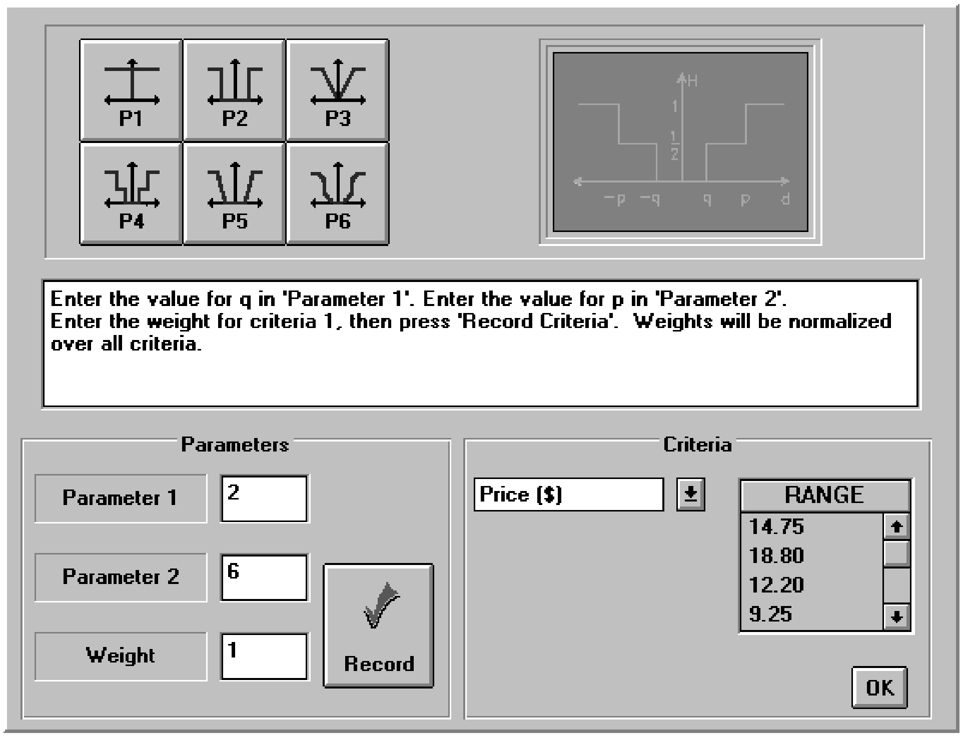
	C1	C2	C3	C4	C5
C1 = Price ($)	1.00				
C2 = Design (Wks)		1.00			
C3 = Quality			1.00		
C4 = Lead Time				1.00	
C5 = Rework					1.00

	C1	C2	C3	C4	C5
Weights					

Saaty's Scale

1 - Equally important or prefered
3 - Slightly more important or prefered
5 - Strongly more important or prefered
7 - Very strongly more important or prefered

Fig. 9. *Entering criteria in PROMETHEE.*



Enter the value for q in 'Parameter 1'. Enter the value for p in 'Parameter 2'.
Enter the weight for criteria 1, then press 'Record Criteria'. Weights will be normalized over all criteria.

Parameters

Parameter 1: 2
Parameter 2: 6
Weight: 1

Criteria

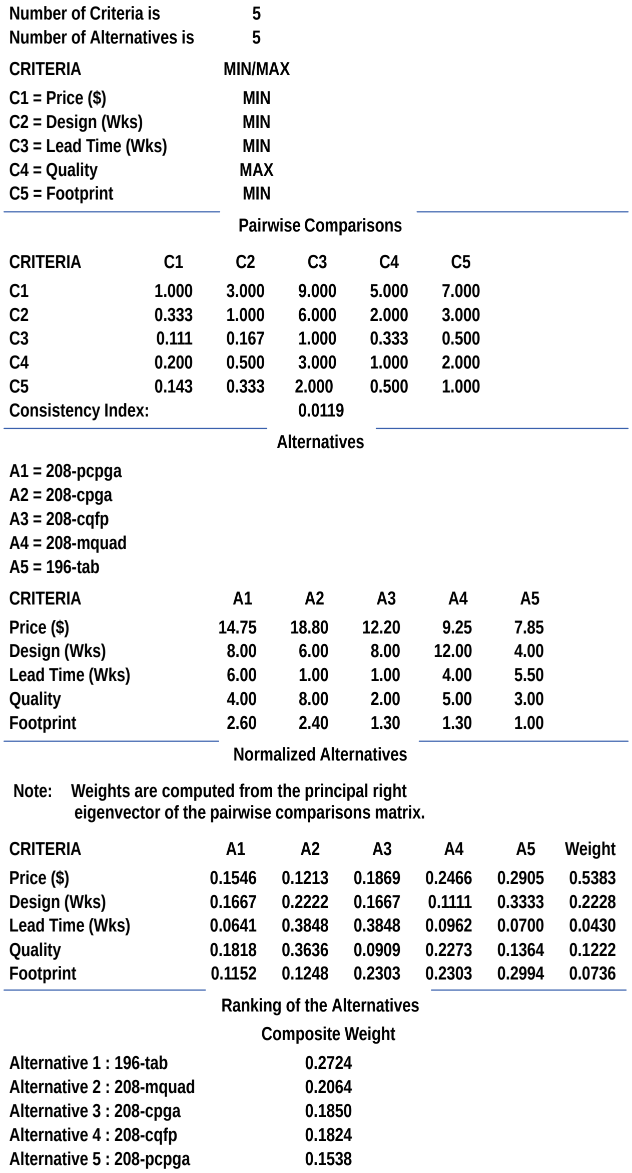
Price ($)

RANGE
14.75
18.80
12.20
9.25

PASS Output

Output from PASS is presented visually and interactively while using the software. ASCII files are also generated; these can be imported into reports or other applications. The output from X-PASS is a list of technically feasible alternatives. With each alternative a confidence factor is reported. The confidence factor ranges from 20% to 100% and is based on the number of questions answered as "unknown." The output from the MADM modules consists of a ranking of the alternatives and their associated weights. A sample of the ASCII output from an AHP session is shown in Fig. 10.

Fig. 10. ASCII output from AHP.



```
Number of Criteria is              5
Number of Alternatives is          5

CRITERIA                        MIN/MAX

C1 = Price ($)                    MIN
C2 = Design (Wks)                 MIN
C3 = Lead Time (Wks)              MIN
C4 = Quality                      MAX
C5 = Footprint                    MIN

                     Pairwise Comparisons

CRITERIA          C1       C2       C3       C4       C5

C1             1.000    3.000    9.000    5.000    7.000
C2             0.333    1.000    6.000    2.000    3.000
C3             0.111    0.167    1.000    0.333    0.500
C4             0.200    0.500    3.000    1.000    2.000
C5             0.143    0.333    2.000    0.500    1.000
Consistency Index:                 0.0119

                         Alternatives

A1 = 208-pcpga
A2 = 208-cpga
A3 = 208-cqfp
A4 = 208-mquad
A5 = 196-tab

CRITERIA                 A1       A2       A3       A4       A5

Price ($)             14.75    18.80    12.20     9.25     7.85
Design (Wks)           8.00     6.00     8.00    12.00     4.00
Lead Time (Wks)        6.00     1.00     1.00     4.00     5.50
Quality                4.00     8.00     2.00     5.00     3.00
Footprint              2.60     2.40     1.30     1.30     1.00

                    Normalized Alternatives

 Note:   Weights are computed from the principal right
          eigenvector of the pairwise comparisons matrix.

CRITERIA                A1       A2       A3       A4       A5    Weight

Price ($)           0.1546   0.1213   0.1869   0.2466   0.2905   0.5383
Design (Wks)        0.1667   0.2222   0.1667   0.1111   0.3333   0.2228
Lead Time (Wks)     0.0641   0.3848   0.3848   0.0962   0.0700   0.0430
Quality             0.1818   0.3636   0.0909   0.2273   0.1364   0.1222
Footprint           0.1152   0.1248   0.2303   0.2303   0.2994   0.0736

                  Ranking of the Alternatives
                       Composite Weight

Alternative 1 : 196-tab            0.2724
Alternative 2 : 208-mquad          0.2064
Alternative 3 : 208-cpga           0.1850
Alternative 4 : 208-cqfp           0.1824
Alternative 5 : 208-pcpga          0.1538
```

Acknowledgments

The development of PASS would not have been possible without the guidance of Dr. John Labadie, who provided expertise in multiple-attribute decision making, and Dr. Thomas Siller, who offered advice on expert systems and inference engines. Both Dr. Siller and Dr. Labadie are professors of civil engineering at Colorado State University.

References

1. A.P. Sage, *Decision Support Systems Engineering*, John Wiley and Sons, Inc., 1991.
2. J.P. Ignizio, *Introduction to Expert Systems. The Development and Implementation of Rule-Based Expert Systems*, McGraw-Hill, 1991.
3. J.P. Brans and P.H. Vinke, “A Preference Ranking Organization Method (The PROMETHEE Method for Multiple Criteria Decision Making),” *Management Science*, Vol. 31, 1985, pp. 647-656.
4. J.P. Brans, P.H. Vinke, and B. Mareschal, “PROMETHEE: A New Family of Outranking Methods in Multicriteria Analysis,” in J.P. Brans, ed., *Operations Research ’84*, Elsevier Science Publishers, 1984, pp. 477-490.
5. T.L. Saaty, “Axiomatic Foundation of the Analytic Hierarchy Process,” *Management Science*, Vol. 32, 1986, pp. 841-855.
6. P.T. Harker and L.G. Vargas, “Theory of Ratio Scale Estimation: Saaty’s Analytic Hierarchy Process,” *Management Science*, Vol. 33, 1987, pp. 1383-1403.
7. R. Saaty, “The Analytic Hierarchy Process—What it Is and How it Is Used,” *Mathematical Modeling*, Vol. 9, 1987, pp. 161-176.
8. T.L. Saaty, *The Analytic Hierarchy Process*, McGraw-Hill, 1980.

Cycle Time Improvement for Fuji IP2 Pick-and-Place Machines

Some of the major enhancements are eliminating head contention, reducing or eliminating nozzle changes, supporting user-defined nozzles, supporting large nozzles for holders 2 and 3, and being able to define multiple part data for a given part number. The cycle time improvement exceeds the original goal of 5%, and the result at one surface mount center was more than 16% over hand-created and optimized recipes. The solution helps both the high-volume and the high-mix centers.

by Fereydoon Safai

Reduction of placement cycle time in an assembly line is one of the major goals in a surface mount shop. It is more important in a high-volume shop than in a high-mix shop because most of the assembly time is spent in part placement. The reduction of placement cycle time at high-volume centers would have a higher impact than at our high-mix centers.

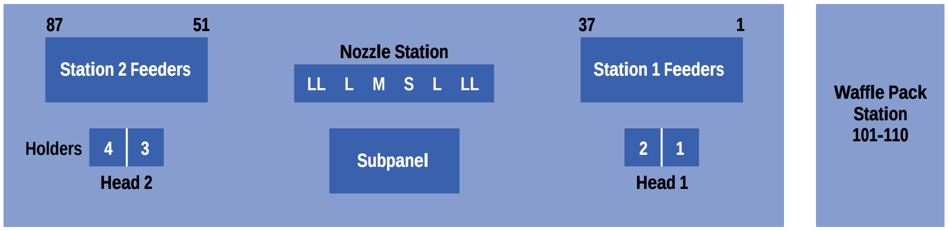
HP owns many Fuji IP2 machines at our surface mount centers, one on each line. The Fuji IP2 machine is a fine-pitch pick-and-place machine capable of placing parts from reel, stick, and waffle feeders. It is considered a general-purpose pick-and-place machine because of its ability to place a wide range of parts. It has two heads, which alternately pick up parts from the feeders and place them on the panel. Each head has two holders, one with a fixed nozzle and one with an automatic nozzle. A fixed nozzle must be installed into the fixed holder before the machine starts placing parts. An automatic nozzle of size S, M, L, or LL can be picked up by the automatic holder from a nozzle station. The nozzle station has six nozzles: one S nozzle, one M nozzle, two L nozzles, and two LL nozzles. The S and M nozzles are shared between the two automatic holders of the two heads. Each automatic holder has its own L and LL nozzles; they are not shared between the two automatic holders.

Since the S and M nozzles are shared between the two automatic holders, it is important that the sequence of placement be arranged so that the two automatic holders do not require the S or M nozzle at the same time. If they do, depending on the particular Fuji IP2's firmware, either one side will halt until the other side finishes its placement and releases the nozzle, or the IP2 software will crash. In either case, head contention is created, which is a problem for IP2 placement machines.

The Fuji IP2 machine has slot numbers 1 through 37, 51 through 87, and 101 through 110. Slots 101 through 110 are used by the waffle unit. If the waffle unit is installed, it interferes with the machine and makes slots 1 through 3 inaccessible. Slots 1 to 37 and 51 to 87 can be used for mounting either reel feeders or stick feeders. Slots 101 to 110 are used for waffle feeders.

The IP2 machine has a number of of constraints. Only one of the automatic holders (holder 1) can access waffle parts from slots 101 to 110. This holder can also access slots 4 to 37 if a waffle unit is installed or slots 1 to 37 if no waffle unit is installed. The other automatic holder (holder 4) can access slots 51 to 87. One of the fixed holders (holder 2) can access slots 4 to 37 (not 1 to 37) and the other fixed holder (holder 3) can access slots 51 to 84 (not 51 to 87). The two fixed holders 2 and 3 can pick up parts up to 3.5 mm in height and the two automatic holders can pick up parts up to 10 mm in height.

Fig. 1. *Fuji IP2 pick-and-place machine layout.*



The Problem

The issues related to the Fuji IP2 are in two categories. One category consists of the issues that reduce the placement cycle time, such as use of the next device, use of multiple part data, the ability to assign a part to both sides, and the ability to assign placements to holders based on reference designators. The other category consists of the issues that make the machine perform correctly. The main item in this category is head contention. If head contention occurs, for certain IP2 firmware the machine halts and the user must change the sequence of the recipe to run the machine again.

The Methods

In this section, we describe our solution for each issue related to the Fuji IP2 machine. This includes elimination of head contention, support of the next device mechanism of the IP2, support for user-defined nozzles such as a modified medium nozzle for holders 2 and 3, use of multiple part data for a part number, and assignment of a part to both sides of the machine to achieve a better head balance.

Our general solution is as follows: we assign placements to holders to balance the load among the four holders. In the process, we consider first the placements that have slots already assigned. This is the case when the part numbers are in the input setups. Then we assign slots to those placements that do not have slots in the input setup. This general solution is used for both setup and sequence modules. In the sequence module, all the placements have their slots assigned.

In the following sections, we describe many issues which are considered when assigning placements to holders. Our solutions have been incorporated in setup and sequence generation modules for the Man-Link recipe generation system, which is used by all HP surface mount centers that have Fuji IP2 machines. Other Man-Link enhancements are discussed in *Article 9*.

Eliminating Head Contention. Head contention occurs when holders 1 and 4 need the same nozzle (S or M) at the same time. At this time, the machine behaves differently depending on what firmware it has. Machines having older versions will stop and the user must edit the recipe to eliminate the head contention. In newer versions, the holder that needs the S or M nozzle must wait until the other holder finishes its placements and releases the S or M nozzle.

In our solution, we do not assign parts requiring the S or the M nozzle to both holders 1 and 4. We assign each nozzle only to one of them, depending on the loads of the holders. This way, the machine in the worst case will place all such parts with one side. This would not be worse than the case in which one side must wait until the other side finishes.

As an example, assume that there are two parts, each requiring the M nozzle and each having 10 placements. Further assume that holders 2 and 3 do not have any nozzles attached to them. Assigning both of these parts to one side, say to holder 1, would not be worse than the case in which one part is assigned to holder 1 and the other part is assigned to holder 4. Using our solution, Man-Link assigns both parts to one side, say holder 1. In this case, the machine will go back and forth and place one part at a time for total of 20 round trips. If one part is assigned to holder 1 and the other is assigned to holder 4, and if the machine has the latest firmware, holder 1 will pick up the M nozzle and go back and forth and make its 10 placements. While holder 1 is placing parts, holder 4 must wait until holder 1 releases and replaces the M nozzle. At that time, holder 4 will pick up the M nozzle and go and make its 10 placements. The machine has to do two nozzle changes for each board, both of which are unnecessary. Our solution does not need any nozzle changes in this particular case, thereby saving 10 to 15 seconds.

