Software Management 2001
San Diego, CA, USA February 12-16, 2001

Managing Concurrent Software Releases in Management and Test

David A. Shinberg
Government Communications Laboratory
Lucent Technologies – Bell Laboratories
Whippany, NJ 07981

ABSTRACT

Customers are requiring frequent and feature rich releases of software products to support Lucent Hardware. The fundamental problem is the time required to develop and test features often exceeds the release interval. One option to meet the needs of our customers is to use concurrent development and testing; however, the use of concurrent development has several potential pitfalls. The primary problems associated with concurrent development are: 1) How to isolate the long lead features from the features that fit within a development cycle, 2) How to manage the propagation of bug fixes between releases that are in the field and releases that are still being developed, 3) Developers must be trained to work in the concurrent paradigm. This paper describes a unique approach, using existing Configuration Management tools, to managing the development load lines in support of concurrent Fixed Interval Feature Delivery (FIFD). Software load line management is the infrastructure and processes necessary to manage the content of the loads. This entails managing where and when changes are made and managing who makes changes to the source code. The goals of load line management are to: 1) provide a managed arena for software development activities, 2) limit the effort required to remove features from deliverables and 3) minimize the effort required to port changes to the software to other load lines.

1. Introduction

There is an ever-growing need to provide complex software products to customers on a short development schedule. Additionally, the customers need to be able to count on release dates for planning purposes. Instead of investing in an entirely new tool set that solves the configuration management issues associated with supporting concurrent development and support, existing tools can be used. This paper focuses on how to adapt and in some cases enhance an existing set of well-known tools to enable Lucent to excel in the market place. To this end, this project chose to implement the Fixed Interval Feature Delivery (FIFD) model of software development.[1]

To help the reader understand why the particular solution was chosen, this paper will begin with a short summary of the original environment. A short description of the desired attributes of the enhanced solution is provided in Section 3, Motivation. Luckily, some improvements were already in place before the implementation of FIFD was started. The enabling technologies are described in Section 4, Enabling Technologies. The remainder of the paper provides details on the enhanced load line management solution including discussing some potential pitfalls.

2. Original Environment

The software development environment uses Sablime[2] for change management and version control and nmake[3] for performing product builds. Sablime is a Lucent Product for performing configuration management of software components and ancillary products such as documentation. There were separate Sablime Generics for each product release, and these generics were created on an as needed basis.[4] This meant that a generic for the next release was

often created months before the prior release was completed or even entered the system test phase.

The way in which the multiple full generics were managed resulted in the need to map both new features as well as bug fixes to subsequent generics. The mapping of these changes incurs additional costs and more than likely introduces more defects.

3. Motivation

As part of the Lucent wide Project Lightning Initiative[5], this project decided to adopt a Fixed Interval Feature Delivery model for software development. The goal of the FIFD implementation was to have major releases every six months and a minor release three months later. The most important attribute of FIFD is that if a feature does not meet the quality standard for a release, that feature must be dropped from the release. In other words, the release date is held constant while the feature content of the release is subject to change. This is contrary to how many commercial software packages are managed.

There was a very limited time to implement the processes and procedures needed to support these frequent releases. Therefore, a decision was made to adapt the existing processes rather than to try to create entirely new processes.

4. Enabling Technologies

The improvements made to the software build environment and the implementation of shadow generics became the enabling technologies used to build the new load line management model. This work was part of a continuous effort to improve software development and configuration management. The application of these technologies is described in section 5, Enhanced Solution.

4.1. Software Builds

Improvements to the build environment were part of the work plan for the software development environment group.[6,7] The build environment is based entirely on nmake. The specific improvements included:

- Parallel builds resulting in dramatically decreased build times.
- Ability to perform viewpathed builds.
- Restructuring of the source tree to correspond to the hardware platform.
- Proper identification of all generated objects and their respective source files.
- Elimination of uncontrolled external shell scripts used to initiate, control and perform integral parts of the builds.

