1-04-10 Configuration Management: An Object-Based Method
Barbara Dumas

Payoff
Configuration management (CM) helps an organization maintain an inventory of its software assets. In traditional CM systems, the only software items tracked are finished systems and versions of the files these systems use. An object-based CM system can track all forms of software—bug fixes, objects, platforms, versions, features, and projects—so the organization retains valuable pieces of software that otherwise would be lost.

Introduction
The proliferation of incompatible systems environments is one reason software developers have headaches these days. Another is that creating software is becoming more complex. The level of complexity is rising despite, or perhaps because of, the growing supply of technology available to support software development.

At least one new technology is an attempt to make software development simpler. Software Configuration Management systems are designed to reduce the complexity of software development. Whereas Computer-Aided Software Engineering tools are usually conceived as a sequence of overlapping tools—each for a phase in the software development life cycle—configuration management systems were created to maintain rational control over development as it progresses through the life cycle. One way configuration management systems achieve this end is by making configuration management as simple and integral a process as possible.

Complexity Factors in Software Development
Complexity factors with which a Configuration Management system must deal include:

- Conflicting operating systems, hardware, and standards.
- The changing nature of software (e.g., embedded shell commands, 4GL statements, executable diagrams, objects, compilation packages, features, functions, attributes, methods, tasks, and messages).
- Updated development practices and tools.
- Multiple and rapidly changing software development technologies.
- Corporate reorganizations.

These complexity factors are similar in at least three ways:
- They have a compounding effect on each other.
· They are relatively new.

· They result from fundamental changes occurring in technology and business.

Technological and organizational changes do not just make the developer's job more difficult. They also signal the new strategic importance of systems developers. It is ironic that software developers need help dealing with development just when their employers rely on them the most. Thus, even more pressure is placed on developers.

**Poor Software Inventory Practices**

If processes are crucial to an organization, so is software, because software (especially software that is customized and developed in-house) implements the organization's proprietary processes. Most processes are automated by software, and even nonautomated processes are facilitated by software. Thus, software is an asset that is as much part of a company's balance sheet as money, real estate, or equipment.

The inventory of software subroutine, programs, object libraries, and other components is often viewed as just as pertinent to an organization's competitive advantage as any other inventory. Personnel come and go, but the organization's unique processes remain and are captured in software.

There is at least one problem with most organizations' software inventories: most software items in the inventory are not labeled. The inventory consists not only of working systems but of constituent programs, files, subroutines, modules, and code segments; it is highly likely that every variation of every software item is neither accounted for nor controlled.

Most items do not even rate status as something that should be tracked. For example, a collection of code changes scattered throughout different files may not even be recognized as an entity that should be tracked, though the changes cost more than $10,000 to write, are an intellectual property, and help accomplish a major function in the organization's way of conducting business.

In most organizations, the only software items that are tracked are finished systems and the versions of the files these systems use. This type of software tracking is known as file-based Configuration Management. A side effect is software attrition. Items that do not rate as files are simply lost. It is as if the software inventory, filled with perhaps tens of thousands of dollars worth of code, were a leaky bucket—out from the bottom spill all the changes, modules, procedures, and code segments. Knowing that a file has been modified does not mean either knowing where to find the specific piece of code that does a specific job or protecting it from future edits.

Software attrition occurs every day in most software development organizations. Software that is too small, too scattered, or too abstract to register as intellectual property regularly disappears. As software disappears, so do valuable company assets.

**The Goals of Configuration Management**

Configuration management (CM) is more than housekeeping; it is the preservation of a corporation's competitive advantage in an environment of unprecedented technological and organizational change. In this environment, any additional complexity is intolerable. Configuration Management systems should minimize all complexity. Therefore, a configuration management system should be able to:

· Track unconventional software entities and map relationships between them.
Facilitate the migration of code across multiple environments (e.g., development paths, projects, hardware platforms, operating systems).

Provide a consistent environment that is adaptable to the organization’s practices, Computer-Aided Software Engineering tools, and methodologies.

Enforce organizational practices.

Provide such high-level management capabilities as access control to software items, an intuitive user interface, and distributed builds.