Our solution not only eliminates the head contention, but has the additional benefit of eliminating nozzle changes because of head contention. In the past, users would not assign to the IP2 parts requiring S or M nozzles because of the head contention issue. This caused the IP2 to be underutilized. Now users can assign such parts to the IP2 when necessary and this will help to reduce the overall cycle time of products being built.

Using Slot Link (Next Device). With Fuji IP2 machines it is possible to place multiple feeders containing the same part on the machine and have the recipe reference one of the slots. When the parts from that slot are depleted, the machine goes automatically to the next slot that has that part and continues placing.

This slot link mechanism is very helpful for waffle parts. Since the waffle feeders do not take many parts, the operator frequently has to stop the machine and replenish the parts. If a recipe uses only one fine-pitch part from the waffle unit, all 10 waffle feeders of the IP2 can be filled at once and all of these parts can be used before the machine needs to be stopped to refill that part.

The mechanism provided by Man-Link is as follows. The user enters, in an input setup, the slots that a particular part is to occupy and then Man-Link takes over and creates the recipe appropriately. As an example, if a part is assigned to all 10 slots of the waffle pack unit, we would have a recipe containing the slots and slot links (next devices) shown in Table I.

As shown in Table I, a circular link is created between slots 101 through 110. The next slot for slot 101 is 102, the next slot for slot 102 is 103, and so on. The last slot, slot 110, is linked to slot 101, the first slot, to create a circular link. The operator will fill all ten trays with part 1. In the recipe for this part, slot 101 is referenced. The machine starts picking up parts from slot 101 and places them on the board until all parts are used up. Then the machine will go to the next slot, which is slot 102, and start placing parts. This will continue until parts from all ten feeders are depleted. Then the machine will stop, the operator will fill all ten trays, and the cycle will begin again.

Table I
Slot Link (Next Device) Example

Part Number	Slot (Device)	Slot Link (Next Device)
Part 1	101	102
Part 1	102	103
Part 1	103	104
Part 1	104	105
Part 1	105	106
Part 1	106	107
Part 1	107	108
Part 1	108	109
Part 1	109	110
Part 1	110	101

User-Defined Nozzles. Holders 2 and 3 take a fixed nozzle. The nozzle sizes that Fuji supplies for these two holders are S and M. One of our surface mount sites created a larger nozzle for these two holders. This expanded the capability of the machine so that it can pick up as many as four larger parts, while the original machine can pick up only two larger parts. Although the nozzles are larger, the Fuji machines require that these nozzles still be called S or M in recipes. Another issue is that these larger S and M nozzles must be distinguished from the S and M nozzles that holders 1 and 4 can pick up from the nozzle station. For these reasons, we provide a mechanism that allows the user to define any name for a particular nozzle and then map it to one of the Fuji-recognized nozzles (S, M, L, LL).

Our mechanism is as follows. The user defines what nozzles are installed in the nozzle station on the machine and what nozzles are installed in the two fixed holders. The names of the nozzles are user-defined. We have provided six configuration parameters to name the six nozzles in the nozzle station and two configuration parameters to name the two fixed nozzles, as shown in Table II.

Table II
Configuration Parameters to Name the IP2 Nozzles

Configuration Parameter	Description	Possible User-Defined Nozzle Names
LEFT_LL_NOZZLE	The left LL nozzle in the nozzle station	LL, ML, XL
LEFT_L_NOZZLE	The left L nozzle in the nozzle station	L
M_NOZZLE	The M nozzle in the nozzle station	M
S_NOZZLE	The S nozzle in the nozzle station	S
RIGHT_L_NOZZLE	The right L nozzle in the nozzle station	L
RIGHT_LL_NOZZLE	The right LL nozzle in the nozzle station	LL, ML, XL
HOLDER2_NOZZLE	The nozzle in fixed holder number 2	S, M, Modified Medium
HOLDER3_NOZZLE	The nozzle in fixed holder number 3	S, M, Modified Medium

In Man-Link, a part number is linked to Fuji part data. In the part data, the user defines a nozzle (like Modified Medium) and then links this nozzle to a Fuji nozzle, which might be M. We use the user-defined nozzles of each part to assign them to nozzles of the machine, which are also specified by the user.

In Table III, the actual dimensions of the different nozzles used at different HP surface mount sites are listed.

Multiple Part Data for a Part. From the discussion of the previous section, it is obvious that a part might be picked up successfully by multiple nozzles. For example, a part might be picked up by both Modified Medium and ML nozzles. Their nozzle diameters are very close: 6 mm and 7 mm, respectively. For this reason, we have provided part data preferences such that the user can decide which part data (and in turn which nozzle) is the best for picking up a part, and give it the highest preference. The user can then provide additional part data, using other nozzles sizes, for that part with lower preferences. Our software will try to use the part data with the highest preference, but it will use other part data if it will improve the balance among the four holders.

Table III
Nozzle Sizes Used at HP Surface Mount Centers

User-Defined Nozzle Name	Fuji Nozzle Name	Holders	Nozzle Diameter (mm)	Light Ring Diameter (mm)
SS	S	1,2,3,4	1.0	21
S	S	1,2,3,4	1.3	21
M	M	1,2,3,4	2.5	33
Modified Medium	M	2,3	6.0	33
L	L	1,4	4.0	71
ML	LL	1,4	7.0	71
LL	LL	1,4	10.0	71
XL	LL	1,4	15.0	91

A user who has multiple part data defined in the system might still want to use the part data with the highest preference. We have provided the configuration parameter MULTI_ PART_DATA with values of YES and NO so the user can choose whether all part data for a part should be considered or just the part data with the highest preference.

Assigning a Part Number to Both Sides of a Machine. Occasionally a product will have a part number that constitutes more than 50% of all the placements for the IP2. We have seen cases in which there is only one part assigned to the IP2. If only one part is assigned to the machine, one of the heads will be busy placing parts while the other head is idle, doing nothing. For this reason, Man-Link allows the user to duplicate the part on both sides of the machine and the software will split the reference designators of that part between the two heads to balance the load.

If a machine is used in a high-volume shop for such cases, multiple feeders of the dominant part can be placed on each side of the machine to take advantage of the splitting of the reference designators described in this section and the slot link (next device) feature described earlier.

Assigning Placements to Holders by Reference Designators. Assume that there is only one part assigned to an IP2 for a product and it is placed on the right side of the machine, where holders 1 and 2 are located. Further assume that the part requires an S nozzle, and an S nozzle is fixed into holder 2. In this case, it is reasonable that the reference designators of the part be split between the two holders, 1 and 2. This will speed up the placement cycle time since the right head can pick up two parts and then go and place both of them. This is what our software will do. It attempts to split the placements among holders such that the load is balanced among all holders.

Of course, if that part is duplicated on the left side of the machine and an S nozzle is placed into holder 3, then three holders would be picking up parts and placing. This way both sides of the machine would be used.

Minimizing Nozzle Changes. Reducing nozzle changes is another important task. It was shown earlier that eliminating head contention also reduced the nozzle changes by two. Each nozzle change takes five to seven seconds. In a high-volume shop, it is important that the number of nozzle changes be minimized.

Since holders 1 and 4 have their own L and LL nozzles, it is important to assign all parts requiring the L nozzle to one holder and the parts requiring the LL nozzle to the other holder. The imbalance between such holders determines whether to assign parts requiring L and LL nozzles to one of these holders. For example, if there are parts requiring the L nozzle and their total placements are much more than the parts requiring the LL nozzle, then we assign the parts requiring the LL nozzle to one side and the parts requiring the L nozzle to both sides. In this case, we create one nozzle change on the side containing the parts with L and LL nozzles. The decision to assign the parts with the L nozzle to both sides of the machine or to a single side is made so as to minimize the total placements including the nozzle change cost.

Since the nozzle change time is not a constant time, we use a configuration parameter to set the cost of nozzle changes in terms of placements. The name of the configuration parameter is NOZZLE_CHANGE_COST. It has the default value of 2.5. This means that each nozzle change takes as long as 2.5 placements.

Prerotation and Rescan. Fuji part data has two fields called *prerotation* and *rescan*. If the prerotation field is set, the part must be prerotated before the camera can look at it. Therefore, the head that performs such placements can only pick up one part instead of two, thereby slowing down the machine. We keep track of such parts since they affect the load balance between the holders.

The effect of the rescan is the same as that of the prerotation. If the rescan field is set for a part, the head that picks up such parts can only pick up one part and not two. Again, we keep track of such parts and take them into account for load balancing among the holders.

Slot 34 and 54. Because of the construction of the Fuji IP2 machine, part placements from slots 34 (by holder 1) and 54 (by holder 4) have the fastest cycle time. Therefore, it is important that the high-count parts are assigned to these two slots. If the parts are assigned to holders 2 and 3, this does not hold. For holder 2, a part at slot 37 has the fastest cycle time, and for holder 3, it is slot 51.

Using slots 34 and 54 for high-count parts has the drawback of potentially wasting one to two slots, so if feeder space is at a premium this mechanism is not appropriate. However, if feeder space is not an issue, the cycle time can be reduced by using this mechanism. This is especially important for high-volume shops. We have provided a configuration parameter for users to choose whether to use this feature.

Slot Numbers. The center-to-center distance of the two holders of each head is 63 mm. The pitch of the feeder bank is 21 mm. The fastest pickup occurs when the parts for two placements made by the two holders of a head are 3 slots away from each other.

After the assignment of placements to holders is completed, we order the placements of each holder by slot number. In general, it may help the cycle time to pick up two parts with each head.

X and Y Coordinates of Placements. Another factor that can help reduce the cycle time is the distance between two placements on the panel. For example, if both holders of a head are picking up two parts, it will be faster if the two placements are close together on the panel. Therefore, after we sort the placements of each holder by slot (as explained above) then we order them by their separation on the panel.

Results and Discussion

One HP surface mount center was editing recipes to balance the load among the four holders. An engineer was doing this task. This was critical because of the high volume of some of the center's products. When our solution was available, the center tried it and got a very good result. Our solution was more than 16% faster than the recipe that was hand-optimized by an engineer. We should note that when this project was funded, the goal was to achieve 5% improvement, but in all cases we have exceeded this initial goal.

As explained earlier, one of the major issues was head contention. Formerly, the user had to edit the recipe to get around this problem. As a result, many users were not assigning certain parts to the IP2. This tended to increase the cycle time of their component placement machines because it increased the load on the fast machines. Since we have eliminated the head contention issue, users have moved more parts to their IP2 machines and have increased their throughput.

At other centers, the next device mechanism has saved 0.5 hours per shift on each line.

For contract manufacturing, especially for high-volume products, the cycle time reduction can provide an important competitive advantage. We can create recipes with our tools and give the optimized recipes to contractors to be used on Fuji machines.

Acknowledgments

The work described in this paper was funded by all HP surface mount centers who have Fuji IP2 machines. All IP2 Man-Link users have helped us to understand the issues described in this paper. Don Martorello, Sheldon Stewart, Pat Manfull, and Jim Hudson have helped in defining many issues related to Modified Medium nozzles and head balancing. All members of the Man-Link team, especially Rick Palm, have contributed to the implementation of this project. Man-Link is one of the products provided by the design automation group of HP's product generation information systems department and managed by Eiko Johnson.

Reducing Setup Time for Printed Circuit Assembly

In 1994, HP's Man-Link recipe-generation system was enhanced to reduce the time required for setting up pick-and-place machines. This was done by ordering the products to exploit the commonality of parts among them and by creating sequences of setups that differ as little as possible from one another. This paper documents the issues and trade-offs and discusses the potential benefits.

by Richard C. Palm, Jr.

Early in 1993, we began an investigation into ways to decrease the cost of setup time for HP surface mount manufacturing centers. Our initial investigation covered a range of options, including:

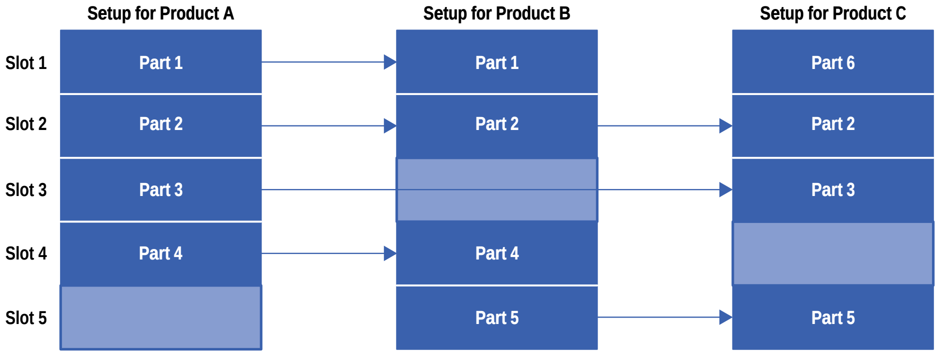
- Common setups for both sides of a printed circuit assembly. All of the parts needed for both sides of the printed circuit assembly are placed in a single setup.
- Partially fixed setups. Commonly used parts are assigned fixed locations on the machines.
- Family setups. All of the parts required to build a group of printed circuit assemblies are placed in a single setup. The printed circuit assemblies in the group are chosen so that all of the required parts will fit into one setup.
- Feeder bank exchange. Some machines offer the ability to change a large number of parts in the setup quickly by means of removable feeder banks. The operator can set up the parts for a product in an offline feeder bank while the machine is building a different product. The operator then trades the offline feeder bank for the online feeder bank, and can start building the new product immediately.
- Optimization by schedule, described below.

We soon narrowed the investigation to two options. We considered these to offer a good return on investment and to be a good fit with the architecture of HP's internal Man-Link system, which creates recipes for pick-and-place machines. The two options were family setups and optimization by schedule. We asked our customers to estimate the benefit to their sites, using mathematical models of these options. Based on their inputs, we chose to proceed with optimization by schedule. This was implemented in 1994.

Setup optimization by schedule takes advantage of the fact that many printed circuit assemblies use common components (Fig. 1). If the machine setup for each new printed circuit assembly is based on the setup of the previous printed circuit assembly, parts that are used in both don't have to be moved, and the total number of parts that need to be set up (called "feeder changes") is reduced. For example, in Fig. 1, four parts each are used on three products. Since parts that are common to multiple products are reused, the operator would only have to do six feeder changes (one each for parts 1 to 6) rather than the twelve that could be required if parts changed slots between products.

Further reduction in feeder changes can be achieved by changing the order in which printed circuit assemblies are built. Printed circuit assemblies with a large number of common parts should be built sequentially.

Fig. 1. *Setup optimization by schedule.*



The advantages of this approach include:

- It works well if feeder space is limited. For example, it has been used effectively in a line containing only one Fuji CP pick-and-place machine and one Fuji IP pick-and-place machine. In contrast, partially fixed setups and family setups require a larger amount of excess feeder capacity to be effective.
- It does not require any additional equipment or stock, as is required to use feeder bank exchange.
- It works well for very small lot sizes, even single panels.

Disadvantages include:

- It requires machine downtime to change feeders. The user can prepare feeders for the next product, but cannot set up offline and use feeder bank exchange or the Fuji CP split-bank mode.
- It has limited effectiveness if the schedule cannot be set by the setup optimization tool. Estimates of the lost effectiveness vary from 10% to 60%.
- Recipe generation must be dynamic, that is, recipes must be generated after identifying the printed circuit assemblies to be built in a particular time period.
- Late schedule changes may reduce the tool's effectiveness, or require that it be rerun.

SOPT

A software tool to optimize schedules by setup, called SOPT (for *s*etup *opt*imization), was created by HP Laboratories and later adapted to an HP proprietary recipe generation system. This system generates programs, or recipes, for pick-and-place machines. Any references to SOPT in this paper refer to this system.

The SOPT tool works on a list of printed circuit assemblies. It considers only those parts assigned to a particular machine for each printed circuit assembly. First, the printed circuit assemblies are ordered using a traveling salesman heuristic with the number of common parts between printed circuit assemblies as the cost function. Next, a setup is generated for each printed circuit assembly, and the number of feeder changes is calculated. Finally, the schedule is perturbed, and the feeder changes are recalculated to determine if a slightly different schedule would be an improvement. When no further improvements can be found, the program stops. The user can alter the order (for example, if one printed circuit assembly must be built first). The program will then recalculate the setups.

Outputs of SOPT include setup files for the printed circuit assemblies, and an operator instruction file giving the ordered list of printed circuit assemblies and the feeder changes required between each pair of printed circuit assemblies.

