By Alan Moore,
Vice President of Product Strategy,
ARTiSAN Software Tools.

Scaleable Model Management for UML Models

The increasing popularity of UML, and the extended range of its supporting modeling tools, has led to increased modeling awareness throughout the development life-cycle from early requirements analysis to detailed design and coding. One of the consequences of this awareness is that project managers are demanding that models should receive a similar level of management to other project assets. The benefits of modeling tend to be greater as teams grow larger. This is because the advantages of a uniform description, a shared specification and a single blueprint have greater value as a team becomes less able to remain synchronized through simple word of mouth and ad hoc documentation. Ironically, many of the modeling tools available for UML today do not scale to large teams, particularly large teams working on a single model. This article talks about the management needs of large modeling teams and the features that a modeling tool needs in order to support them.

MODEL MANAGEMENT

Model management is to models what configuration management (CM) is to files. Models may contain many hundreds of often fine-grained items, with a complex web of dependencies, and so in order to control them the familiar concepts of configuration management have to be extended.

In the remainder of this article we explore the key issues of working with models, and discuss how the concepts of configuration management need to be extended to deal with models. In order to do this we loosely follow a standard development process, starting with analysis, through design to coding.

THE EARLY STAGES

During analysis and early design, the model is evolving from many different viewpoints. There are frequent significant structural changes and the boundaries between modellers change as the model structure is evolved towards a shared view of the problem and its solution. For example, the UML technique of decomposing use cases using interaction diagrams results in an object model whose classes evolve as each use case is explored. Frequently, use case exploration is handled in parallel by different domain experts whose principle area of interaction therefore is the object model.

This type of collaborative approach enhances the power of many modeling techniques supported by UML, but it imposes on the modeling tool a requirement to provide support for multi-user collaboration.

Packages

As a model grows, the class/use-case/actor level of granularity quickly gets confusing without some organizing principle. Fortunately UML provides the package for exactly this purpose. From UML [1]

”A package is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain subordinate packages as well as other kinds of model elements. All kinds of UML model elements can be organized into packages. Note that packages own model elements and are the basis for configuration control, storage, and access control. Each element can be directly owned by a single package, so the package hierarchy is a strict tree.”

There are various techniques for organizing models into packages, which is a topic beyond the scope of this article, but the package concept has proven very powerful as the basis for model management.

Figure 1 shows a stylised view of a candidate package hierarchy for the analysis activity. A number of domain experts work on their own detailed functional areas, and have organized the structure of the ’Function’ package accordingly, but all are contributing to the ’Object’ sub-package of the ’Architecture’. The ’System Engineer’ is interested in the whole of the architecture section, whereas the ’Client’ representative is only interested in the top-level functional definitions.

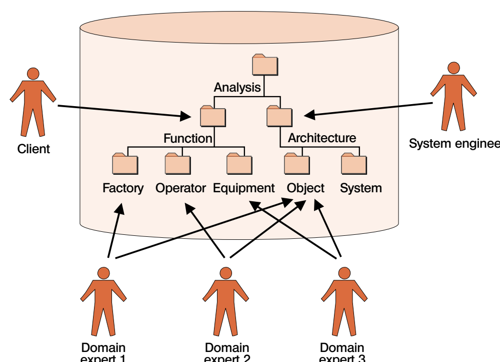

Figure 1. A multi-user team collaborating on a model.

Model Access

The traditional paradigms for multi-user software development (the ones supported by traditional configuration management) either:

- Serialize changes by sharing a single locked copy of a model, which is frustrating and can elongate the analysis/design process;
- Develop in parallel on several separate copies of a model, and then reconcile conflicting changes, either frequently or occasionally. Of the two:
  - Reconciling frequently is time-consuming and error prone.
  - Reconciling occasionally results in a lot of miscommunication.

If the team is using these paradigms, then performing multi-user collaboration is inefficient. Shared access to packages and the consequent interaction between modellers is an inherent and important part of collaborative techniques, but with either of the approaches described above, this is costly to manage.

Multi-user collaboration is far more efficient when all users are provided with simultaneous on-line access to a single centralized model. This requires an active model repository that automatically propagates changes made by one user to all others who are accessing the same model.

Multi-user Access Control

Given the complex structure of models and the collaborative nature of many modeling activities, sharing a simple lock on a package is often too coarse, because it doesn't allow controlled access for more than one user. Access control for packages needs to be multi-user/group to allow a set of (approved) users to have simultaneous access, and to restrict the access of others. Each user working on the model will then have a set of access privileges that is valid for their needs.

