
Kim Letkeman
July 03, 2007

IBM® Rational® ClearCase® is an enterprise-class software configuration management (SCM) system that you can use in myriad configurations to satisfy virtually any artifact versioning and storage requirements. However, this very flexibility can make it difficult for you to choose the best way to use ClearCase to manage modeling projects. This article discusses several options, and recommends some best practices for using ClearCase with models in IBM® Rational® Software Architect.

View more content in this series

Introduction
Managing model files is a very different exercise from managing traditional 3rd-generation language (3GL) source files. Model files have many hidden interconnections as required to implement the UML specification. Source language files have some number of similar interconnections, but they are explicitly defined as imports or references, and are very easily repaired.

It is not easy to repair damage to model interconnections, because the XMI text format that is used for serialization unfortunately propagates all of these interconnections out to the file format. XMI is difficult to read, and references such as are used to implement these model interconnections are extremely difficult to repair, because they are complex and not obvious to the inexpert observer.

It is therefore of the utmost importance that you protect model integrity at all stages of development, in order to avoid damage of this sort.

Model integrity
The first and foremost method for protecting model integrity is to keep the model files synchronized in your workspace. This means that the model files should all be of the same baseline so that all interconnections are intact and valid. A baseline is simply a label across a set of files that are at
the same point in their evolution. In other words, all change sets have been accepted and there is no partially completed work. In ClearCase, several important commands work with baselines, for example it is possible to set your workspace context to a baseline at any time by rebasing to that specific label. The Unified Change Management facility is built around baselines and provides significant workspace integrity protection by virtue of its ability to avoid out-of-synchronization workspaces.

So, for example, you must never respond Yes when IBM® Rational® ClearCase® asks if you would like to check out a later generation of a file than is loaded in your workspace! This will put the models out of synchronization by definition, and can lead to model corruption.

Merging models

The "check out latest" facility exists to provide the ability to reach through your view into the repository and retrieve the latest checked-in version of any artifact in order to prevent merges. This is necessary for artifact types for which no ClearCase-type manager or three-way merge support exists. This includes binary files, documents and presentations in vendor-proprietary formats, and so on. Never use this feature to accept a more recent version of an artifact when you work with model files! IBM® Rational® Software Architect fully supports the ClearCase merge methods. Note that the feature is best avoided with 3GL source files, but the ease with which those can be repaired will provide plenty of temptation, because it is essentially impossible to fatally damage 3GL source files.

Rational Software Architect contains several model integrity protection mechanisms when merging artifacts that have evolved in parallel. One key mechanism is the use of composite deltas to group related changes into atomic sets – that is, sets of changes that must be accepted or rejected together. This guarantees that Rational Software Architect will not find the merge result unpalatable or confusing.

A simple example of an atomic composite change set is the gesture drag. Having moved a shape from one (x,y) position on a diagram to another, the compare tool will show two deltas – change in x position, and change in y position. The compare support will group these two into a drag composite, and will make it atomic. This atomicity prevents the merge tool from allowing the choice of accepting one part and rejecting the other. This means that the only two valid positions on the diagram are the before and after positions of (x1,y1) or (x2,y2). The other two possible positions (x1,y2) and (x2,y1) are prevented by the atomic change set. Thus, the compare and merge facility cannot get into a state that never existed in the original or the changed version. Although trivial, this example shows the integrity protection in action. Compare support for models works very hard to avoid states that the tool did not create, which prevents the tool from having to deal with combinations of elements and notation that may not be legal.

It follows that Rational Software Architect can do a better job of protecting model integrity when it has access to a large part of (or the entire) model. You should consider this carefully when you partition your models. Breaking the models up randomly, or too finely, could lead to a breakdown in integrity protection. The ideal is a single model file. Of course, this only works while the model file remains reasonable in size for the typical machines in use in the workplace.
Performance and memory

Models expand between 4 and 6 times when loaded in memory. This means that a 10MB model will use up to 60MB of heap space. A standard three-way merge requires 4 copies of the model to be in memory, so the heap space used will hit 240MB.

A machine with 512MB RAM will use a significant portion of its base memory just to load the operating system, leaving little memory available for Rational Software Architect and all your other applications. All applications will have to share what little memory is left. It follows that system memory is the single most important resource for application performance. 512MB is relatively small by current standards, and will likely force a fairly significant model partitioning exercise. 2GB, on the other hand, helps all applications perform more quickly, which will naturally have a positive impact on developer and modeler productivity. The resulting overall system performance improvement that you gain from increasing your RAM could save you many minutes of developer or modeler time per day.

If you need any further justification to upgrade your modeling or development machines, here are two strong arguments:

### Single model best practices

Note the phrase “physically partitioning a single model” at the end of the performance and memory section. These best practices apply to a single logical model, however large. They can also be applied to a cluster of related models that would make sense sharing a repository and related integration streams. However, a different article will discuss enterprise issues and major architectural decomposition issues that affect the larger scope of logical partitioning.

- Modeling is an intellectual pursuit, in many ways similar to writing and programming. Indeed, these are all high-concentration tasks. To get exactly the right meaning into your models, you must manipulate them until the model reflects exactly the intent of the modeler – addressing all details, no matter how subtle. If each operation takes 3 or 4 times longer than it should (minimal base memory does that and worse to any major application), then the modeler or developer may begin shortcutting the refinement process, with the result that the output (model, document, system) is of lower quality than expected. This can mitigate the purpose or value expected of the task in the first place.
- The difference between 512MB and 2GB is so dramatic for most major applications that you can save tens of minutes each day that are otherwise spent waiting for the machine to boot, for application windows to swap, and for major applications like Rational Software Architect to pause and continue. Even 15 minutes per day at current salaries will mean an ROI on the order of days or weeks for that investment in RAM. All major applications and IDEs can benefit. This issue is not specific to Rational Software Architect at all. The productivity benefits can quickly outweigh the costs of upgrading development machines to higher levels of system memory.