Tracking unconventional software entities and mapping the relationships between them is probably the most important capability because it is central to Configuration Management. The basic challenge is how to represent software assets universally, not necessarily just as files or as files in a particular context (e.g., as files on a specific hardware platform, operating system, or a particular development path). Solving the problem of how to represent software assets can aid greatly in solving the complexity problems of software development in general.

Managing Complex Data

Complex data, for lack of a better term, is data that mixes conventional and unconventional data types. This includes both compound data (e.g., mixed text, bit map, tables, sound, and video) and abstract data (as in Object-Oriented Programming). File-based systems are at a distinct disadvantage when dealing with complex software. A text file, by definition, cannot handle generated bit maps or spreadsheet tables, nor is it well suited for storing objects. Developers, however, work with all kinds of data and are often required to track relationships among such different data types as Computer-Aided Software Engineering design diagrams, UNIX source text, legacy programs implemented in Common Business Oriented Language, and new routines implemented in a proprietary 4GL.

An abstract change container is more appropriate than a file for handling complex software. Physically, an abstract change container contains only pointers and status information. Thus, such a container can reference changes to bit maps or objects as easily as changes in text files.

The benefits for managing complex data through an object-based Configuration Management system are obvious. An object-based Configuration Management system allows developers to handle different types of data by using different systems without having to learn different commands or operating procedures. Such a Configuration Management system also provides a common point of control as well as logical links among different development activities. Thus, a developer does not have to guess which changes in the source code belong to which documentation revisions or to which modifications in the CASE design tool diagrams.

Object-Based Configuration Management

One way to represent software that does not depend on files is to use object-based technology. An object can be defined as an abstract data type, which has a logical specification (i.e., it is specified by the developer). A file, on the other hand, is an example of a natural data type, like a character or an integer, whose specification is defined by the system’s physical parameters.
In Object-Oriented Programming, the developer’s object specifications are called classes. Objects are particular instances of classes and contain a data structure and a procedure, which is known as a method. Data structures and methods are called features. Features can have attributes, such as the name of the object, the size of a data field, and the location of a data base from which to process data. Attributes thus provide the means to identify, group, and configure objects. These attributes can have logical meaning (e.g., the name of a method that one object employs or inherits from another object).

As a means to represent software entities, objects in traditional object-oriented programming provide more capabilities than are needed for Configuration Management. The capabilities that matter for configuration management are:

- The capability to contain physical data (e.g., American Standard Code for Information Interchange text, binaries).
- Representation of abstract entities (e.g., product features, functions).
- Expression of logical relationships (e.g., code or hardware dependencies).

To be useful for configuration management purposes, objects need not have all the characteristics of a traditional object. For example, they do not need both a data and process section, a formal class hierarchy definition, or inheritance capabilities. Those properties are useful for reusable objects that carry out functions; in a configuration management system, an object is only a container.

A natural data type, such as an integer, is predefined in various environments, but abstract data types are implementation specific. For example, a configuration management system developer defines the specific characteristics of an abstract configuration management container. One implementation of abstract containers is the cset used in the Aide-De-Camp system from Software Maintenance and Development Systems, Inc. A diagram of a cset is given in Exhibit 1.

**Diagram of a Cset**

**Elements of a Cset**

The Aide-De-Camp implementation of an object-based change container, the cset, contains the following elements:

- **Author.** The name of the person who defined and installed the cset.
- **Date.** When the cset was defined and installed.
- **Status.** This includes such information as whether the cset has been defined, is in process, or has been installed.
- **Brief description.** A short version of the abstract.
- **Abstract.** Documentation detailing the purpose of the change and any other information the developer deems relevant.
• **Changes.** Any modification to files, relationships, dependencies, and other system components.

• **Attributes.** Logical tags (e.g., feature, operating system, and Central Processing Unit names).

The cset's main advantage over a file is that it is a logical entity that can represent an arbitrary collection of entities. In general, developers are more interested in logical entities (e.g., product features, feature enhancements, and bug fixes), which have no definition in a physical file system. Logical entities are produced by several physical entities, specifically code segments, that happen to be contained in one or more files. A cset enables the developer to capture in one place all code segments that produce a logical result.

Csets model the way most software evolves—through a set of changes. Csets are precisely a set of changes. Each line of code is tagged by the csets that create or delete the line. The line itself never disappears, so old software can be retrieved. A code segment is contained in a cset when its lines are tagged with the identifier for that cset. In the same cset are lines of code from multiple code segments in multiple files but corresponding to one logical change.