Man-Link

Man-Link is a newer HP-proprietary tool that generates recipes for pick-and-place machines. The assembly process is modeled as an ordered list of steps. In each step, a machine or person installs parts on one side of the printed circuit assembly. To generate recipes for all of the steps in a process, Man-Link executes the following sequence of tasks:

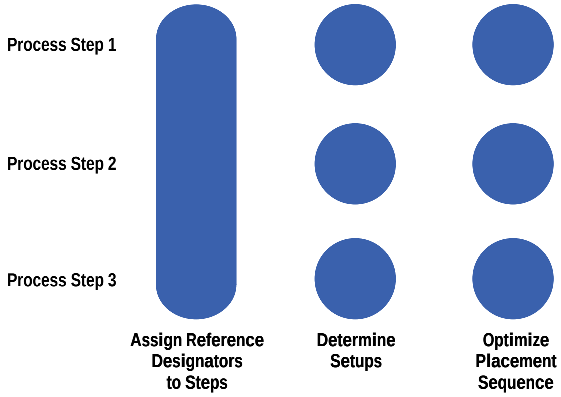
- Assign Reference Designators. Responsibility for the placement of each reference designator is assigned to a step in the process. In making assignments, Man-Link considers other setups, user-input responsibilities, user step preferences, the number of parts each machine can handle, balance among steps, and other factors. This task must be done once for the entire process, to ensure that each placement is assigned to exactly one step.
- Generate Setups. For each step, Man-Link determines where the parts assigned to the step should be loaded on the pick-and-place machine, considering other setups, machine constraints, placement speed, and other factors. This task is normally done separately for each step.
- Optimize Sequence. For each step, Man-Link determines the order in which parts should be placed. This task is always done separately for each step.

Fig. 2 shows these tasks. In Figs. 2, 3, and 4, each shape identifies a part of the problem space that is solved as a unit, either by a single program invocation or by a series of program invocations sharing data. In Fig. 2, assignment of reference designators to steps is done by a single program invocation to ensure that each reference designator is assigned to exactly one step. The other tasks are done by individual program invocations for each step because there is no need to share information among steps.

These tasks are performed in two phases. The first, called *strategic recipe generation*, performs an initial subsequence of these tasks and stores its output in a database. The second phase, called *tactical recipe generation*, performs the rest of the tasks, producing the final recipes that are used to build the printed circuit assembly.

It did not seem prudent to integrate the SOPT tool into Man-Link for a number of reasons. These included the SOPT implementation language (Pascal) and the data structures for setups (SOPT uses files, while Man-Link uses database tables). We decided to create a Man-Link solution using SOPT ideas and algorithms. In the following sections, some of the design issues are discussed.

Fig. 2. Man-Link recipe generation model—single product, multiple steps.


Process Step 1
Process Step 2
Process Step 3
Assign Reference Designators to Steps
Determine Setups
Optimize Placement Sequence

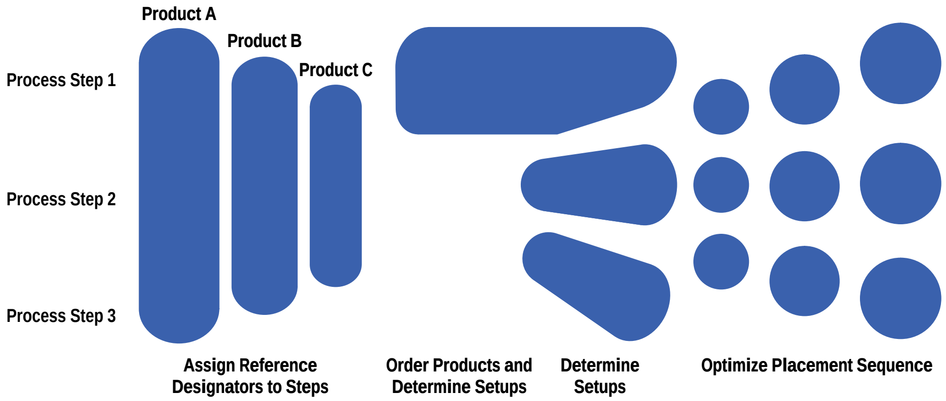
Build Lists

To handle multiple printed circuit assemblies in Man-Link, we introduced *build lists*, which are lists of printed circuit assemblies. We modified the programs to generate recipes for all of the printed circuit assemblies in a build list. In the process, Man-Link must order the build list and create setups with minimal feeder changes.

Ordering and Responsibility

An early issue in the design was how to integrate build list ordering into the list of tasks given above. The SOPT implementation has a single program that orders the list and generates setups for one step (Fig. 3). For each of the other steps, all of the setups are determined together but the list order determined for the first step cannot be altered. The advantage of this approach is that the ordering algorithm has detailed knowledge of what parts will or will not fit on a machine, and can therefore calculate (and optimize) the exact number of feeder changes required. The disadvantages are that the ordering program must include all of the setup code for the target machine (and therefore be machine-specific), and that the ordering considers only one step. An order that is very good for one step in the process may be very bad for another.

Fig. 3. SOPT recipe generation model.


Product A
Product B
Product C
Process Step 1
Process Step 2
Process Step 3
Assign Reference Designators to Steps
Order Products and Determine Setups
Determine Setups
Optimize Placement Sequence

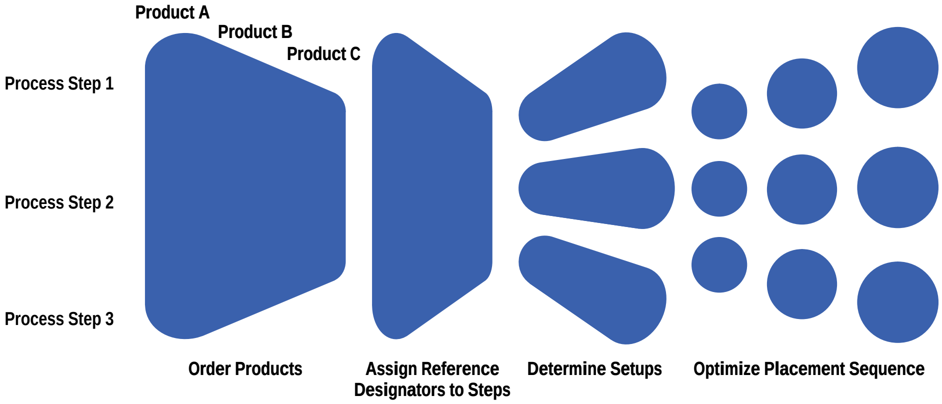
For the Man-Link implementation, we wanted to keep our code modular and to consider all steps when ordering. For these reasons, we decided to separate ordering and setup generation.

As a separate task, build list ordering must precede the generation of setups, since the setups are dependent on the printed circuit assembly sequence. The next question is whether ordering should precede or follow the assignment of reference designators. If ordering is done first, then assignment can use that information. On the other hand, if assignment is done first, ordering can focus accurately on the parts assigned to selected machines.

Perhaps the most compelling consideration was that our users wanted to be able to alter the order after the ordering program had run, but before the setups had been generated. To do this within the Man-Link architecture, we had only two options:

- Make ordering the last task in strategic recipe generation. This means that setup generation must be tactical, and therefore, tactical recipe generation would always have to be run for the entire build list.
- Make ordering a separate phase preceding strategic recipe generation. The strategic phase could then include setup generation, and the tactical phase could be run for a single printed circuit assembly on a single step.

Fig. 4. Man-Link recipe generation model—multiple products with setup optimization.

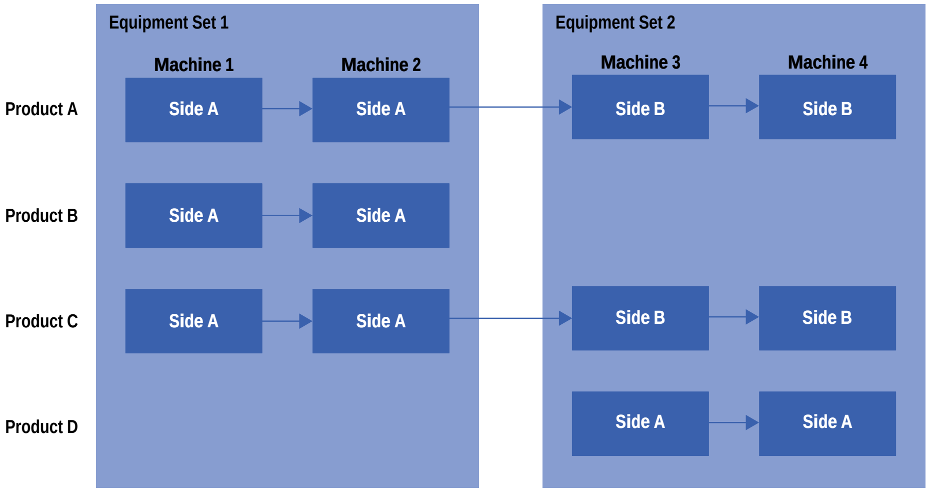


After discussing these issues with our users, we decided to implement the second alternative. Fig. 4 shows that ordering is done for all products, independent of steps, before any other task. Assignment of reference designators to steps is then done by a series of program invocations sharing data and using ordering information.

Equipment Sets

Since build list ordering precedes reference designator assignment in Man-Link, the ordering cannot consider the set of parts assigned to a particular step. We therefore thought we could just order printed circuit assemblies based on all of the parts on the printed circuit assemblies. The problem with this approach is that the parts on the two sides of a printed circuit assembly are often quite different. This could lead to poor results for processes consisting of one set of machines for top-side placement and a second set of machines for bottom-side placement. For this reason, we decided to order top-side parts separately from bottom-side parts. To do this requires the concept of an *equipment set*, which is defined as the set of machines used to place all of the parts on one side of a printed circuit assembly. For example, in Fig. 5, machines 1 and 2 form one equipment set, and 3 and 4 form a second equipment set. The ordering program will consider separately the parts assigned to each of the two equipment sets.

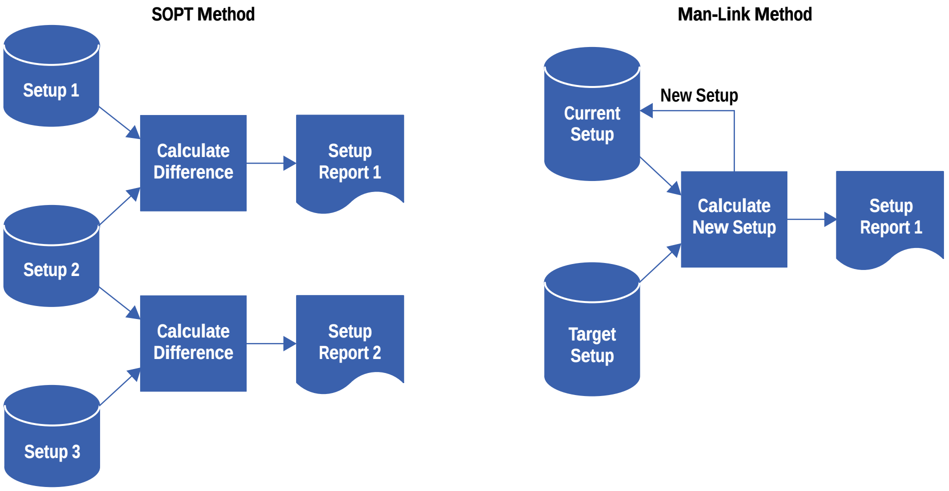
Fig. 5. An equipment set is the set of machines used to place all of the parts on one side of a printed circuit assembly.



Schedule Dependence

In the SOPT implementation, any change in schedule normally requires that the SOPT program be rerun. We hoped to alleviate this in Man-Link by separating setup generation from operator instruction generation. The idea is to keep track of the parts loaded on a machine. When the operator selects the next printed circuit assembly to be run, Man-Link runs a machine dependent program that compares the current setup with the setup for the new printed circuit assembly (Fig. 6). This program lists the minimum number of changes required to get the required parts for the new printed circuit assembly in the right slots. If the operator follows the determined schedule, changes are minimized. If the operator deviates from the schedule, there may be more changes, but the setups do not have to be regenerated. This should be particularly effective if the only change is to insert a prototype into the schedule.

Fig. 6. *Calculating operator instructions.*



The difficulty with this approach is that Man-Link must always know the current setup. We have a means in Man-Link to ensure this, but it is not robust and has been a source of frustration to our users.

Steps and Machines

A Man-Link process is made up of a sequence of steps, each of which refers to a machine. A particular machine may be used for more than one step in a process. In this case, Man-Link could do one of two things. First, Man-Link could treat each step independently, creating a separate sequence of setups for each step. This assumes that the user runs all printed circuit assemblies through a given step before going on to the next step, which is not realistic. Alternatively, Man-Link could generate all setups for a given printed circuit assembly before going on to the next printed circuit assembly, and create a sequence of setups for each machine. We chose to implement the second approach.

One result of this decision is that the setups for each printed circuit assembly must be sequential in the setup list. For example, in a two-sided surface mount process, the setup for each printed circuit assembly's bottom side will immediately precede the setup for its top side, assuming the same machine is used for both sides. (See the next section for a way around this constraint.)

This is a change from the SOPT implementation, which treats different sides of a printed circuit assembly as separate printed circuit assemblies. This turns out to be both good and bad—good because the sides can be ordered with other printed circuit assemblies to require fewer feeder changes, but bad because SOPT may put the sides in the wrong order, requiring manual shuffling of the schedule.

One related note—Man-Link can be configured to create a single setup for a pair of steps using the same machine. For example, if a two-sided process uses the same CP3 machine to place both the top and the bottom sides of a printed circuit assembly, then the user can configure Man-Link to create one CP3 setup containing all the parts required for both sides. In this case, the sequence of setups for the CP3 would have only one setup for the two-sided printed circuit assembly.

Separating Sides

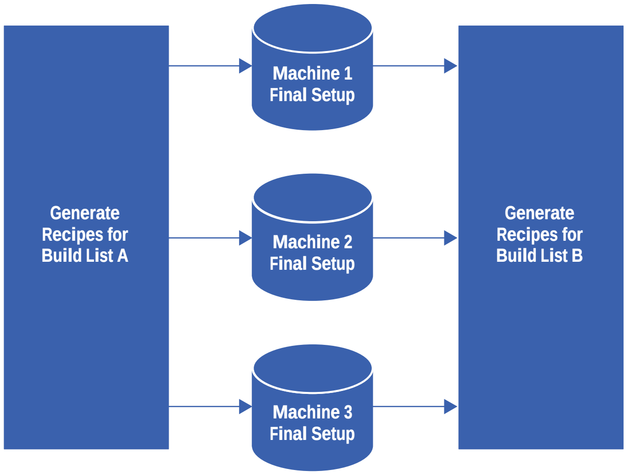
For certain applications, it may be advantageous to treat the different sides of a printed circuit assembly as separate printed circuit assemblies. This would allow the top and bottom sides of the printed circuit assembly to be separated in the build list, or even to be in different build lists. This feature would be useful for processes that use different machines to place the top and bottom sides.

Man-Link does not support this capability directly, but does provide a way to achieve it. The user must define a dummy machine that does not cause setups or recipes to be generated. To satisfy the reference designator assignment module, steps using this machine can take responsibility for parts. The user must then define a top-side process with a dummy step for the bottom side, and a bottom-side process with a dummy step for the top side. To generate recipes for a double-sided printed circuit assembly, the user must run recipe generation using both processes. The two printed circuit assembly/process pairs can be included at different points in a build list, or even in different build lists.

Starting Setups

To get the most out of optimization by schedule, it is best to take advantage of any parts left on the machines from previous builds. In other words, the last setups for yesterday's build list should be an input for recipe generation for today's build list (Fig. 7). In generating the setups for today's first printed circuit assembly, we can use any parts left over from yesterday's last printed circuit assembly setups.

Fig. 7. *Using starting setups.*


Generate Recipes for Build List A
Machine 1 Final Setup
Machine 2 Final Setup
Machine 3 Final Setup
Generate Recipes for Build List B

Man-Link provides this capability by saving the last setup generated for each machine. This works well as long as setups are generated by only one person at a time. If two people try to generate setups for the same machines at the same time, they will overwrite each other's starting setups, causing all manner of confusion. Man-Link therefore has the constraint that strategic recipe generation for build lists may not be done for the same machines by more than one user simultaneously.