The ability to perform viewpathed builds, as well as the decreased build times, enabled the proliferation of generics that are a basis of the load line management strategy.

4.2. Shadow Generics

Shadow Generics are Sablime Generics that contain only the files required to implement a new feature or, in some, cases a bug fix.[8] This type of generic can effectively be used to isolate the long lead feature development from the current release. Additionally, due to the relatively sparse nature of a shadow generic, one can easy identify the file belonging to this new feature. Developers map the changes into the mainline generic on a file by file basis limiting the cost and duration of this integration effort. Although shadow generics were designed for infrastructure changes, such as adding a new processor board, they are also suitable for normal feature work. There is an upper limit to the number of shadow generics that can be managed per release. Empirical evidence indicates that there should be no more than two shadow generics per release.

As a practical example, we delayed the creation of a normal generic (call it G5) by four weeks. In that time, there were 600 changes to the predecessor generic (G4), while the shadow generic (G5x) contained only 120 files. Once G5 was created from G4, only two weeks were required to merge the changes from G5x into G5. If we did not delay the creation of G5, the 600 changes would have been ported to G5 from G4.

Finally, shadow generics are easily closed after the changes are incorporated in the main line generic. This reduces the long-term administration and maintenance effort for the change management system. As a side note, the use of shadow generics also reduces disk space requirements as well as maintains the performance of the change management system.

5. Enhanced Solution

The enhanced solution is a load line management strategy that allows parallel development streams to be managed in tandem. Load line management

is the tools and procedures used to create software releases. The load line management plan should be part of a larger configuration management plan. In this paper, several requirements are placed on the configuration management and other management processes. The actual implementation of those processes, however, is beyond the scope of this document.

As an integral part of configuration management, load line management provides a method to control access to source code, across releases. That is, the ability to manage where and when changes are made and by whom. The model used for load line management consists of generics that contain source files and a set of transitions that can be applied to the generics.

5.1. Types of Generics

A generic is a Sablime repository that contains source files. There are four types of generics used in this load line management strategy.

A *Feature Shadow Generic* contains a subset of the source code needed to generate the product and provides a controlled area for development before the *Development Generic* is available. *Feature Shadow Generics* are used for developing features for a given release before the *Development Generic* is available.

An *Infrastructure Shadow Generic* is very similar to a *Feature Shadow Generic*; however, it is used for infrastructure changes such as adding a new processor board or migrating to a new compiler. *Infrastructure Shadow Generics* tend to exist for a longer time than *Feature Shadow Generics*. This should not be surprising because infrastructure work is more complicated and therefore, takes more time and is associated with more risk than normal feature development.

A *Development Generic* is a fully populated generic used for feature development and some testing for a given major and minor release in that order. That is, development is complete for the major release before work is started in the *Development Generic* for the features in the minor release. *Development Generics* are never used for builds targeted Beta Test or General Availability.

A *Release Generic* is a fully populated generic used for the completion of testing including FOA, GA and subsequent field support. New development is not allowed in a *Release Generic*; however bug fixes are allowed and expected if the problem exceeds a threshold. A spelling error in a message, for example, is not likely to be changed.

5.2. Milestones

There are two categories of milestones:

- *Generic Milestones* that apply to a generic, such as creation, development complete, system test complete and closure.
- *Feature Milestones* that apply to the features themselves, such as commitment, development complete and system test complete.

A generic is composed of multiple features and unfortunately a few bug fixes. Therefore, it should not be surprising to find some overlap between the feature milestones and the generic milestones. Features themselves are composed of one or more Modification Requests (MRs).[9]

The milestones are states in the lifecycle of a feature or generic. Transition rules are used to move a feature or generic along its lifecycle. A primary objective of the transition rules is to ensure that the release date is not compromised. This is accomplished by first defining the milestones for a generic and then ensuring that the feature milestones support the generic transition dates.