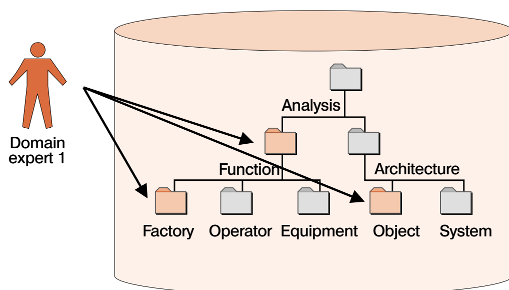

Figure 2. Access to packages for Domain Expert 1.

Figure 2 shows the view of the analysis package hierarchy seen by Domain Expert 1; she has write access to the 'Factory' domain, the high-level function definitions and the object architecture. A helpful tool User Interface (UI) will indicate the current user's access rights, perhaps by greying read-only packages as in this example. Each user will see their own version of the hierarchy depending on their access permissions.

CM Integration

It is neither necessary, nor desirable, for a modeling tool to provide their own model management facilities, where these are similar to those offered by well-established Configuration Management (CM) tools. CM tools are used by many projects as the centralized repository for their project assets, including code, test cases, documents and models. They offer robust version control, access control and often configuration control of the assets (represented as files) under their control. Integration with CM tools is part of a complete and integrated model management solution.

Version Control

In CM, each asset is called a configuration item (CI for short). A CM tool will store successive versions or snapshots of a CI, often using some efficient form of storage, such as storing just the changes (or deltas). This process happens when checking in a CI. The most significant advantage of this approach is that later on this version can be recovered, for example if a later version has gone awry and needs to be discarded.

Access Control

CM tools allow specified users to gain exclusive access to configuration items by locking them. This process is called checking out a CI. Once checked out, a CI is locked and cannot be checked out by anyone else, so multiple updates to that CI are prevented. A check-in will typically unlock a CI, so that someone else can subsequently check it out.

Configuration Control

A configuration links together all configuration items used to create some significant project deliverable, such as a customer build of the system. If later in the project work needs to be done against that specific configuration (often called a baseline), then the CM tool can restore it from the appropriate CI versions.

A project configuration will contain much more than just model data, and so a CM tool is required to maintain a baseline of all the project assets that contributed to a deliverable.

Placing a Model Under Configuration Management

Standard CM tools can be used to provide model version control and to integrate model assets into the overall project configuration. However, unlike CM tools where the unit of configuration is a file, it is less clear what the equivalent for models should be. An entire model is too large to be a CI, because every time someone wished to record a version of anything, the whole model would be versioned. The other extreme would be to make every element - every class and use case - a separate CI. This does not scale well, because for larger problems, the number of individual items to check in and out would become unmanageable. The UML package turns out to be the right organizational unit for a CI, because it can own any number of any model elements, and so provides a flexible container.

In order to manage a model's content using a file-based configuration management tool, each package needs to be exported as a file and checked in to the CM repository. The locking provided by the CM tool then provides added protection, checked-out packages are accessible in the model to which they were

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

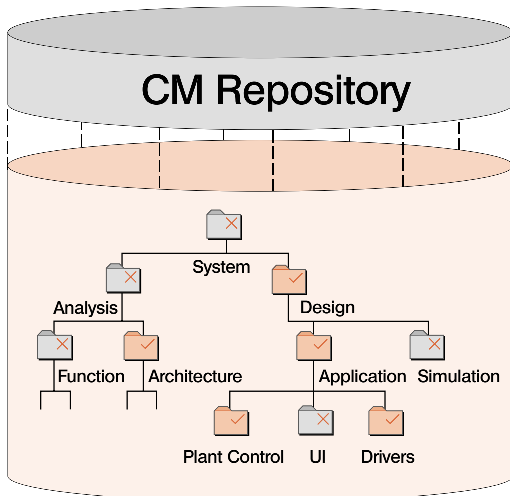

Figure 3. The model under Configuration Management.

checked out, but not elsewhere. Subsequently, checking in a package will create a new version of the package file in the CM repository. Within, the modeling environment however, the model access control should still allow multi-user access to checked-out package, or the potential for multi-user collaboration on that package is lost.

Figure 3 shows the model at a later point in the development cycle, with each package now under configuration management. The model still contains all of the packages, but they are now 'mastered' by the CM repository. The tool UI should offer menu choices to perform standard CM operations and indicate which packages are checked out (for example a red tick) and which are not (a red cross).

LATER DEVELOPMENT

During the early stages of modeling (and depending on project preferences at later stages as well) the most effective way of working is for the entire team to share a single centralized model. Administration, such as backup, is straightforward, data integrity is not compromised by problems with client machines, and the ease of communicating changes will be greater. However, even early on, and certainly as development proceeds, there are situations where the ability to distribute and/or isolate a model is important. This section will look at two cases and show how integration with a CM tool will help in each case.