A suggested way around this constraint is to have a directory for starting setups for each user. This would keep multiple users from interfering with each other, but does not address the problem of determining the correct starting setup for each machine, that is, how each machine will be set up before the first printed circuit assembly in the build list is started.

Estimating Benefits

To assist users in projecting the benefits of optimization by schedule, we constructed this model of the setup process:

- Let $N =$ the number of printed circuit assemblies to be built in a schedule.
- Let $C =$ the average number of parts in the printed circuit assemblies.
- Let $P =$ the probability that any given part used by a particular printed circuit assembly is also used by a second given printed circuit assembly. In other words, P is the average fraction of parts shared by any two printed circuit assemblies in the schedule. For typical build lists, this varies from around 0.13 to 0.37, with an average of about 0.25.
- Let $F =$ the number of parts the machines can hold. For a CP3, this is around 112, assuming that three times as many one-slot parts are used as two-slot parts.
- Let $U =$ the total number of unique parts for all of the printed circuit assemblies in the schedule.
- Let $X =$ the total number of feeder changes required by the setups.

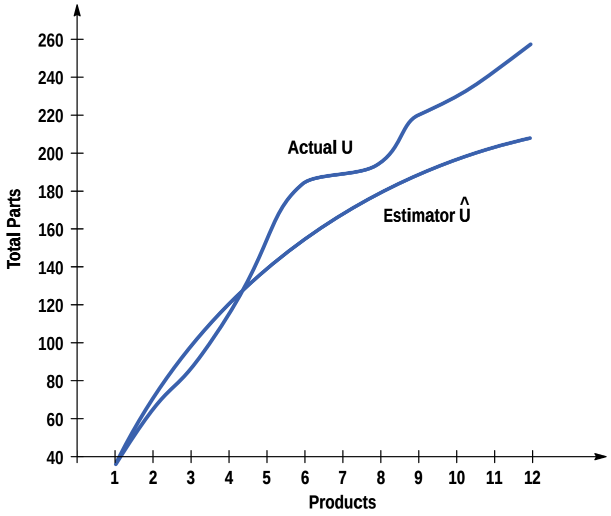
The average number of parts that are common to two printed circuit assemblies is CP. The total number of parts in two printed circuit assemblies is therefore $C + (C - CP)$, or $2C - CP$. Assuming that commonality is uniformly distributed, the average number of parts common to three printed circuit assemblies is CP^2, so the total number of parts in three printed circuit assemblies is $C + (C - CP) + (C - 2CP + CP^2)$. Continuing this reasoning, we arrive at the following estimator of U:

$$\hat{U} = \frac{C}{P}\left(1 - (1 - P)^N\right). \qquad (1)$$

This should immediately raise some alarms, because it predicts that the total number of unique parts converges to C/P as the number of printed circuit assemblies (N) gets large. In fact, this turns out to be a reasonable first-order estimator if the number of printed circuit assemblies is not too big. If N is between 2 and 5, it is pretty good, but as N approaches 10, it is consistently low. Fig. 8 shows estimator values and actual values based on a random collection of products from one surface mount center.

The number of feeder changes required, X, depends on F. If $F < C$, the parts for a printed circuit assembly will not fit on the machines, so C must be a lower bound for F. If $F \geq U$, all of the parts may be mounted at once, so the number of changes depends on the commonality with the starting setups. If the starting setups do not contain any parts in the printed circuit assemblies, then $X = U$. If the starting setups have all of the parts, $X = 0$. In the real world, X is generally between these two extremes, and in fact, $\hat{U}$ turns out to be a reasonable estimator. In an analysis of 25 build lists used at surface mount centers, $\hat{U}$ predicted a 61% decrease in feeder changes, compared to an actual decrease of 65%.

Fig. 8. *Accuracy of the U estimator based on data from one surface mount center.*



Ordering the build list for maximum commonality can raise the effective value of P from an average of 0.26 to an average of 0.28 in the 25 build lists mentioned above.

Setup vs. Cycle Time

Most strategies that reduce setup time do so at the expense of cycle time. In other words, you can save time setting up the machine if you are willing to live with setups that do not build printed circuit assemblies as quickly. There are several reasons why this occurs:

- A setup that can be used for multiple printed circuit assemblies is less compact. It requires greater machine movement (either of the part feeder carriage or of the placement heads) to access the parts for each printed circuit assembly.
- On the Fuji CP machines, a movement of one slot on the part feeder carriage between placements does not slow the machine down. The Man-Link sequence generators take advantage of this by optimizing placement sequences for pairs of adjacent parts. Careful selection of parts to put together in the setup can therefore have a big impact on the cycle time.
- The Fuji CP3 must place all "tall" parts after all "short" parts. An optimized setup will have all of the tall parts at one end of the part feeder carriage, so each printed circuit assembly can be built with a single pass through the carriage.
- Fuji IP2 optimization is compromised. When Man-Link generates a setup for a single printed circuit assembly, it will balance the loads on the two heads. The user can request that a high-use part be loaded on both banks to further improve the balance. Also, high-use parts are assigned to slots with the lowest access times.

The combined impact of these effects has been measured to be an increase of cycle time of between 5% and 10%. For small lot sizes, this is a good trade-off, but for large lot sizes (greater than 100 panels) overall throughput will suffer.

Parallel versus Serial Lines

As noted above, the benefit derivable from this setup strategy is dependent on the amount of excess feeder capacity available. The immediate temptation is to put machines in series to increase the effective feeder capacity. This could be a mistake, however, because the utilization of machines in series decreases as a result of the increased overhead of conveying the panels and reading marks more than once. This cost can be modeled as follows:

- Let V be the overhead per panel. This is the sum of the conveying time, the time to read fiducials, and the time to read image-reject marks.
- Let S be the average number of seconds required to place a single component. For the CP2 and CP3 machines, a figure of 0.3 second works well.
- Let M be the number of placements to be done by the CP machines.

For one CP machine, the time required to build a panel is
$V + MS$ seconds. If two CPs in series build the panel, each placing half of the parts, the total time is:

$$\text{Total machine time} = 2 \times \left(V + \frac{M}{2} \times S\right) = 2V + MS. \qquad (2)$$

The increase in cycle time caused by having two machines in series rather than in parallel is:

$$\text{Change in total machine time} = \frac{2V + MS}{V + MS}. \tag{3}$$

Assume that the overhead is 17 seconds. This is enough time for conveying the panel and reading four panel fiducials and two image-reject marks. For panels with 100 placements, the cycle time is increased by 36% by the serial configuration. Again, for some shops this will be a reasonable trade-off.

The Bottom Line

To estimate the potential value of this optimization method, the user must:

1. Estimate the reduction in setup time. If the surface mount center is currently performing a complete teardown and setup for each printed circuit assembly, the number of feeder changes should be reduced from $N \times C$ to approximately $\hat{U}$ from equation 1. If it takes T seconds to change one feeder, the savings is $(NC - \hat{U})T$.

For example, for five products, with an average of 50 parts per product, and part commonality of 0.20, the expected number of feeder changes $\hat{U}$ will be 168. This means that the reduction in feeder changes will be $250 - 168 = 82$. If each feeder change takes one minute, the total reduction in feeder changes will be 82 minutes, or about 16 minutes per product.

2. Estimate the increase in cycle time. This will be about 10% of the total run time. If machines that are currently running in parallel need to be put in series to get sufficient feeder space, the user should also estimate the resulting increase in total machine time, using equation 3 above.

In the above example, if the average number of placements per product is 400, and the average placement time is 0.3 second, the increase in cycle time will be about $400 \times 0.3\ \text{second} \times 10\% = 12\ \text{seconds} = 0.2\ \text{minute/panel}$.

3. The break-even lot size can then be calculated by setting the decrease in setup time equal to the increase in cycle time. In the above example:

$$\frac{16\ \text{minutes/product}}{0.2\ \text{minute/panel}} = 80\ \text{panels/product}.$$

4. Alternatively, the user can calculate the time savings per lot. In the above example, assuming 10 panels per lot:

- Without optimization: 50 minutes setup + 20 minutes build = 70 minutes
- With optimization: 34 minutes setup + 22 minutes build = 56 minutes
- Savings per lot = 14 minutes.

This represents a 20% decrease in the time to build a lot.

We expect an improvement for lines that are dominated by setup time, such as lines for prototypes or high-mix, low-volume products. If the lot size is consistently less than 100 panels, optimization by schedule may be a good fit.

Conclusion

Life is a series of trade-offs. We believe that the choices we made in this work will provide the best benefit to our customers. We have received encouraging statistics from the surface mount centers that are using the product. The estimation methods given above should allow other surface mount centers to evaluate this strategy for their shops.

Acknowledgments

Many thanks to Shailendra Jain for dreaming this up, creating a prototype, and providing crucial input on our efforts. Thanks to Fereydoon Safai and Ed Katz for their work in the design and implementation of both the SOPT and Man-Link solutions. And thanks to our customers, who funded the project, and to Eiko Johnson, who managed it.

Low-Temperature Solders

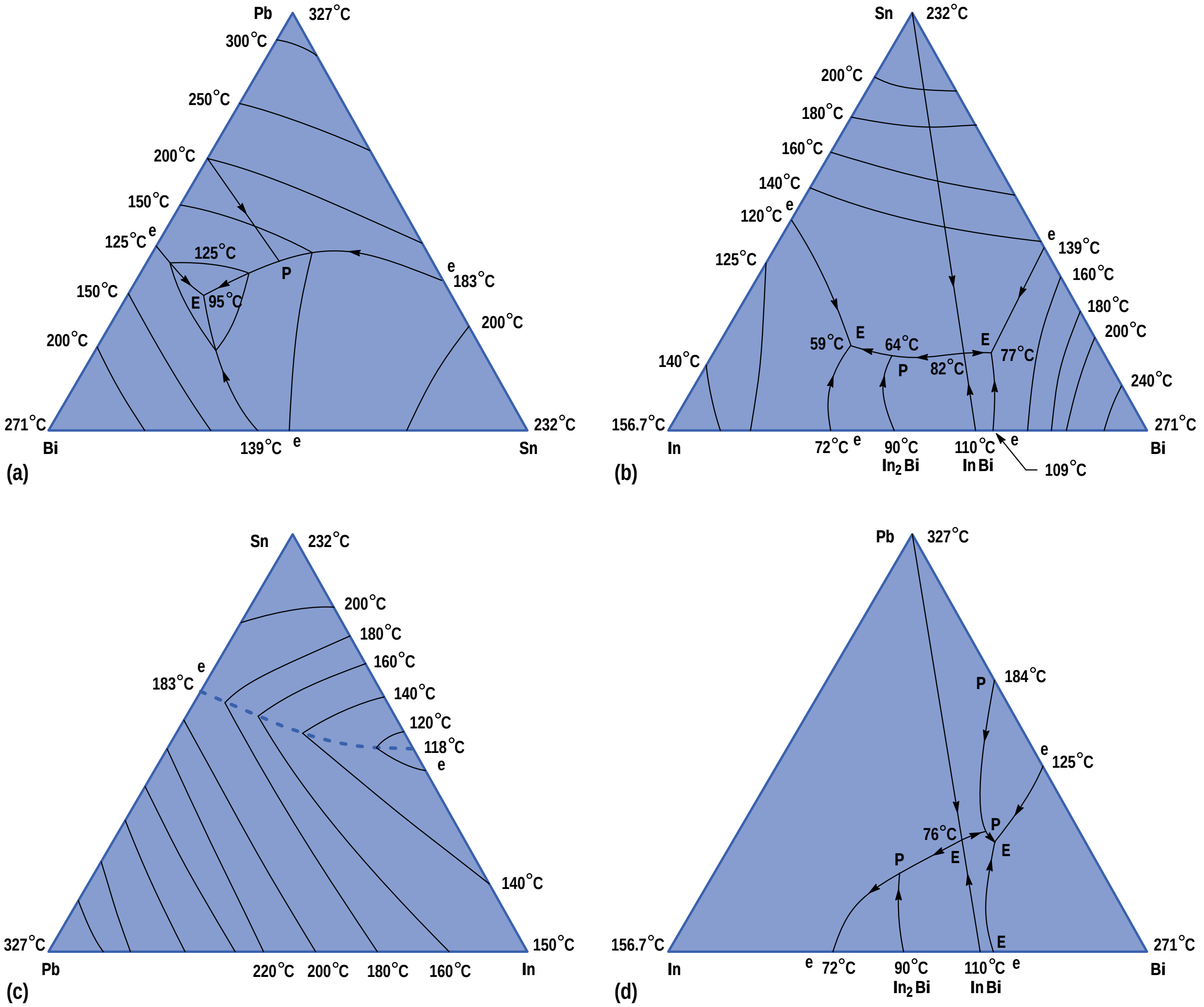
The application of low-temperature solders in surface mount assembly processes for products that do not experience harsh temperature environments is technically feasible. One single alloy may not be appropriate as a universal solution.

by Zequn Mei, Helen A. Holder, and Hubert A. Vander Plas

Low-temperature soldering has been a subject of research at HP's Electronic Assembly Development Center (EADC). Several benefits may come from developing this technology, including thermal shock reduction, step soldering capability, and possibly, lead (Pb) elimination.

Thermal Shock Reduction. The risk of thermally induced damages will be reduced if the peak exposure temperature is reduced. A significant decrease in the peak reflow temperature (the oven temperature at which the solder melts and makes the connections between the components and the board) will reduce damage to components. Currently, peak reflow temperatures are around 210°C to 230°C. These temperatures are sufficient to cause phenomena such as *popcorning*, a fairly well-known phenomenon in which air and moisture that have been trapped in the plastic package of an IC are heated to the point where they expand and cause the component case to crack open.

Fig. 1. *Melting points of ternary systems of all possible combinations of (a) BiPbSn, (b) BiInSn, (c) PbInSn, and (d) BiInPb.*



The damage from popcorning is immediate and usually detectable, but there are other thermally induced damages that can cause long-term problems, such as warping of printed circuit boards or damage to ICs, which would also be reduced with lower peak temperatures.

Step Soldering. The availability of solders with lower melting points will make multiple reflow processes on a single board possible. For example, all of the normal components that can tolerate higher reflow temperatures could be soldered to a board using the standard process, and then the lower-temperature components could be added in another reflow process. Since step soldering is a bulk reflow process, it takes less time and is more uniform than hand soldering, and doesn't take any different equipment or special training.

Possible Pb Elimination. Many low-temperature solders contain no lead.

Selection of Low Melting Alloys

We call a solder alloy *low melting* if it melts at temperatures below 183°C and above 50°C. Most of the alloys that meet this requirement are made of four elements: Sn (tin), Pb (lead), Bi (bismuth), and In (indium). The Cd (cadmium) bearing alloys are not considered because of their extreme toxicity. Various compositions of these elements produce alloys that melt at any given temperature between 50°C and 183°C. Commercially available low-melting alloys are listed in Table I. The numbers associated with each alloy in Table I are the percentages by weight of the components that make up the alloy.

To better understand the correlation between the alloy compositions and their melting temperatures, we can use the ternary diagram of melting temperature. A ternary diagram uses a triangle to represent chemical compositions of a three-element alloy system. A physical property, such as melting temperature, is plotted over the triangle. Figs. 1a to 1d show the melting points of ternary systems of all possible combinations of the elements BiPbSn, BiInSn, InPbSn, and BiInPb.

These diagrams show what are called the liquidus temperatures, as opposed to the solidus temperatures. A typical alloy melts not at a single temperature but over a temperature range. The solidus temperature is the highest temperature at which an alloy remains solid, while the liquidus temperature is the lowest temperature at which an alloy remains liquid. At the temperatures between the solidus and liquidus temperatures, an alloy is a mixture of solid and liquid. The solidus temperatures of these alloy systems are not shown in Fig. 1. However, for a few specific compositions labeled "e" or "E" in Fig. 1, the so-called eutectic alloys, the solidus and liquidus temperatures are equal. Alloys with eutectic compositions or small differences between their liquidus and solidus temperatures are often favored for soldering applications because they melt and solidify rapidly instead of over a range of temperatures.

Not all the compositions found on the ternary phase diagram are suitable for soldering applications. To determine which are most appropriate, see Table 1.