Milestones that apply to a feature are:

Commitment: the feature is committed to a given release.

Design Complete: the design has been completed and reviewed.

Feature Test Complete: the developer(s) have completed testing the feature.

System Test Complete: the feature has passed system testing.

The milestones that apply to a generic are:

Created: the generic is created and available for use.

Deliverable Test Complete: all of the code for the various features has been built and passed basic functionality testing

Feature Test Complete: all features targeted for the release have passed feature testing

System Test Complete: all features targeted for the release have passed system testing.

General Availability: the release supported by this generic is generally available to customers.

Closed: the generic is closed.

The transition rules that apply to the creation of a *Shadow Generic* are different from those that apply to the creation of a *Release Generic*. Instead of listing the myriad of transition rules for this particular implementation, the usage of the transition rules are discussed in the following section.

5.3. Release Scenario

A scenario is sometimes helpful in visualizing complex systems. This scenario describes configuration management activities that are required to produce a hypothetical major release called 16.0 and the corresponding minor release called 16.1. The release number is not important and is used strictly to avoid the need to have release N and N+1.

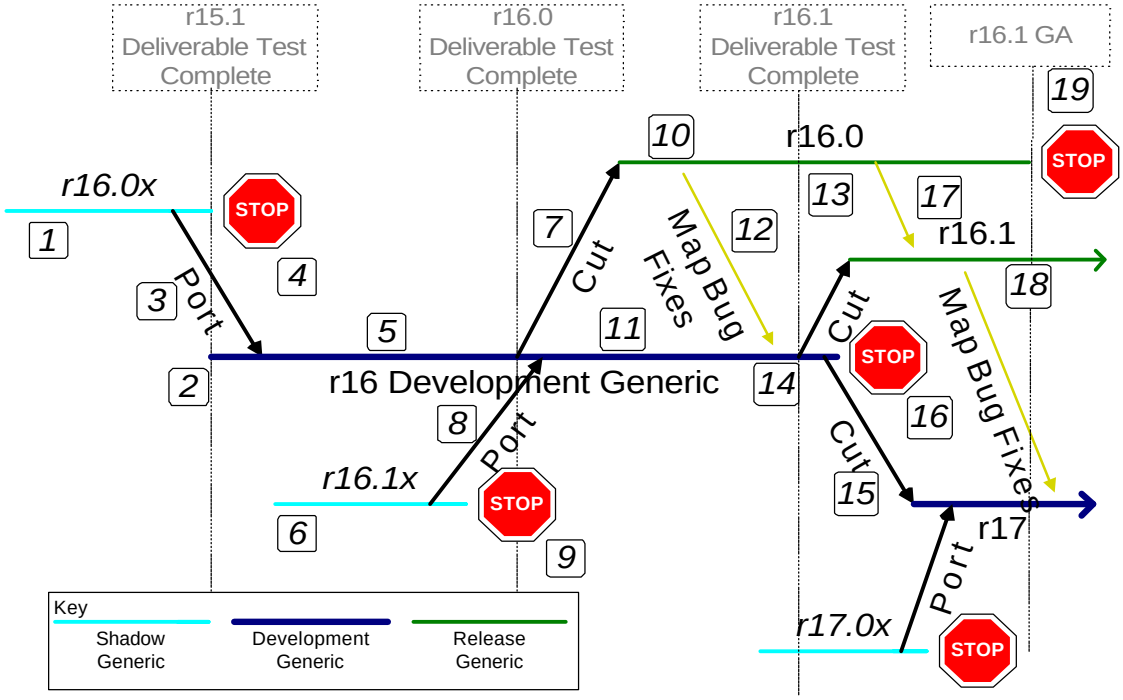

Figure 1: Release Scenario for Releases 16.0 and 16.1

Step	Applicable Generic(s)	Description
1	r15.1 → r16.0x	The r16.0x generic is created based on a good build of r15.1. Note that because r16.0x is a *Shadow Generic*, files are added only when a developer needs to change that file.
2	r15.1 → r16	The fully populated r16 *Development Generic* is created from a good build of r15.1.
3	r16.0x → r16	Developers port the changes made in r16.0x to r16 before other work is started in r16.
4	r16.0x	The r16.0x generic is closed
5	r16	Developers perform development and testing of the features targeted for Release 16.0.
6	r16.1x	The r16.1x *Shadow Generic* is created and developers may begin working on Release 16.1 targeted features.
7	r16 → r16.0	The r16.0 *Release Generic* is created and populated from the most recent build in r16 that passed deliverable testing. The creation of r16.0 allows features that do not meet the quality standards to be dropped from Release 16.0. However, those features in the r16 generic, but not in the build used to create r16.0, will likely end up in Release 16.1. This is facilitated by continuing development in the r16 generic for Release 16.1.
8	r16.1x → r16	Developers port the changes made in r16.1x to r16 before other Release 16.1 related work is started in r16.
9	r16.1x	The r16.1x generic is closed.
10	r16.0	System Testing and related bug fixes are performed for Release 16.0
11	r16	Developers perform development and testing of the features targeted for Release 16.1.