**Structures and Dependencies**

Some code segments may contain such identifiable structures as procedures, modules, or subroutine. These entities are named and have dependencies on other structures. For example, a C-language procedure may be contained within a module, which may call a subroutines. The Aide-De-Camp system is language sensitive and recognizes legal structures that appear within code segments written in different programming languages and the dependencies that occur between these structures (e.g., a C module contains another C module).

A cset may include code that contains multiple structures and dependencies. Changes to the code in the cset may have side effects for other code segments. The generation of a report on these effects is possible because the cset enables programmers to know which structures depend on which other structures, or at a more abstract level, what changes affect other changes. By knowing the effects of a specific code change on various structures, a developer can be more effective.

Although a cset consists mainly of pointers, its contents are considered to be lines of code, structures, dependencies, and attributes. A cset is an abstract device used for grouping information from various sources as single logical entities. One copy of each line of source code, for example, is stored in the object-oriented system database. Dependencies and attributes are pointers to other csets or to such external objects as documentation or hardware platforms.

**Attributes**

Attributes make it possible to create a view of the code corresponding to logical criteria specified by the programmer. For example, attributes can be used to show a certain release. Therefore, the developer is not confined to listing named versions of releases only. The developer can select or remove any changes from the software inventory to create entirely new variations of the software.

Not all attributes are supplied by the developer; some are supplied by the system. Systems-supplied attributes include file names, version numbers, and entity types.
Multipath Development

In object-based Configuration Management, software development projects are managed as groups of changes that are selected or removed according to a base version. Software releases made by the project can be more than collections of named versions of files. A release can consist of any desired set of changes (i.e., features) whether or not any previously constructed versions of files are included in the release. Changes from different development paths can be combined; different, parallel development paths can be merged; or changes from one development path can be migrated to another development path.

The advantages of object-based configuration management become clear in examining the way in which software is typically developed. Source code from one development group is copied by another, and both groups proceed to make changes to the common code. At some point during the project, one or both groups want to import changes from the other, but they cannot. They can, however, import files that contain the desired changes, inspect those files for changes, and either rewrite or cut and paste between files. In the process, common code becomes scrambled, and developers may as well be working on two entirely different projects.

Limitations of File-Based CM

File-based configuration management systems cannot help this situation, because they cannot distinguish logical changes from physical files. These systems regard a release as simply a collection of files that exist as a sequence of versions. To build a release, developers pick a version of each file to make up the release. Version control is knowing which versions of which files go into which release. However, developers do not have the option of picking parts from several versions to build a release, unless they open the files and cut the wanted pieces. In addition, these pieces do not have logical handles, as entities, that would enable the developer to use changes to compose a feature.

File-based configuration management systems use one of three approaches for managing code:

- Traditional.
- Differences.
- Integrated differences.

None of these approaches can be used to extract, migrate, and apply logical collections of code segments.

The traditional approach is simply to tag or record which files and which versions of those files belong to which release. It does not provide a means for developers to assign logical meaning to the files or changes, and developers subsequently must devise their own way to do so.

The differences approach is a more space-efficient variation of the traditional approach. In the traditional approach, each new version is stored in its entirety. In the differences approach, new versions are stored as the base version plus any differences (i.e., deltas). A version is thus equal to the base version plus all the deltas made since the base version.

Deltas have no logical meaning and are simply the difference between successive versions of the same text file. It is conceivable that a given delta implements part of a logical change and that deltas to other files implement the rest of the change. Developers
cannot select which deltas to include in a version and which to omit from a version. Deltas are used because they allow the file-based configuration management system to save space in memory. Deltas do allow variations of a program to be specified. Deltas cannot be:

- Related to specific reasons for a change.
- Logically grouped with other deltas of other files that are part of the same logical change.
- Easily extracted from the version of which it is a part.

A file delta is, however, an efficient way to store named versions of files, which a file-based configuration management system tracks for users.

The integrated-differences approach differs from the differences approach model in that the deltas and the base version are all stored in the same file. This master file is encoded so that the configuration management system knows which lines belong to which version. Again, how source code is stored is not an issue that is highly relevant to managing development complexity; however, how source code is logically identified is important.