- Wettability. A metal is said to have *wetted with a surface* if it forms a sound metallurgical bonding with the surface. Wetting is essential in the soldering process because it ensures that the joint created won't come apart at the interface. Any new alloy must be able to wet to the common pad surface finishes: Cu, PbSn, and Ni coated with Pd or Au.
- Reliability. Lower-temperature alloys should still be reliable, so we measure the following properties to estimate how reliable solder joints made of an alloy will be: shear strength, creep resistance, isothermal fatigue resistance, and thermal fatigue resistance.
- Long-term stability. Microstructural evolution, grain growth, and recrystallization contribute to changes in the solder joint mechanical properties over time, so we want to make sure that the changes are slow and stable and won't reduce the mechanical properties of the solder joints to unacceptable levels over the life of the joint.
- Practicality. Alloys used for mass production should be cheap and widely available. It should be possible to make them into solder pastes so that they can be used in standard assembly processes, and suitable fluxes should be available. The alloys shouldn't be more toxic than what's currently used.

To begin our alloy selection and evaluation, we found references in the available literature to low-temperature alloys that might fit these requirements. Three alloys were selected for further evaluation:

- 43Sn43Pb14Bi. The solidus temperature of this alloy is 144°C and the liquidus temperature is 163°C, 20°C lower than 63Sn37Pb, but with similar mechanical properties.
- 58Bi42Sn. This composition is a eutectic alloy that melts at 139°C. It is lead-free and strong, but brittle. Also, its fatigue resistance is questionable.[1,2]
- 40Sn40In20Pb. The solidus temperature of this alloy is 121°C and the liquidus temperature is 130°C. It is soft and ductile. It doesn't have the problem of embrittlement when soldering to thick gold surfaces, like PbSn, because of the high In content. Unfortunately, the high In content drives the price of this alloy up because In is extremely expensive right now.

Table I
Low-Melting Alloys

Chemical Composition	Liquidus Temperature (°C)	Solidus Temperature (°C)	Chemical Composition	Liquidus Temperature (°C)	Solidus Temperature (°C)
49Bi21In18Pb12Sn	58	58	34Pb34Sn32Bi	133	96
51In32.5Bi16.5Sn	60	60	56.84Bi41.16Sn2Pb	133	128
49Bi18Pb18In15Sn	69	58	38.41Bi30.77Pb30.77Sn0.05Ag	135	96
66.3In33.7Bi	72	72	57.42Bi41.58Sn1Pb	135	135
57Bi26In17Sn	79	79	36Bi32Pb31Sn1Ag	136	95
54.02Bi29.68In16.3Sn	81	81	55.1Bi39.9Sn5Pb	136	121
51.45Bi31.35Pb15.2Sn2In	93	87	36.5Bi31.75Pb31.75Sn	137	95
52Bi31.7Pb15.3Sn1In	94	90	43Pb28.5Bi28.5Sn	137	96
52.5Bi32Pb15.5Sn	95	95	58Bi42Sn	138	138
52Bi32Pb16Sn	95.5	95	38.4Pb30.8Bi30.8Sn	139	96
52Bi30Pb18Sn	96	96	33.33Bi33.34Pb33.33Sn	143	96
50Bi31Pb19Sn	99	93	97In3Ag	143	143
50Bi28Pb22Sn	100	100	58Sn42In	145	118
46Bi34Sn20Pb	100	100	80In15Pb5Ag	149	142
50Bi25Pb25Sn	115	95	99.3In0.7Ga	150	150
56Bi22Pb22Sn	104	95	95In5Bi	150	125
50Bi30Pb20Sn	104	95	42Pb37Sn21Bi	152	120
52.2Bi37.8Pb10Sn	105	98	99.4In0.6Ga	152	152
45Bi35Pb20Sn	107	96	99.6In0.4Ga	153	153
46Bi34Pb20Sn	108	95	99.5In0.5Ga	154	154
54.5Bi39.5Pb6Sn	108	108	100In	156.7	156.7
67Bi33In	109	109	54.55Pb45.45Bi	160	122
51.6Bi41.4Pb7Sn	112	98	70Sn18Pb12In	162	162
52.98Bi42.49Pb4.53Sn	117	103	48Sn36Pb16Bi	162	140
52In48Sn	118	118	43Pb43Sn14Bi	163	144
53.75Bi43.1Pb3.15Sn	119	108	50Sn40Pb10Bi	167	120
55Bi44Pb1Sn	120	117	51.5Pb27Sn21.5Bi	170	131
55Bi44Pb1In	121	120	60Sn40Bi	170	138
55.5Bi44.5Pb	124	124	50Pb27Sn20Bi	173	130
50In50Sn	125	118	70In30Pb	175	165
58Bi42Pb	126	124	47.47Pb39.93Sn12.6Bi	176	146
38Pb37Bi25Sn	127	93	62.5Sn36.1Pb1.4Ag	179	179
51.6Bi37.4Sn6In5Pb	129	95	60Sn25.5Bi14.5Pb	180	96
40In40Sn20Pb	130	121	37.5Pb37.5Sn25In	181	134
52Sn48In	131	118			

These three were chosen mostly because there was more information available on them than on other low temperature alloys, not necessarily because we thought they would make the best solders. They provided a starting point.

Because the technical data on the low temperature alloys was limited and inconclusive,[3] we conducted a series of tests based on our selection criteria listed above.

Wetting and Solderability

Two types of tests were conducted to look at the wetting performance of these alloys: spreading tests and wetting balance tests.

In spread tests, a dollop of solder paste is deposited on a copper board or test coupon. The coupons are then heated to 30°C above the liquidus temperature of the alloy in an oven under a nitrogen atmosphere. The dollop of solder paste melts, and as long as the flux is active enough to remove the surface metal oxides, the solder forms a bead, or cap (see Fig. 2). The diameter and height of the solder cap can then be measured to determine the contact angle (α) of the solder to the board. This contact angle, or wetting angle, is a measure of how well the solder will wet in a surface mount process—smaller is better.

Fig. 2. *Solder bead formed by reflowing paste on a plain Cu surface. α is the wetting angle.*

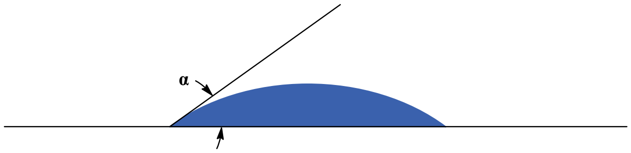
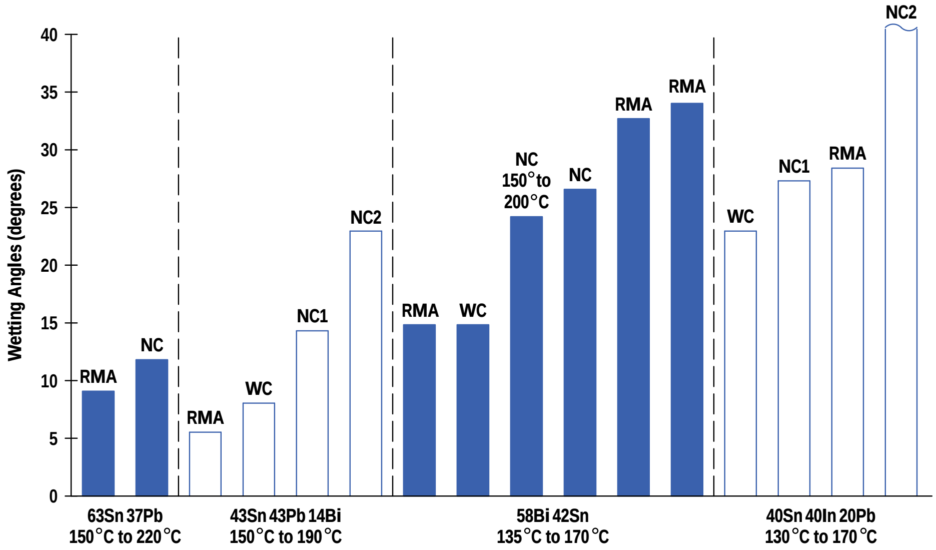


Fig. 3. *Wetting angles determined from spreading tests of solder pastes on copper, reflowed in a nitrogen oven. The x axis indicates the solder alloys and reflow temperatures. The fluxes are indicated at the tops of the bars (WC = water-clean, NC = no-clean, RMA = rosin mildly activated).*



Factors that affect the spread test include the activity of the flux, the surface tension of the molten alloy, and the alloy's ability to make a metallurgical bond with the surface metallization. All of these factors have to be taken into account when interpreting the results of spread tests.

The results of the wetting angle tests are shown in Fig. 3. The 63Sn37Pb and 43Sn43Pb14Bi alloys both wetted well and similarly with the same flux. The 58Bi42Sn and 40Sn40In20Pb alloys generally wetted the copper surface ($\alpha < 90°$), but not as well as the other two alloys, averaging two to three times the wetting angle with the same fluxes. In fact, the 40Sn40In20Pb alloy didn't wet at all with one no-clean flux (NC2). These differences may have to do with the fact that indium and bismuth oxides are more difficult to remove than tin and lead oxides. These alloys also have lower surface tensions than Pb Sn.

Another factor in how the lower-temperature alloys performed is that the current water clean and no-clean fluxes were developed for 63Sn37Pb and activate at about 150°C. They may not be suitable for the low-temperature solders since most of the low-temperature solders melt at temperatures below 150°C. Wetting balance tests were conducted to find fluxes that would be appropriate for use at lower temperatures, and the results of those tests are presented in reference 4 and in ***Article 11***.

Reliability and Long-Term Stability

Before we could suggest that anyone change from PbSn solder to an alternative alloy, we needed to understand the mechanical properties of the alloy well enough to know what the trade-offs would be. Therefore, the bulk of the tests we did to evaluate the alloys focused on the areas of shear, creep, isothermal fatigue, and thermal fatigue.

Shear. Solder joints experience shear because of coefficient of thermal expansion mismatches. To look at the behavior of solder joints of different alloys in shear, we used specimens as shown in Fig. 4. These specimens have nine solder joints of dimensions 0.050 by 0.080 by 0.010 inch sandwiched between two copper plates. When the ends are pulled in a testing machine at different temperatures and strain rates, the stress in the solder joints can be measured. Plotting the measured maximum stress against the strain rates gives us the relative shear strength of the different alloys and allows us to compare them to PbSn.

Our shear tests were conducted at three temperatures (25°C, 65°C, and 110°C) and at three strain rates (10^{-2}, 10^{-3}, and 10^{-4} per second). The results of the shear strength tests for the low-temperature solders and several high-temperature solders are plotted in Fig. 5.

From these plots we can see that at 25°C, under the same strain rates, 58Bi42Sn is the second strongest, inferior only to a high-temperature Pb-free alloy. 43Sn43Pb14Bi had about the same strength as 63Sn37Pb, while 40Sn40In20Pb is the softest. As the temperature increased to 110°C, the low-temperature solders became much softer while the high-temperature solders were still relatively strong.

Creep. If a constant load is applied to a material while it is held at an elevated temperature, it will deform, or flow, over time. This time dependent deformation is called creep, and is most significant at absolute temperatures greater than about half the melting point of the material. Since creep is the main deformation mechanism in solders, it's important to know how creep resistant a new solder alloy will be.

Fig. 4. *Specimen for shear and creep tests.*

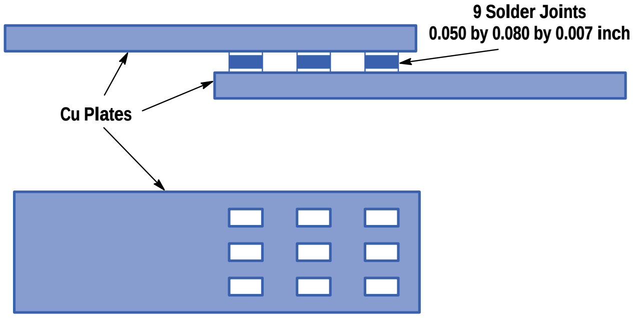


Fig. 5. *Results of shear strength tests for the low-temperature solders and several high-temperature solders at (a) room temperature, (b) 65°C, and (c) 110°C.*

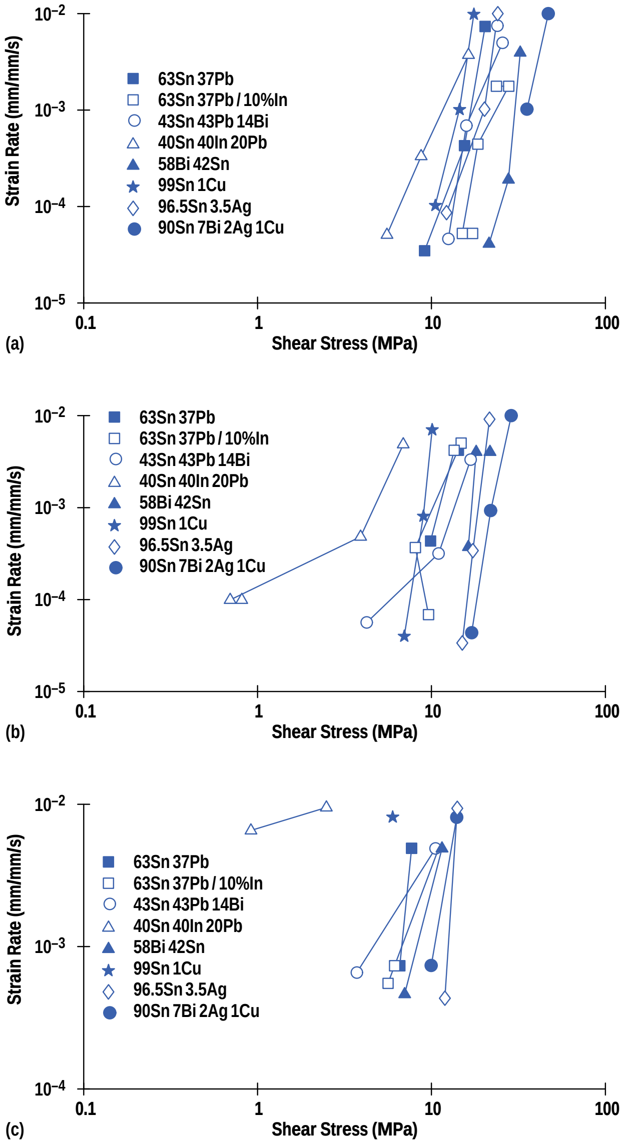
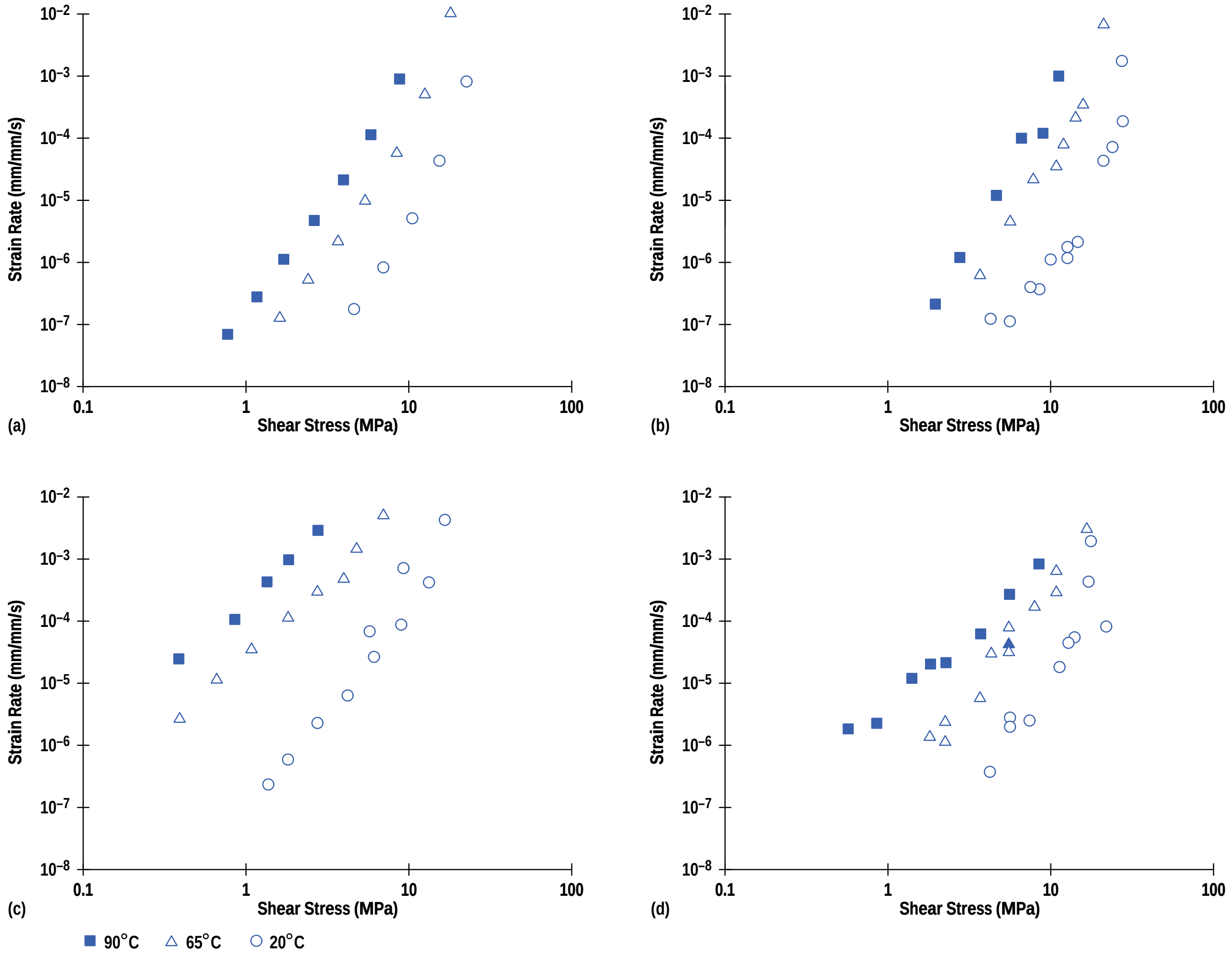