12	r16.0 → r16	Developers map bug fixes from r16.0 to r16.
13	r16.0	General Availability of Release 16.0. This includes all features that passed quality assurance and met the schedule for Release 16.0. Likely, some features were originally targeted for Release 16.0 that did not make release 16.0. These features either will be in subsequent release or dropped depending on the needs of our customers.
14	r16 → r16.1	The r16.1 *Release Generic* is created and populated from the most recent build in r16 that passed deliverable testing. The creation of r16.1 allows features that do not meet the quality standards to be dropped from Release 16.1. However, those features in the r16 generic, but not in the build used to create r16.1, will likely end up in Release 17.0.
15	r16 → r17	The r17 *Development Generic* is created from a stable build of the r16 *Development Generic.* There may be some features that are in this build that are now targeted for Release 17.0; but have not completed deliverable testing.
16	r16	The r16 generic is closed. Recall that the r16 generic is never used for releases that reach a customer. There may be some features that are killed at this point (i.e., removed from the source code tree).
17	r16.0 → r16.1	Any bug fixes applied in r16.0 need to mapped to r16.1 prior to GA of Release 16.1. In addition, r16.1 is used for any field support releases to fix urgent problems that can not wait until GA Release 16.1.
18	r16.1	General Availability of Release 16.1. This includes all features that passed quality assurance and met the schedule for Release 16.1. Additionally, problems found in Release 16.0 that were unfortunately detected by customers should be fixed in Release 16.1. Likely, some features were originally targeted for Release 16.1 that did not make release 16.1. These features either will be in a subsequent release or dropped depending on the needs of our customers.
19	r16.0	After the GA of Release 16.1, any new problems found in Release 16.0 will be fixed in Release 16.1. There may be one more bug fix release based on r16.0 for bugs that were already reported and currently being worked. After that, the r16.0 *Release Generic* is closed to limit the number of supported releases.

Table 1: Transition steps for Releases 16.0 and 16.1

A graphic depiction of the generics associated with Release 16 is shown in Figure 1: Release Scenario. The boxed numbers correspond to the steps described in Table 1: Transition steps for Releases 16.0 and 16.1. The steps provide a sequence of events that occur throughout Releases 16.0 and 16.1. Specific terminology was used to distinguish between a generic (e.g., r16.0) and a release (e.g., Release 16.1).

6. Benefits of Enhanced Solution

Recall that the goals of load line management in a FIFD environment are:

- Provide a managed arena for software development activities.
- Limit the effort required to remove features from deliverables.[10]
- Minimize the need to port or map features between generics.
- Prevent rapid growth of releases that are supported in the field.