Distributed Models

Although in an ideal world, everyone who wishes should be able to work from the central repository there will be cases where this is not possible, or desired. For example, the repository may not be accessible from sites outside of a WAN or VPN, or an individual may wish to work on a model while not connected to any sort of network, necessitating a local copy to be made.

The integration with the CM tool can help here, either directly if the CM tool is capable of supporting distribution, or indirectly by passing controlled files (in this case representing a condensed form of the package) to remote sites but maintaining a centralized lock.

Figure 4 shows a hypothetical case where the customer has a separate site from the development organization; the CM Repository is now effectively the master for both, providing control across the distributed model. The 'System Engineer' periodically travels to the customer site where they both work on the functional definition; hence the 'Function' package is often

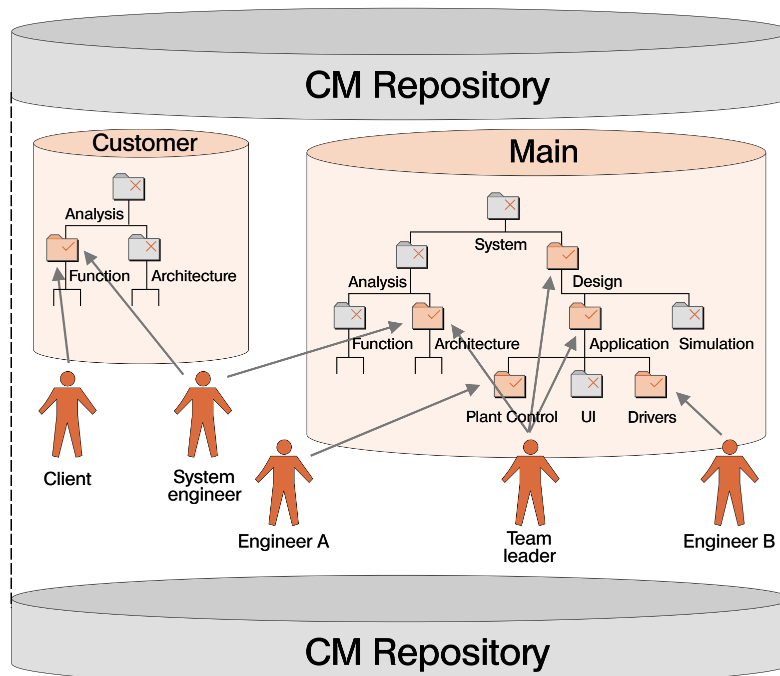

Figure 4 Users at two sites share a single distributed model.

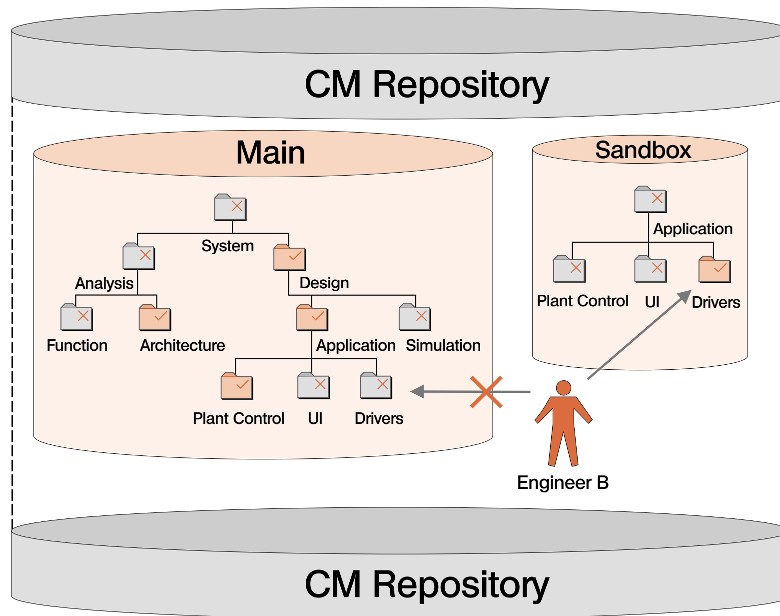

Figure 5 An engineer working in a remote copy of the model.

checked-out to the 'Customer' copy (sometimes called a replica) of the model, preventing users on the 'Main' replica from updating it.

Note that several users are still sharing each model, with access control at the package level. For example, the 'Team Leader' may need access to the whole of the 'Design' and the 'Architecture', whereas individual engineers may only need access to individual design packages.