Fig. 6. *Steady-state creep (strain) rates at 20°C, 65°C, and 90°C as a function of shear stress for 63Sn37Pb (a) and the low temperature solders: (b) 58Bi42Sn, (c) 40Sn40In20Pb, and (d) 43Sn43Pb14Bi.*



The same kind of specimens used in shear tests were used in the creep tests. The steady-state strain rates as a function of shear stress at 25°C, 65°C, and 90°C are plotted in Fig. 6. The data has been fitted with standard creep (Dorn) equations:

$$\frac{d\gamma}{dt} = A\tau^{n}e^{-\Delta H/RT},$$

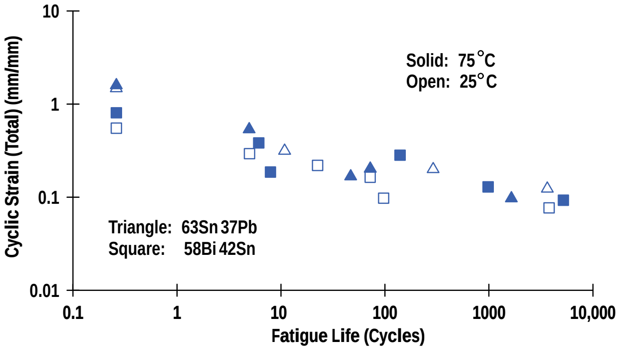
where γ is the shear strain or creep, A is a materials constant, τ is the shear stress, n is an empirical constant typically between 3 and 7, H is the activation energy, R is the gas constant, and T is the absolute temperature in K. The resulting Dorn equation parameters are listed in Table II.

Table II
Creep Equation Parameters for Three Solder Alloys

Alloy	A	n	ΔH (kcal/mole)
40Sn40In20Pb	4.0488×10^{4}	2.98	22.00
58Bi42Sn	5.5403×10^{-7}	4.05	16.85
43Sn43Pb14Bi	0.11552	2.94	17.05

The rupture strains of the low-temperature solders were also determined from the creep tests. 58Bi42Sn showed the slowest creep rate but the least rupture strain for the same stress compared with the other low-temperature solders and the 63Sn37Pb, while 40In40Sn20Pb exhibited the fastest creep rate but the largest rupture strain.

Fig. 7. *Isothermal shear fatigue test results.*



Isothermal Fatigue. When materials are subjected to small repeated loading, they can eventually fracture. This process of gradual fracture is called fatigue. Solder joints experience loading because of coefficient of thermal expansion mismatches. These loads are cyclic, caused by temperature excursions during operation. Isothermal strain cycles can be used to rapidly simulate joint exposure to show relative fatigue lives of different solder alloys. There is a relationship called the Coffin-Manson Law, which is one way of estimating the fatigue life of the material. Fatigue life is defined as the number of cycles at a given strain that will cause failure in the material.

Coffin-Manson relations for the low-temperature solders have been determined at both 25°C and 75°C. The data for 58Bi42Sn and 63Sn37Pb is shown in Fig. 7. The isothermal fatigue life of 58Bi42Sn is shorter than 63Sn37Pb under the same cyclic strains.

Thermal Fatigue. Although isothermal fatigue can be used to estimate fatigue life, we also do actual thermal cycling to show how the joints will perform as the temperature cycles. For our thermal fatigue tests, a new type of test vehicle was designed (see Fig. 8). Five ceramic plates, all 1/16 inch thick, and 4, 2, 1, 1/2, and 1/4 inch square respectively, were soldered onto a 1/8-inch-thick FR-4 board. Eight solder joints 0.010 inch thick and 0.050 inch in diameter, located in a ring, were sandwiched between each ceramic plate and the FR-4 board. Each solder joint was individually tested for electrical continuity while being temperature cycled in a thermal chamber. Two temperature profiles were used, 25°C to 75°C and −20°C to 110°C.

The results of the −20°C-to-110°C test are plotted in Fig. 9. Since the test is still in progress, only the fatigue data for the failed solder joints is plotted. 63Sn37Pb lasted longer than 58Bi42Sn, and approximately the same number of cycles as 43Sn43Pb14Bi. The 40Sn40In20Pb solder joints have the longest fatigue lives.

Practicality

To examine the practical side of using these alloys, we did a prototype build. Since the 40Sn40In20Pb alloy is so expensive, it's an unlikely candidate for large-scale production, so we excluded it from the prototype builds. The 58Bi42Sn alloy is harder to solder than 43Sn43Pb14Bi (it has a lower melting temperature and its oxide is harder to remove), so we chose to test the worse case of the two remaining alloys and build with 58Bi42Sn.

The 58Bi42Sn alloy was made into a solder paste with a water-soluble RMA flux.[5] This kind of flux was used because, unlike most standard no-clean fluxes, it is active at the lower oven temperatures used with BiSn. The assembly we chose for this build had a variety of components, including 0.025-inch-pitch components.

Fig. 8. *Test vehicle for thermal fatigue tests.*

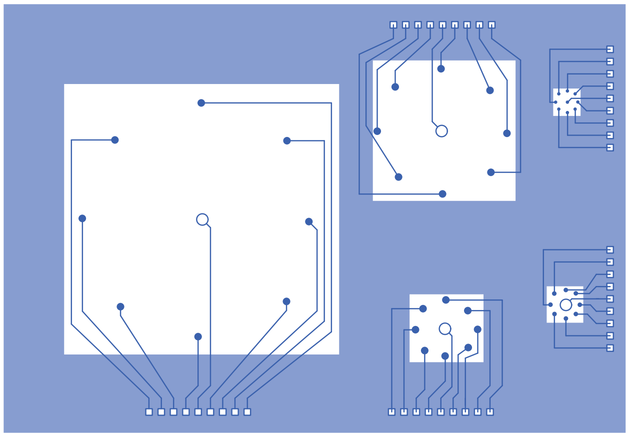
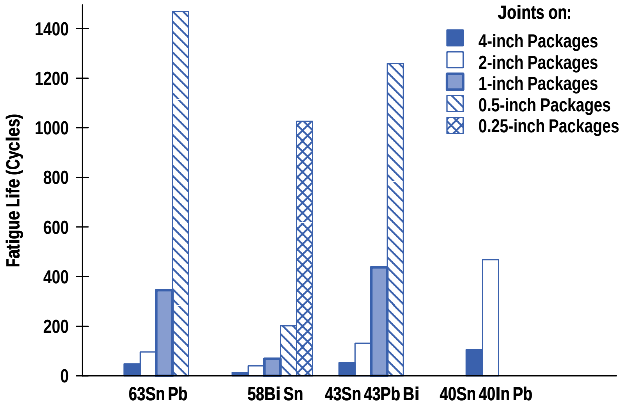


Fig. 9. *Results of the −20° C-to-110° C thermal fatigue test. Fatigue lives are shown only for joints that had failed at the time of writing.*



Two types of board platings were used: organic coated copper (OCC) and hot air solder leveling (HASL). These coatings protect the copper pads from oxidation before the reflow process. For OCC, the copper pads are coated with a thin layer of a polymer that preserves the solderability of the surface by preventing the oxidation of the copper underneath, but burns off during the reflow process to allow for metallurgical bonding between the surface and the solder. HASL or HAL (hot air leveling) accomplishes the same protection but uses a thin layer of PbSn solder that has been blown level with air knives.

The entire assembly process was the same as for 63Sn37Pb, except that a different reflow profile was used. The low-temperature profile had a preheat period of 4 minutes at 130°C and a peak period of 1.5 minutes at temperatures between 138°C and 175°C (0 to 39°C above the melting point of the alloy).

Twenty boards were built with no defects. The boards passed functional tests as well as out-of-plane random frequency vibration (45 minutes at 6g) and board environmental stress testing (BEST—thermal cycling from −45°C to 100°C, 1 hr/cycle, functionality monitored throughout).

Failure of 58Bi42Sn on Pb-Containing Surface

During the thermal cycling of the prototype boards, we observed a thermal fatigue failure mechanism of the BiSn solder on Pb-containing surfaces.[6] Some components on the prototype boards fell off after about 500 cycles of BEST. Boards soldered with 63Sn37Pb failed after about 900 cycles.

Fig. 10 shows top views of the 58Bi42Sn solder joints before and after BEST. Before BEST, the solder joint surfaces were smooth. After BEST, the solder joints between OCC boards and the components with Ni-Pd coating remained smooth, but the solder joints between either the HAL boards or the components with PbSn coating developed very rough surfaces. This roughness corresponded to the extraordinary grain growth as shown in the cross-sectional views of solder joints in Fig. 11.

The reason for the accelerated grain growth and phase agglomeration was that the Pb from component leads and HAL coatings on the pads had dissolved into the BiSn joints during the reflow process and formed 52Bi32Pb16Sn, the ternary eutectic phase of the BiPbSn system (point E in Fig. 1a), which melts at 95°C. Since each cycle of the test took the temperature to 100°C, that phase became liquid at the grain boundaries and provided channels for fast atom transportation.

Although only a tiny percentage of Pb on the boards or on the component leads dissolved into the BiSn joints, the small amount of the ternary eutectic ruins the mechanical properties over the course of thermal cycling to 100°C. The joint goes from having a fine microstructure (as formed) to essentially having large chunks of Sn and Bi held together by some weak BiPbSn, which indicates that BiSn is only compatible with Pb-free surfaces.

Discussion

With all the data we've collected, it's still difficult to conclude which low-temperature alloy is the best in general. Each has different advantages and disadvantages. They offer a spectrum of melting ranges: 43Sn43Pb14Bi melts at 144°C to 163°C, 58Bi42Sn melts at 138°C, and 40Sn40In20Pb melts at 121°C to 130°C. Each has certain benefits we might want, such as 40In40Sn20Pb soldering on Au-coated surface without embrittlement, but also has trade-offs, such as BiSn's intolerance for Pb on the printed circuit board and component leads or In's extremely high cost.

Most of the test data obtained so far is positive, with a couple of exceptions. These results seem to indicate that low-temperature soldering with one or more of the alloys we investigated (or some closely related alloys) is feasible as a manufacturing technology. The exceptions include (1) the nonwetting of 40In40Sn20Pb with the no-clean flux, and (2) microstructural coarsening and early failure during the thermal cycling of 58Bi42Sn joints on Pb-containing surfaces. The first problem is being addressed in a flux development program, working with paste vendors to create fluxes intended for use in low-temperature applications with the harder-to-solder alloys such as 58Bi42Sn and 40In40Sn20Pb. The solution for the second problem has not been obtained, although several options are being pursued.

Fig. 10. *BiSn joints (a) between a Ni-Pd component lead and an organic coated copper board before thermal cycling from −45° C to 100° C, (b) between a Ni-Pd component lead and an organic coated copper board after thermal cycling, (c) between a Ni-Pd component lead and a hot air leveled board after thermal cycling, and (d) between a PbSn-coated component lead and an organic coated copper board after thermal cycling. (Reprinted from ASME Technical Paper 95-WA/EEP-4. © Copyright 1995 ASME. Reproduced with permission.)*

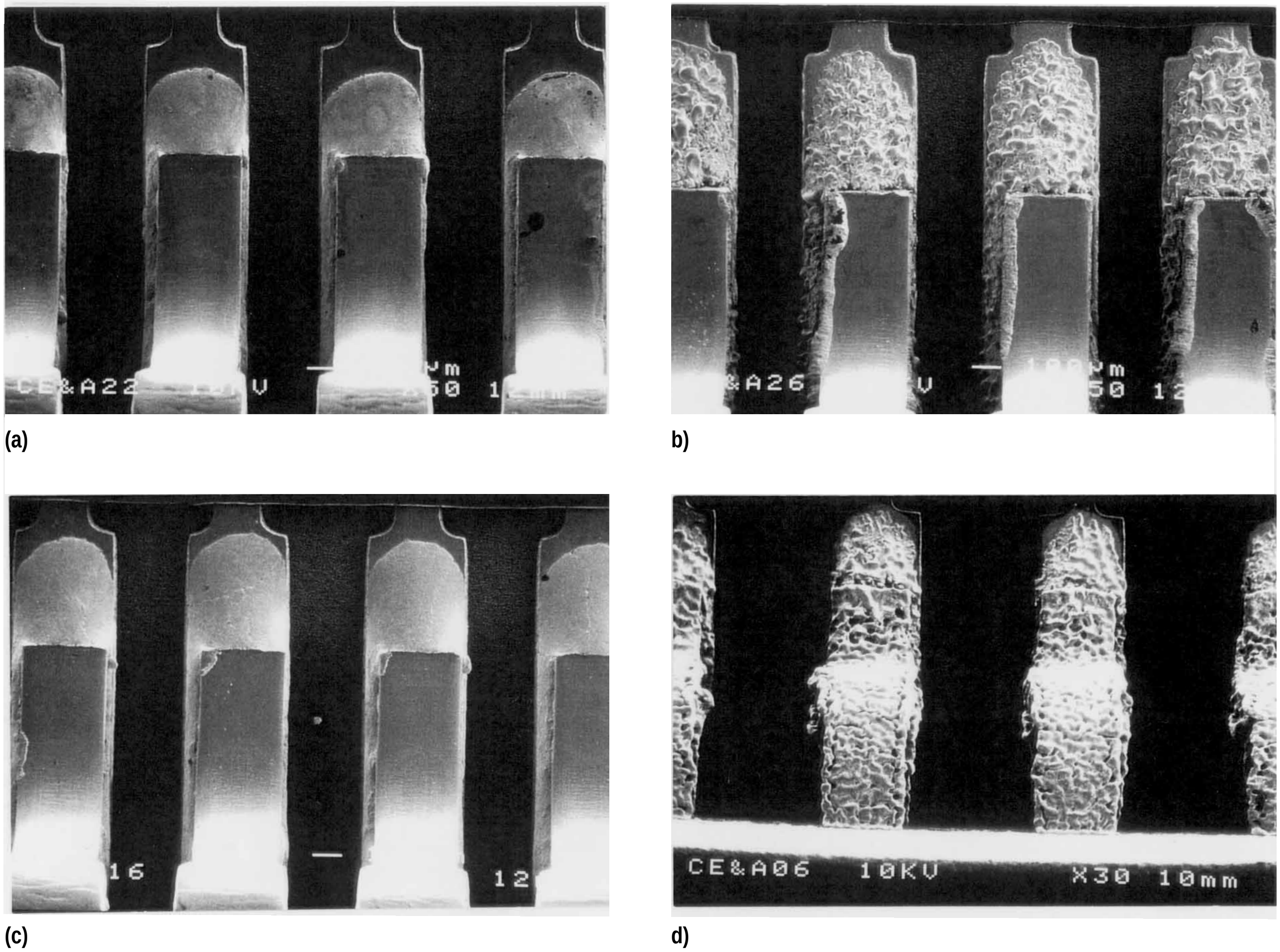
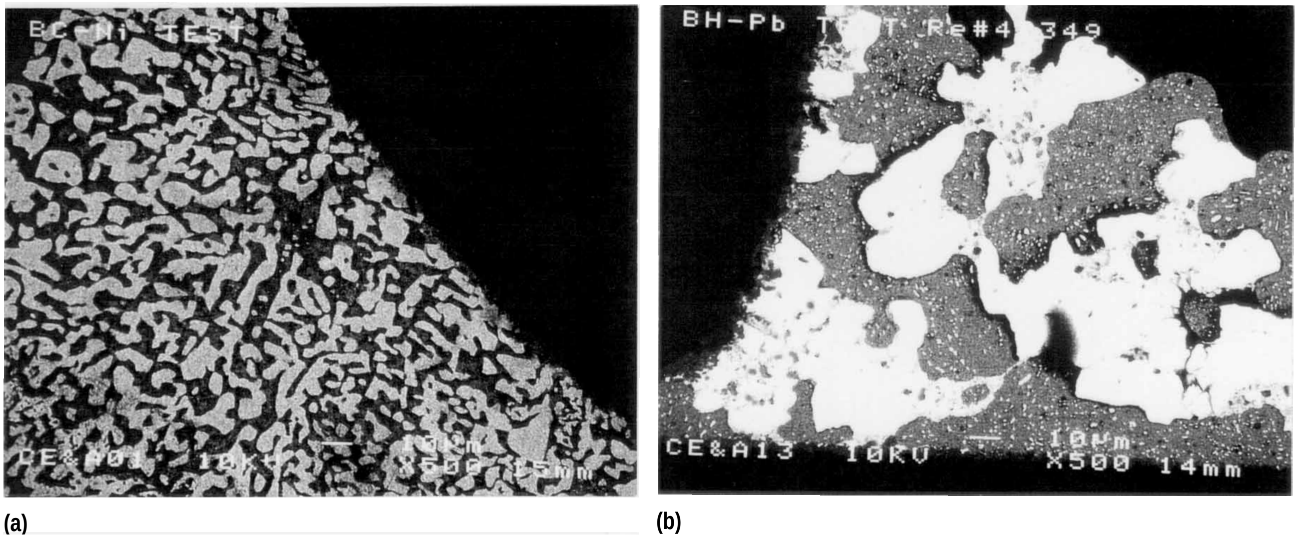