The solution described in previous sections of this paper accomplishes the goals listed above. More significantly, this solution was based on existing tools, which greatly reduced the implementation time and the total cost of the solution. The problematic costs are not the ones associated with purchasing new tools and associated hardware. Rather they are the costs of training staff to use the new tools. The direct training costs can be measured, while, the more significant cost of lost productivity is not well understood or measured.

The rest of this section is devoted to demonstrating how the solution achieves the goals.

6.1. Managed Arena for Development

The *Shadow Generics*, which can be created as soon as a feature is committed to a release, provided an early managed arena for development. All other development occurs in a fully populated generic.

6.2. Ease of Removal

The method in which the fully populated generics are created allows specific changes associated with a feature to be omitted from the creation of the subsequent generic. Additionally in the case of a shadow generic, the work done in the shadow generic does not need to be ported to the development generic. This proves to be an excellent approach to mitigating the uncertainty associated with infrastructure changes.

6.3. Minimal Rework

The development work is complete and tested before a subsequent generic is created. Therefore, developers do not need to implement the features in more than one generic. The only changes that are mapped between generics on a regular basis are bug fixes. Another technique used to limit rework is the closing of generics as soon as they are no longer needed.

6.4. Limit Field Support

Field support is limited by discontinuing support on the major release (e.g., Release 16.0) once the minor release (e.g., Release 16.1) is generally available. Certainly, bug reports will still be accepted on the major release; however any fixes will be made in the *Release Generic* corresponding to the minor release (e.g., r16.1).

7. Caveats and Constraints

The solution focuses on a change management system to allow concurrent development by multiple teams of developers. This solution can not work without a disciplined approach to managing the software development process. One critical success factor is the management of changes made to the source code. Strict entry requirements are necessary to prevent unauthorized changes being made to the source code.

There are three distinct functional entities that must work together to ensure the stability of the code base. The entities are project management, development and configuration management. Project management is responsible for committing a feature to a given release. Development is responsible for following the defined process for performing software development including applicable documentation and reviews. Finally, configuration management is responsible for ensuring that the defined development processes are followed. For this model to be successful, the configuration management organization needs to be empowered to prevent unauthorized changes.

The different roles for feature development of the three entities are shown in Figure 2, Roles of Organization Entities. There should be no work on a feature until it is committed by project management in the top left-hand box. After that, developers need to have a MR that can only be used the in appropriate generic(s) to control the documentation and code. The development process requires that a design review be performed before changes are made to the source code. Therefore, configuration management prevents changes to the source code until the design review has been completed successfully.

After developers change the code and place it under source code control, the code and a plan for testing the changes must be reviewed. If these reviews are successful, then the change can be submitted to be built. After the code is built in a load, then deliverable testing is performed. Once deliverable testing is complete for a feature, which is likely composed of multiple MRs, project management is notified.

This process can deteriorate when no one is responsible for managing the release from a MR instead of feature basis. Project management is concerned about the features; however, Development must be concerned with the MRs.