Object-based configuration management differs from the three approaches to file-based configuration management in that object-based configuration management frees developers from having to use only named versions of files to build software releases. In object-based configuration management, each release consists of those objects (i.e., csets and logical changes) that the developer chooses to apply to a particular release, and it does not matter whether those changes are from other software projects or parallel development paths.

**Protection Against Software Attrition**

Software attrition happens not because companies destroy major systems but because programmers routinely lose pieces of their work as a result of inadequate Configuration Management. An object-based configuration management system protects against software attrition because in such a system:

- Software does not have to be a file to be managed.
- Software objects are specified as features that have value.
- The implications of any change are quickly apparent.
- No line of code is ever deleted from the data base.
- Any previous state of the software can be recreated instantly.

Developers usually do not easily lose software entities that have names, especially if those names reflect a recognized feature (e.g., password-validation-function or Graphical User Interface-window-sizing). More easily lost are miscellaneous code segments scattered among several files. A change to a few lines in a file is not the same as a change to a named product feature. In addition, developers will be less hasty in making changes if they are aware of the side effects of doing so.

Of course, the best protection against losing software assets is simply not being able to modify or delete them. In an object-based configuration management system, files are not
changed; abstract change containers (i.e., csets) are. When lines of source code are changed, a new cset is created and new source lines, reflecting the change, are written to the data base. If a change deletes some source lines, the new cset references this fact. In neither case are the original lines physically changed or deleted from the data base. A developer cannot go into the code, modify some lines, and erase the concept on which the code was based.

Managing Effects of Organizational Change

Object-based configuration management systems address many of the environmental causes of complexity in development. They also help developers cope with organizational change. The configuration management system's command facilities, which include user interface, command set, and macro command language, are features just as important as the change container for managing the effects of organizational change. Although the key is the enforcement of a development process that does not disappear when those who implemented it leave the organization, a configuration management system must nevertheless be flexible enough to fit any future needs of the organization.

Csets can provide the required enforcement and flexibility, because they provide solutions to software attrition, complex data, and multipath development. Thus, csets are the basis on which development groups can establish processes to cope with organizational change. Csets protect data from loss; they easily allow intermixing of new and old forms of data as well as the different technologies that created the data; they allow easy migration from one development path to another.

Csets do not necessarily provide the command facilities to enable the realization of all a cset's capabilities; however, a list of a configuration management system's command criteria is not difficult to specify. Those command facilities should provide:

- An out-of-the-box system that can readily increase productivity and that can be easily adjusted for future configuration management requirements.
- An easy-to-use GUI.
- A facility to automate and preserve configuration management policies and procedures.
- A language that exploits the capabilities of csets.
- Such security and data integrity features as the ability to assign certain rights to users.

Multipath development is probably the most important configuration management capability for dealing with organizational change. Mergers and acquisitions—two causes of organizational change—often occur so that one organization can acquire another's software and integrate it with its own. For example, a software company may wish to improve its product by offering another company's product as an integrated function. Multipath development facilitates such integration. In addition, the formation of new organizations usually involves the development of new software, which can be accomplished through parallel development paths.
Conclusion

Complexity may be the most pressing problem currently facing developers. Software is becoming increasingly complex because of more varied data types, among other factors. There are also more platforms, operating systems, and markets, all of which require separate but coordinated development paths. Technology and organizations are changing, and those changes not only increase the complexity of development but increase the potential payoffs and risks.

Software assets have value, but these assets are being lost because of poor complexity management. The attrition of software is no longer a cost that companies can afford to keep paying.

Configuration management systems can play a major role in reducing complexity in the development environment. To fulfill this role, Configuration Management systems must track software assets and not just the physical files that hold those assets. Flexible, powerful command interfaces and languages help configuration management systems track software assets, but the software change container is the key. Change containers must be sufficiently robust to map logical entities (e.g., features and bug fixes) to physical entities (e.g., code segments and other structures scattered among multiple files). Object-based configuration management, which incorporates such logical identifiers as attributes, dependencies, and object status information, is a powerful alternative to file-based configuration management. Object-based configuration management systems allow developers to keep track of bug fixes, objects, platforms, versions, features, and projects and to concentrate more on developing software.

Author Biographies

Barbara Dumas

Barbara Dumas is vice-president, of Engineering at Software Maintenance and Development Systems, Inc., Concord MA