Fig. 11. *SEM cross section views of two solder joints at the same magnification after thermal cycling. (a) BiSn joint between a Ni-Pd component and an organic coated copper board. (b) BiSn joint between between a PbSn-coated component and a hot air leveled board. (Reprinted from ASME Technical Paper 95-WA/EEP-4. © Copyright 1995 ASME. Reproduced with permission.)*



Conclusion

The application of low-temperature solders in surface mount assembly processes for products that do not experience harsh temperature environments is technically feasible. Low-temperature assembly appears promising as an addition to the surface mount landscape as a way of increasing process flexibility and component reliability. However, one single alloy won't be a universal solution. Specific component and assembly requirements will have to be considered in choosing or tailoring the best solder alloy for each application.

Acknowledgments

The authors would like to thank Jerry Gleason for providing direction and guidance for this project in its early, critical stages. We would also like to thank Judy Glazer, Fay Hua, Jim Baker, Charlie Martin, and Meng Chow for their help and support.

References

1. J. Seyyedi, "Thermal fatigue of low-temperature solder alloys in insertion mount assembly," *Journal of Electronic Packaging*, Vol. 115, 1993, pp. 305-311.
2. J. Seyyedi, "Thermal fatigue behavior of low melting point solder joints," *Journal of Electronic Packaging*, Vol. 115, 1993, pp. 305-311 (sidebar).
3. Z. Mei, H. Vander Plas, J. Gleason, and J. Baker, *Proceedings of the Electronic Materials and Processing Symposium*, 1994, Los Angeles, California, pp. 485-495.
4. H.A. Vander Plas, R.B. Cinque, Z. Mei, and J. Baker, "The Assessment of Low-Temperature Fluxes," *HP EAMC Conference Proceedings*, 1995.
5. H. Vander Plas, J. Gleason, Z. Mei, and G. Carter, *Results of building BLD Ponderosa formatter boards with 58Bi-42Sn solder paste*, HP internal report, August 1994.
6. Z. Mei, *A failure mechanism of 58Bi-42Sn solder joints*, HP internal report, September 1994.

Bibliography

1. G. Humpston and D.M. Jacobson, *Principles of Soldering and Brazing*, ASM International, 1993, p. 63.
2. Choongun Kim and J.W. Morris, Jr., University of California at Berkeley, unpublished work.
3. R. Strauss and S. Smernos, "Low Temperature Soldering," *Circuit World*, Vol. 10, no. 3, Spring 1984, pp. 23-25.
4. A. Prince, "A Note on the Bi-In-Pb Ternary Phase Diagram," Materials Research Bulletin, Vol. 11, 1976, pp. 1105-1108.
5. J.R. Sovinsky, Pb-free alloys program manager, Indium Corporation of America, communication, February 1994.
6. B.R. Allenby, J.P. Ciccarelli, I. Artaki, J.R. Fisher, D. Schoenthaler, T.A. Carroll, D.W. Dahringer, Y. Degani, R.S. Freund, T.E. Graedel, A.M. Lyons, J.T. Plews, C. Gherman, H. Solomon, C. Melton, G.C. Munie, and N. Socolowski, "An Assessment of the Use of Lead in Electronic Assembly," *Proceedings of Surface Mount International*, August 30 to September 3, 1992, San Jose, California.
7. S*mithells Metals Reference Book, 6th Edition*, Butterworths, 1993.
8. *Metals Reference and Encyclopedia*, The Atlas Publishing Co., Inc., pp. 37-39 and pp. 115-116.

Assessment of Low-Temperature Fluxes

The subject of this paper is the evaluation of the wetting balance as a technique for studying the flux activity of newly developed low-temperature solder paste fluxes. The most effective configuration of the wetting balance was the standard configuration with only one change: the PbSn eutectic solder was replaced with a eutectic solder alloy with a melting point of 58°C. Since 58°C is significantly less than the proposed activation temperatures of the solder fluxes, wetting curves as a function of temperature could be studied for each of the fluxes. The resulting data was used to rank the fluxes in terms of their activation requirement.

by Hubert A. Vander Plas, Russell B. Cinque, Zequn Mei, and Helen Holder

Solder alloys with melting temperatures between 110°C and 160°C are currently under evaluation within Hewlett-Packard. An investigation of the mechanical properties of these solders has indicated that a suitable alloy can be found in the ternary or binary subsets of the BiInPbSn system (see ***Article 10***). However, alloy selection is only the first step in developing a low-temperature soldering process. A suitable flux must be chosen for use in a solder paste and the alloy-flux interaction must be studied. Thus, the ability of fluxes to activate at temperatures 20 to 30°C below the melting point of the alloy must be evaluated. In the case where different solder metallurgies have similar mechanical properties, the optimal metallurgy for a low-temperature process may be determined by the availability of the appropriate flux chemistry.

For the flux selection phase, there is no standard procedure for testing the activity of a solder flux. The degree of wetting in a system (solder, substrate, atmosphere, flux) may be characterized with a sessile spread test or by a wetting force measurement. (The sessile spread test is often simply called a spread test.[1]) The two tests are complementary. Each of the tests involves a balancing of surface tensions at a three-phase junction. For an assessment of flux activity, a dynamic measurement is more appropriate than a static measurement. Thus, the wetting force measurement is preferred to the sessile spread measurement.

The wetting balance was developed to test the solderability of component leads in a wave solder process. The technique has been adapted to characterize the solderability of surface mount component leads.[2] The focus of this paper is the evaluation of the wetting balance as a technique for studying the flux activity of newly developed low-temperature solder paste fluxes. Specifically, the Multicore MUST System II wetting balance was modified to evaluate flux activity at lower temperatures. Sample preparation and testing procedures were adapted to compare the wetting of various low-temperature solder alloy/flux combinations.

Review of the Wetting Balance

A wetting balance measures the force produced by the solder meniscus when a solid test specimen is partially immersed into a molten solder. The force is plotted as a function of time to produce a wetting curve. The measured force, F, is the sum of two components: a wetting force F_w, and an Archimedes buoyant force F_b.

$$\begin{aligned} F &= F_w + F_b \\ &= p\gamma_{lv}\cos\theta - \rho g V. \end{aligned} \tag{1}$$

where p is the sample perimeter, γ_{lv} is the liquid-vapor interfacial energy, θ is the liquid contact angle, ρ is the solder density, g is the gravitational acceleration, and V is the submerged volume of the solid. Fig. 1 shows the relationship between the solder meniscus and the wetting curve. The buoyant force, shown as a dashed horizontal line in Fig. 1, is determined by the immersed volume. Since this remains constant throughout the test, the evolution of the wetting curve reflects changes in wetting force as the solder meniscus rises. The act of immersion (Figs. 1a, 1b) causes the meniscus to curve downward, producing a negative wetting force. As the meniscus rises (Fig. 1c) and becomes horizontal, the wetting force tends to zero. If solder wets the specimen, the meniscus will climb above the level of the bath, producing a positive wetting force. Eventually, the solder meniscus reaches its equilibrium configuration (Fig. 1d) and the wetting curve comes to an equilibrium value.

Fig. 1. *Relation between the solder meniscus and the wetting curve.*

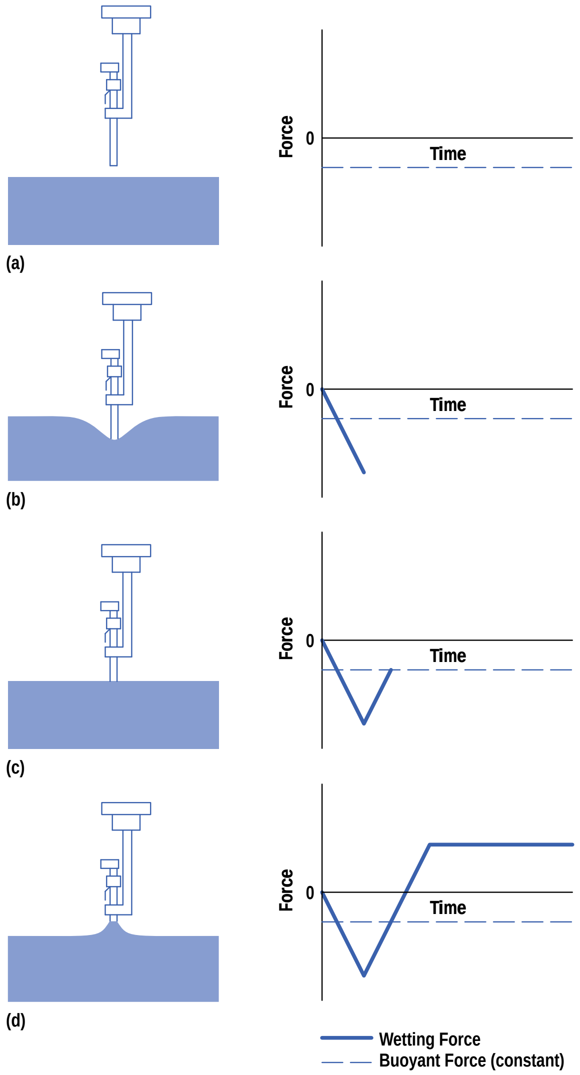
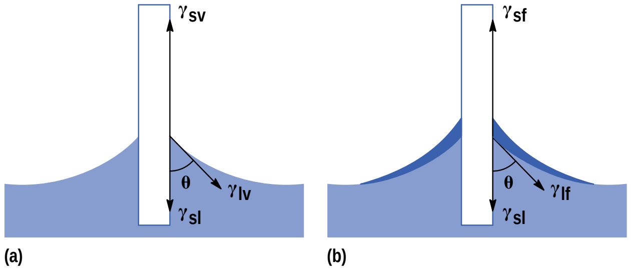


Fig. 2. *The balance of surface energies at the three phase junction. (a) Basic system. (b) More complex system with flux forming a viscous ring around the solid.*



When using wetting force measurements to study flux efficacy, it is essential to understand how fluxes may affect wetting curves. There are essentially only two points of comparison for wetting curves: the equilibrium wetting force and the rate of wetting. From equation 1, it can be seen that the wetting force is proportional to the cosine of the solder contact angle θ. The equilibrium wetting force is, therefore, proportional to the cosine of the equilibrium contact angle θ_{eq}. For simple systems, as shown in Fig. 2a, the equilibrium contact angle is given by Young's equation:[3]

$$\gamma_{sv} - \gamma_{sl} = \gamma_{lv}\cos\theta_{eq} \qquad (2)$$

where γ_{sv} is the solid-vapor interfacial energy, γ_{sl} is the solid-liquid interfacial energy, and γ_{lv} is the liquid-vapor interfacial energy. By combining equations 1 and 2, the equilibrium wetting force $F_{w,eq}$ is determined by the difference in the solid-vapor and solid-liquid surface energies:

$$F_{w,eq} = p(\gamma_{sv} - \gamma_{sl}). \qquad (3)$$

Thus, fluxes may improve the wetting force by increasing solid-vapor surface energies or by lowering solid-liquid surface energies. Notice that the liquid-vapor surface energy does not appear in equation 3.

The system becomes more complex when flux forms a viscous ring around the solid. As shown in Fig. 2b, flux replaces vapor at the original three-phase junction, altering the surface energies, which determine the wetting force. Furthermore, the mass of flux and the creation of two additional three-phase junctions may alter the wetting force by distorting the shape of the solder meniscus. Despite these complications, measurement of the wetting force should still provide useful information that reflects the efficacy of the flux.

The second point of comparison for wetting curves is the rate of wetting. This can be reasonably defined in a number of ways. In this paper, the rate of wetting will be taken as the time to reach the buoyant force ($\theta = 90°$). In the standard mode of operation, the heat for flux activation is supplied when the specimen is immersed into the solder bath. In this case, progress of wetting may be limited by the rate at which flux reduces the surface oxide of the specimen. The time to wet, t, normally follows the exponential form expected for an activated process:

$$t = t_0 e^{Q/kT},$$

where t_0 is a constant, k is Boltzmann's constant, T is the temperature of the solder bath, and Q is an activation energy. It follows that higher temperatures and more active fluxes will produce more rapid wetting. The situation is altered when a furnace preheat is used for flux activation. In this case, oxide reduction and the rate of wetting will depend on the time at high temperature in the furnace. If the flux works properly the surface oxide will have been reduced before initiation of the wetting test.

Procedure

The activation requirements of low-temperature fluxes and the effects of flux alloy interaction were investigated using a Multicore Universal Solderability Tester (MUST II). All tests were conducted in air using a standard copper wire as the test specimen. Standard samples consisting of 1-mm-diameter copper wire were prepared by etching the as-received wire to remove all surface oxides. These samples were aged at 100°C for one hour in air to produce a uniform, repeatable oxide coating on the samples.

Fluxes were applied in one of two ways. First, when the flux was in liquid form, the specimen was dipped into the liquid to produce an even coat. The liquid fluxes (Actiec 5, Actiec 2, and SM/NA) are standard fluxes provided by Multicore with the MUST II system. Second, when the flux was part of the solder paste flux vehicle, a uniform weight of flux vehicle (approximately 4 to 5 mg) was evenly spread on the surface of the Cu wire.

Since the goal was to evaluate each flux as a component of a solder paste, all of the experimental fluxes were obtained as part of the solder paste flux vehicle, that is, the solder paste without the solder. All of the fluxes—flux 1, flux 2, and flux 3—evaluated in this program were low-temperature flux systems under development. The performance of the developmental fluxes was compared to a flux vehicle developed for use with eutectic PbSn solder.

The Multicore wetting balance was set up in the three different configurations shown in Fig. 3.

- Standard configuration (Fig. 3a). This is the normal mode of operation for the Multicore MUST II. The copper wire is dipped into a large volume (6 cm in diameter by 5 cm deep) of molten solder. Two different solder alloys were used in the large bath: eutectic PbSn, which melts at 183°C, and 49Bi21In18Pb12Sn, a quaternary eutectic alloy (the numbers are the percentages by weight of the four components of the alloy) that melts at 58°C. Heating the sample occurs when it is dipped into the molten solder.