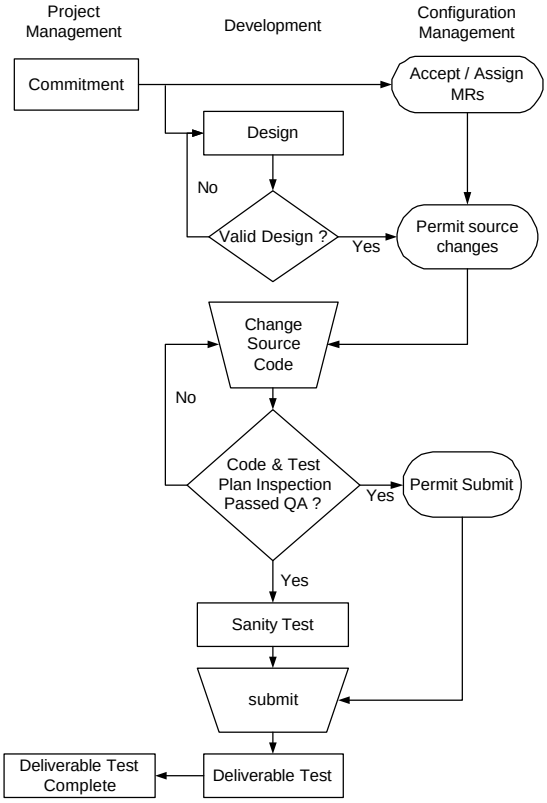

Figure 2: Roles of Organization Entities

8. Conclusion

The approach described for managing concurrent development has several advantages. The first is that it was based totally on existing tools and procedures. Therefore, this approach minimizes the impact on staff associated with adjusting to new tools and procedures. Integration of multiple features targeted for the same release is performed early in the development cycle. This prevents problems later in the development cycle with conflicting changes. This approach preserves the integrity of a release, by applying strict entry criteria. Features can be dropped up to the creation of a *Release Generic*, which is after deliverable test of a release.

The timely implementation of the fixed interval feature delivery development model for development would not have been possible without the techniques described in this paper

9. Acknowledgements

The author acknowledges the help and guidance of several colleagues and managers, without whom this work would never have been completed.

Tom Reddington, my current supervisor, provided the encouragement and the push needed to actually write this paper. Hank Schottland provided the guidance needed to promote the techniques described to the development community. Tom Morton developed the concept of *Shadow Generics* that are a key enabling technology. Tom also spent countless hours discussing the current solution as well as configuration management in general.

[1] Fixed Interval Feature Delivery was based on Timebox Development as described in S. McConnell, 1996. *Rapid Development.* Microsoft Press, Redmond, WA pp. 575-583.

[2] More information and documentation on The SABLIME® Configuration Management System can be found at: http://www.bell-labs.com/project/sablime/.

[3] More information and documentation on nmake product builder can be found at: http://www.bell-labs.com/project/nmake/

[4] A generic in Sablime is a version of the product that has been or may be released and must be maintained. It contains the source code and associated change history for the given version of the product.

[5] The Project Lightening Initiative is a Lucent wide project to reduce intervals in research and development activities.

[6] Morton, T. V., Shinberg, D. A., *Makefile Improvements to Support Enhanced Builds.* Lucent Technologies Technical Memorandum (September 12, 1997).

[7] Shinberg, D. A., *Ongoing nmake work priorities*. Lucent Technologies Technical Memorandum (July 15, 1999).

[8] Morton, T. V., *Shadow Generics.* Lucent Technologies Internal Training Presentation (May 3, 1999)

[9]More specifically, a Modification Request is the description of an enhancement or of a problem in the existing product. In the Sablime system, an MR is required to request or make changes to the controlled product.

[10] Removal is required when something goes inexplicably wrong. The load line management process can help enforce entry criteria to limit the number of feature that need to be removed; however, other parts of the organization are better suited for this task.

David Shinberg

David Shinberg is currently working on Internet research in Bell Labs examining internet mapping techniques and computer security.

He graduated with a BS in Chemistry and Computer Science from Union College in 1988 where he was awarded an MS in Computer Science in 1989. As a graduate assistant, David founded the Computer Science Crisis Center. While working at Lucent, he also received an MBA from the Stern School of Business in 1992. After joining Lucent, David was the lead UNIX System Engineer on mainframe computers for what became Unix System Laboratories. His responsibilities included defining the computer architecture used by all software developers and managing the UNIX configuration management system.

David's next assignment was the lead engineer on a real-time system that completed a successful sea trial on a submarine. He was also responsible for all aspects of software configuration management for this project. David spent two years in Lucent's wireless business working on software construction processes including implementing a new set of *nmake* based makefiles. His main interest is the efficient use of software development process from a technical and managerial perspective to gain a competitive advantage for an organization.