Often, a given site may only require a partial replica of the configured model, i.e. with only some of the packages available. Partial replicas take up less space, are quicker to construct, have less clutter, and allow certain operations, such as searches, to complete more quickly.

Merging

Unlike source files, which when retrieved from the CM repository simply overwrite the current one, a package when used to update a model needs to update the web of interconnections from its model elements to those of existing packages. It is important that a 'merge' feature is available to do this, although if there is just a single source of update, as in the shared single lock approach shown previously, then the process is completely automatable. In reality of course, there are interconnections between source files, it's just that CM tools don't manage or recognise them, and it's only during compilation and linking that you find out any 'connectivity' problems.

Isolated Models

As the design proceeds, stable boundaries between groups, and even individuals, will result in more autonomy and a lower degree of coupling. This stability is more easily achievable if the interaction between packages is understood and controlled.

Three techniques can be employed to support stable interaction:

- Controlled access to model elements, via model element visibility.
- Explicitly defined package dependencies so that package usage can be traced in the event of changes to publicly visible elements.
- Isolated (sandbox) models so that any changes to required package interfaces need only be accepted when the isolated team is ready to take them.

Dependencies and Visibility

In UML, packages can reference other packages, modelled by using an access dependency. This allows the owner of a package to know which packages (and hence which users) will be effected by changes to the contents of their package. To further tighten control, items in packages all have a visibility property; the two principle visibility levels are:

- Public - any package with a dependency may use these items
- Private - only items within the package may use these items;

In practise the package owner therefore need only be concerned about changes to public items, allowing them complete freedom to amend the private items. This in a sense defines a package interface.

Consistent use of dependencies and visibility can limit the number of times that conflicts take place and ensure that when they do all interested parties are made aware of them.

Sandbox

The sandbox option is often used when an engineer is working in a dedicated edit/compile/debug cycle, and they do not want to be distracted by changes from other sources. Code needs to be syntactically correct

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

in order to compile, and changes to required elements in other packages can destabilize the code base, especially with languages that do not separate interface from implementation. This distraction is especially acute when code is generated from the model.

In this case the engineer will create a separate repository (either locally or on a server) and replicate the relevant parts of the model, checking out the piece that they are modifying. Periodically they will update their model from the CM tool to pick up changes and integrate them.

In Figure 5 Engineer B has checked out the 'Drivers' package to his own 'Sandbox' repository, thus insulating it from any changes in the main model; it cannot therefore be altered in the main model. Periodically the contents of other packages, such as the 'Plant Control' package, can be refreshed from the CM Repository using a 'Get Latest' operation.

Visual Difference

Differencing is a technique for identifying the differences between two 'versions' of the same thing. This is very useful to allow a user of an isolated model to compare their current model with that in the CM repository, in case anything useful has been changed or added. In more formal change-control environments this feature can be used as an audit trail of the changes made to a package.

Parallel Development

Sometimes it happens that a parallel build of the system is needed, either to create a true variant (perhaps for some special customer), or to proceed with target-specific testing while the main team continues with the main functionality of the system. CM tools often support this by branching the configuration (project) and allowing some files to have multiple copies proceeding independently. The model management features described previously, when integrated to a CM tool, can be used to handle these situations for models. However, merging the changes from multiple variants does require manual intervention, and so parallel development is not recommended unless absolutely necessary. In the absence of a multi-user modeling tool parallel development is often effectively the norm, which can significantly and unnecessarily increase model management overhead.

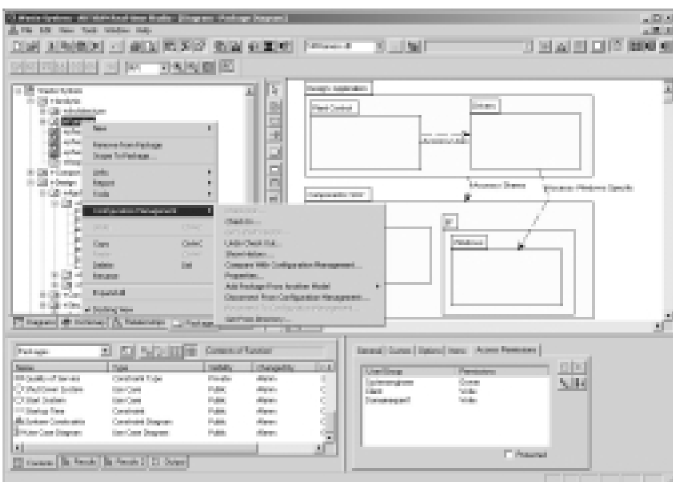

Figure 6. Model Management Features.