Fig. 3. *The three configurations of the wetting balance studied. (a) Standard configuration. (b) Furnace preheat configuration. (c) Microbath configuration.*

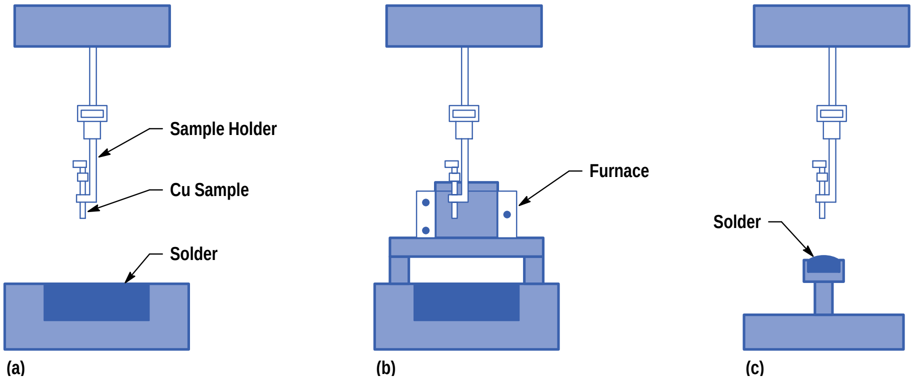
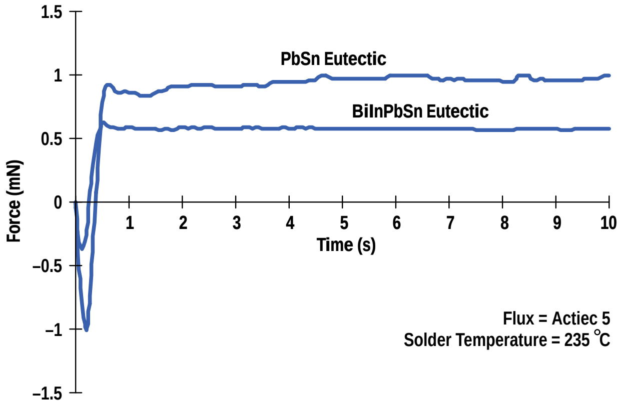


Fig. 4. *The effect of solder composition on the equilibrium wetting force.*



- Furnace preheat configuration (Fig 3b). A separate heater is attached on top of the solder bath in the standard configuration. This allows the sample to be heated before immersion into the molten solder. The preheat temperature can be controlled independently of the solder bath temperature.
- Microbath configuration (Fig. 3c). The volume of molten solder is reduced to 0.6 cm in diameter by 0.5 cm deep. This configuration minimizes the amount of each solder alloy that is required to do the tests.

When the quaternary eutectic alloy is used in the standard configuration, the bath temperature can be set to investigate the activation as a function of temperature for the low-temperature fluxes. The furnace preheat technique is a variation of the standard test that was developed to imitate more closely the thermal cycle of a standard surface mount soldering process. The furnace, mounted directly above the solder bath, provides control of the specimen temperature independent of the solder bath temperature. Finally, the microbath was used to investigate the effects of flux-alloy interaction for a variety of alloys. The microbaths are aluminum containers that were machined to sit atop the MUST II globule heater. The baths, which hold less than 10 grams of solder, minimize the amount of solder required for testing. The microbath setup yielded more reproducible results than the standard globule tests provided by Multicore.

Results

Equilibrium Wetting Force. Differences in equilibrium wetting force can be produced either by varying the solder bath composition or by varying the flux composition. Fig. 4 shows the differences in the equilibrium wetting force produced by varying the solder composition. In this test, both the PbSn eutectic solder and the BiInPbSn quaternary eutectic solder were used with the standard configuration and a bath temperature of 235°C. One flux, Actiec 5, is plotted for both alloys and clearly shows the difference in the equilibrium wetting force.

Fig. 5 shows the differences in the equilibrium wetting force that can be produced by changing the flux composition. Flux 1 and flux 2 are two different experimental low-temperature fluxes. They were both used with the eutectic quaternary alloy and a bath temperature of 190°C in the microbath configuration. Tests conducted using flux 2 exhibit a consistently greater equilibrium wetting force.

Rate of Wetting. Differences in the rate of wetting can be produced by varying the flux composition or by varying the bath temperature. Fig. 6 shows the difference in wetting rate as the hydrochloric acid content (flux composition) is varied from 0 to 5% using the standard liquid fluxes supplied by Multicore. These tests used the standard configuration with PbSn eutectic

Fig. 5. *The effect of flux vehicle composition on the equilibrium wetting force.*

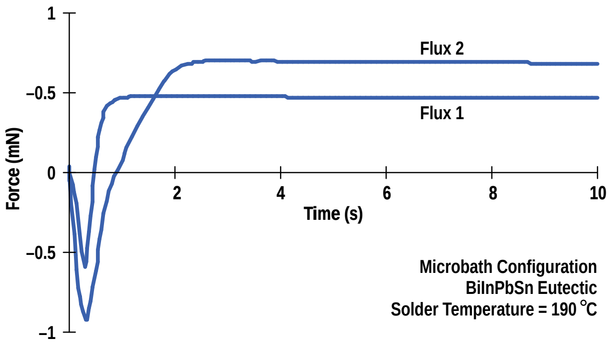
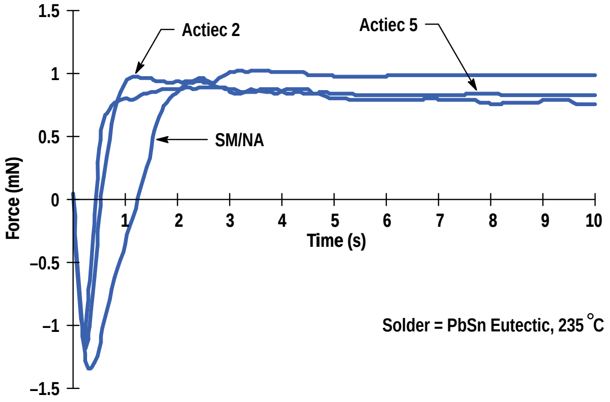


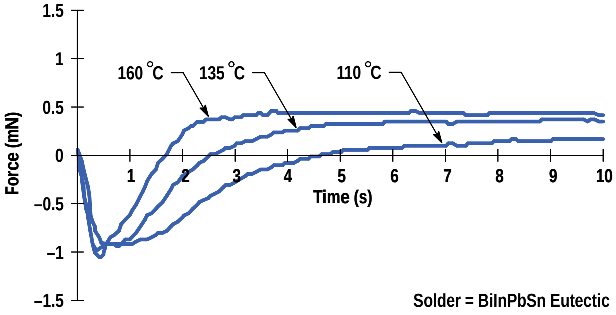
Fig. 6. *The effect of flux activity on wetting times.*



solder and a bath temperature of 235°C. They illustrate that the wetting balance is capable of detecting a change in wetting rate as the flux composition is changed.

Differences in the rate of wetting produced by varying the bath temperature are shown in Fig. 7. These tests were conducted using the standard configuration, the quaternary alloy, and flux 1. The bath temperature was varied from 110°C to 160°C as indicated. Each curve represents the average of two tests. The wetting behavior improves as the bath temperature is raised. At 160°C, the equilibrium wetting force reached its maximum value of 0.5 mN.

Fig. 7. *Wetting curves for flux 1 as a function of solder temperature using the quaternary alloy as the solder bath.*



The furnace preheat configuration provided a method of changing the flux activation temperature while maintaining a constant solder bath temperature. For the data presented in Fig. 8, the solder bath temperature was set at a constant value of 100°C using the quaternary eutectic solder. The activation temperature was changed by varying the power to the heater. In each case, the sample was submitted to a short (30 s) preheat at the indicated power setting before immersion into the solder bath. Fig. 9 shows the temperature as a function of time for each power setting. Insertion into the solder bath and the start of the 30-s preheat are at time = 50 s in Fig. 9. As the power is increased from 0 to 60% (~50°C to ~160°C), the rate of wetting improves and the equilibrium wetting force remains constant. The furnace preheat was intended to replicate the flux activation time used in a normal reflow furnace. Thus, preheat times of a few minutes were planned. However, longer preheat times produced a progressive deterioration of the wetting behavior. The low-temperature fluxes were unable to protect the specimens from reoxidizing during the longer preheats. Preheating in a nitrogen atmosphere was considered.

Fig. 8. *Wetting curve for flux 2 as a function of furnace preheat power settings.*

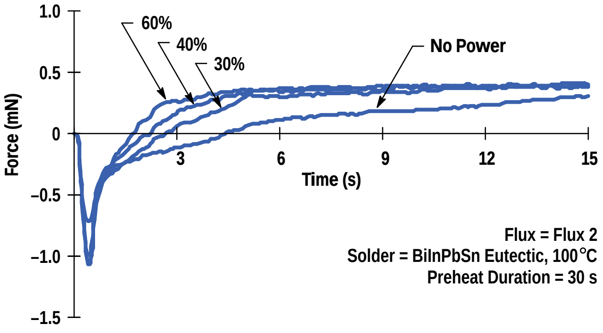
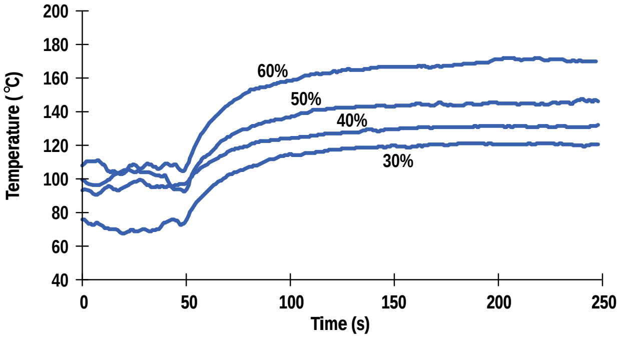


Fig. 9. *Temperature calibration curves for the preheat furnace.*



However, this option requires a major modification of the equipment that was outside the scope of this project. As a result, the objective of using the preheat configuration to assess flux activation requirements was not accomplished.

Analysis

Equilibrium Wetting Force. Figs. 4 and 5 indicate that the wetting force measurement is able to distinguish the effects of varying the solder alloy and the flux composition on the equilibrium wetting force. From equation 3, the wetting force is reduced by the difference ($\gamma_{sv} - \gamma_{sl}$). For the test shown in Fig. 4, the solid, vapor, and flux are constant. As a result, γ_{sv} should remain constant and the differences in the equilibrium wetting force are produced by variations in γ_{sl}. In Fig. 5, the solid, liquid, and vapor remain constant and the flux is varied. The equilibrium wetting forces produced by flux 2 are consistently greater than those produced by flux 1. This difference is difficult to associate with either γ_{sv} or γ_{sl}. The equilibrium wetting force is a function of both the solid-liquid and the solid-vapor interface. Both of these interfaces may depend on the presence of the flux. The solid-vapor interface will be influenced by the presence of the flux. In addition, different fluxes will remove surface oxides from the solid with different efficiencies. Thus, the observed differences in equilibrium wetting forces are difficult to associate with either interface.

Rate of Wetting. Figs. 6 and 7 indicate that the rate of wetting, defined here as inversely proportional to the time to cross the $F_w = 0$ axis, is a function of both the flux composition and the activation temperature. In Fig. 6, the solid, liquid, and vapor remain constant and the flux is variable. The rate of wetting increases as the acid content of the flux increases. The rate of wetting is dependent on the degree of oxide removal, which is a function of acid concentration and temperature. In Fig. 7, only the temperature was varied. The rate of wetting increases with increasing temperature and the equilibrium wetting force reaches its maximum at 160°C. This indicates that the flux has effectively removed surface oxides within the first three to four seconds at this temperature.

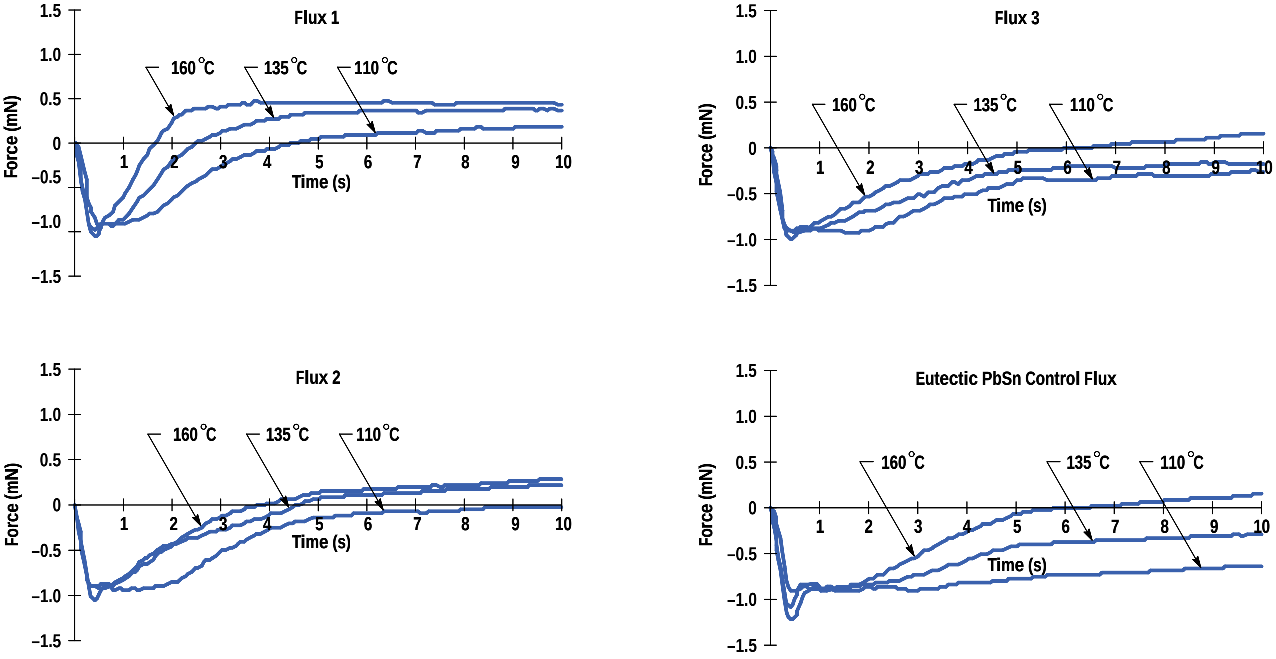
Ranking of Low-Temperature Fluxes

The four plots in Fig. 10 compare the wetting behavior of the three developmental fluxes—flux 1, flux 2, and flux 3—to the control flux formulated for use with eutectic PbSn solder. In each test, the standard configuration was used with the quaternary eutectic alloy. The bath temperature was varied from 110 to 160°C. As expected, the control flux shows no wetting until the bath temperature is 160°C. Flux 3 performs similarly to the control flux and would not be a candidate for use with low-temperature solders. Flux 1 is the most promising of the three developmental flux vehicles. Its wetting times at 110°C are equivalent to the wetting times of flux 2 at 160°C. In addition, the equilibrium wetting force of 0.5 mN is greater than the equilibrium wetting force of flux 2. In summary, the ranking of fluxes in terms of activation requirements is (flux 1 < flux 2 < flux 3 and the control flux).

Conclusions

The wetting balance as a method to discriminate the wetting behaviors of various solder alloys and fluxes has been generally successful. Replacing the standard eutectic PbSn solder with a low-temperature solder alloy in the standard configuration was the most effective method for evaluating the low-temperature fluxes. Operating in the standard configuration with a low-melting-point quaternary eutectic alloy, the wetting balance was able to rank the fluxes in terms of activation requirement (flux 1 < flux 2 < flux 3 and the control flux). The results illustrate the usefulness of wetting force measurements in the characterization of low-temperature fluxes.

Fig. 10. *Wetting curves comparing the wetting behavior of flux vehicles. The BiInPbSn eutectic solder was used for each test. The label for each curve is the solder bath temperature.*



Acknowledgments

Glenn Carter of the HP Boise Printer Division assisted during the project planning and provided the aluminum containers used in the microbath configuration. Dr. N.-C. Lee of Indium Corporation suggested the use of the low-temperature solder in the wetting balance.

References

1. F.G. Yost, F.M. Hosking, D.R. Frear, editors, *The Mechanics of Solder Alloy Wetting and Spreading*, Van Nostrand Reinhold, 1993, pp. 215-216.
2. *Ibid*, pp. 16-17.
3. *Ibid*, p. 146.