Shared Components

A centralized CM tool makes an excellent place to store common components and their specifications. A package can be used to document a component - public items comprise the interface, with the implementation being private to the component builders. This package can then be used in many different applications by sharing it into many different models, with the CM tool keeping track of all its users. Good examples of components are RTOS's or standard Board interfaces.

CONCLUSION

Models are important project assets, whose relevance and value cover the entire project lifecycle from early analysis through to coding and to maintenance, and so they need the same degree of configuration management as other project assets. However, not only are models more complex than simple files, modeling is also a team activity, which means that more sophisticated configuration management is needed; this is termed model management.

At an early stage, modeling is a collaborative activity requiring shared access to models and instant propagation of model changes to all users. During later development, stability is key with defined team/individual boundaries and explicit dependencies.

Model Management is an Extension of Configuration Management

Model management extends and adds to the traditional CM feature-set. Integration with a CM tool is very important when it comes to integrating model assets with the rest of the project configuration, and will also provide version control and distributed development support. In addition, the following extensions to CM paradigms are needed in the modeling tools themselves to provide adequate model management:

- The UML package artifact replaces the standard file as a configuration item (it needs to be converted to/from a file for integration with traditional CM tools).
- Multi-user rather than single-user access control is needed to support collaborative modeling techniques.
- Visibility and package dependencies enable stable 'component-based' development during later stages of the project.
- Model-specific merging and visual differencing provide support for distributed development.

In Figure 6, a snapshot of the Real-time Studio UI shows many of the additional features required to extend configuration management into the model domain.

The left-most pane shows an obvious graphical metaphor for the package concept, a hierarchy of folders. These folders have very similar properties to

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

Windows™ folders, in that they can contain any type of element, including diagrams, and can be nested. The bottom right pane shows the access control status for the 'Function' package; the 'System Engineer' in this case has Owner access, 'Client' and 'Domain Expert' have Write access. The bottom left pane shows the contents of the 'Function' package, while the diagram in the main pane shows dependencies between design packages. Visibility is indicated by a + or - in the Package Explorer view, or by the keywords public/private in the Contents view.

Multi-User Access is Key

The addition of the model management features described above can significantly enhance the success of large modeling projects, but the key scalability feature as models and teams grow is the management of multi-user access to models. This may be, at one extreme, on-line collaboration in a 'war-room' setting, or at the other extreme, isolated 'sandbox' development.

Today's modeling teams need an active object repository to provide instantaneity of communication, overlaid with a package-based architecture to provide firm boundaries and secure access control. Such a package-based object repository is a natural basis for both centralised and distributed development, furnishing flexible multi-user support throughout the project lifecycle ■

Alan has 15 years of experience in the development of real-time and object-oriented methodologies, and their applications in a variety of problem domains. He has been actively involved in product development, training and consulting related to OOAD and structured development tools during that time. Alan has co-authored a book on GUI design and published several papers, and has lectured on a wide variety of analysis and design issues.
Alan is responsible for the specification, planning and management of the ARTiSAN product strategy. He is the author of ARTiSAN Real-time Perspective, a pragmatic approach to the development of real-time systems and is an active participant in the Real-time Analysis and Design Group (RTAD) of the Object Management Group (OMG).

REFERENCES

1. OMG Unified Modeling Language Specification Version 1.3, First Edition: March 2000

Dedicated Systems Magazine

ADVERTISEMENT INDEX

QNX SOFTWARE SYSTEMS	2
TEXAS INSTRUMENTS	19
IMAGE MEDIA	23
NATIONAL INSTRUMENTS	47
MICROGOLD	67
POLYHEDRA	75
EMBEDDED SYSTEMS 2001 - NURNBERG	79
ENEA OSE SYSTEMS AB	91
RTS 2001 - PARIS	95
APPLIED COMPUTING CONFERENCE & EXPO 2001	99
VMIC	115
VMETRO	116

For more information on advertisement opportunities, call +32-2-520.55.77 or send an e-mail to info@dedicated-systems.com

EDITORIAL CALENDAR

1Q 01 Development methodologies & tools
(UML, lower and higher CASE tools, New compiler technologies, etc.)

2Q 01 Chips Issue
(Soc, DSP, Processsors, Controllers, Debugging Tools,..)

Special Edition
Applications & Dedicated Software

3Q 01 RTOS Update
(Freeware, standardisation issues, safety-critical RTOS,...)

4Q 01 System Architectures
(SoC, Software Intensive Systems, Complexity issues, Networking..)

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)