That said, there are a couple of obvious partitioning choices that can help address memory footprint issues, so that you can use even small machines with very careful partitioning:

- Leave the model in large chunks and merge
• Break the model up into very small pieces and avoid merging

Rational Software Architect and ClearCase support both techniques very well. These two ends of the partitioning spectrum are, in fact, the two best practices for physically partitioning a single model.

Introduction to ClearCase

ClearCase is an enterprise-class software configuration management (SCM) system that can be used for small teams (IBM® Rational® ClearCase® LT), large enterprises (ClearCase), and globally distributed enterprises (IBM® Rational® ClearCase MultiSite®).

There are many documents and books that describe ClearCase configurations in great detail, this article will not repeat all that information. The following is an introduction to the basics of streams and views, and how these relate to modeling. These concepts and suggestions apply across all versions of ClearCase.

Unified change management

ClearCase is a very powerful versioning system and database. It can be scripted to do almost anything. Over time, though, it became apparent that every major enterprise was creating a similar set of scripts to control baselining and branching. Therefore, the ClearCase team released a built-in facility called unified change management (UCM.)

UCM uses two or more versioned object bases (VOBs), and organizes your artifacts into projects and components. One VOB is called the Project VOB, or pVOB. This is where all metadata is stored, including project, stream, and activity data. Activity data is of particular interest, since it helps you track and define the content for each baseline.

You access and manage these VOBs by implementing a hierarchical set of integration and development streams, along with their related views. A view is your local access to a stream.

Project

A project is the highest-level concept in UCM. It is often used to encompass the shared work of a team. You might choose to create a project for a complete logical model (all artifacts), with several components grouped under the project. You might choose to create one project per team with your organization, but remember that ClearCase LT only allows you to create one high-level project. So when you use LT in this mode, you would create a separate set of VOBs for each team.

A project comes with a default stream under which many other streams can be created.

Stream

A stream is a logical flow of artifacts forward in time. An integration stream is used to contain the master set of artifacts for a particular group. For example, you could store an enterprise set of models in a topmost integration stream, with several line-of-business oriented streams below that. In another scenario, each major team could have their own integration streams, where they
integrate their own components before pushing the components upwards into the enterprise stream for final integration.

Each individual practitioner can have a private development stream that is a mirror of its parent integration stream. This allows you to work in isolation from others until your model changes are complete, and then deliver the changes in one atomic change set from your development stream into the integration stream. UCM provides atomic change sets during delivery, so that you can roll back a change if you encounter significant issues during merges. This allows the integration stream to remain "clean" while you work out issues in the private development stream.

To perform the merges in the development stream, where they can take place with no impact on others, you rebase the development stream from the parent integration stream, normally choosing the latest recommended baseline as set by your project's model manager. This has the effect of compiling all the most recent changes that have been delivered by other practitioners, and that have been accepted and validated by the model manager.

Rebasing
Rebase frequently! This prevents excessive merging, and allows ClearCase to merge models automatically and silently. Rational Software Architect supports very fine-grained conflict resolution, so many changes to the same element can be combined automatically.

This setup might look like that depicted in Figure 1:

**Figure 1. Different project streams**

Enterprise integration stream

Team 1 integration stream

Modeler 1 development stream

Modeler 2 development stream

Team 2 integration stream

Modeler 1 development stream

Modeler 3 development stream

Note that modeler 1 has a private development stream on each of the team integration streams. There is no limit to the number of development streams a modeler may have. If there is something unique to contribute to each team's work, then it makes perfect sense to contribute to each team separately.

**Baseline**

As already discussed in the article's introduction, a baseline is a label across a related set of artifact versions. The intent is that the artifacts can be used as a "matched set" and retain the highest possible level of mutual integrity. That is, all cross-links and relationships are valid at every "official" label. There is a subtle meaning to the word official here: it is quite possible to automate labeling and have one applied to a repository every night or on every build. But only validated
builds – those that have passed their sanity tests for example – should be promoted to any official status. In ClearCase UCM, this is done by recommending the baseline so that others can rebase their workspaces to the new baseline.

**View**

A view is required to see into a stream. It is possible and sometimes desirable to create a view on an integration stream. But it is more common for views to be created on development streams. When creating a development stream, ClearCase always asks to create the view at the same time. Go ahead and let it do that.

Views come in two flavors: dynamic and snapshot. Technically, there is a third flavor, Web view, but these operate like snapshot views so there are only two behavioral classes here.

A dynamic view provides direct access to the stream contents. As the stream content changes, so does the view content. If a file that is already opened in a Rational Software Architect editor suddenly changes, you will be asked if you want to reload the file. Please respond **Yes**, because otherwise you will have files out of synchronization, which is a potential model integrity issue.

Note that this is not the same issue as checking out a newer version in the repository in a private stream. In that case, the stream in intended to remain at a previous baseline level. In the shared dynamic view case (this one), the views are tracking the latest work, and it is necessary to accept every incoming change immediately to remain synchronized.

**Dynamic view**

Dynamic views have several useful properties:

- They are stored on the server, which is presumably backed up. Thus, you may check in your files at night and be assured of their safety. This is only possible when you use private development streams of course, otherwise your work in progress would hit the integration stream every night, which is an unacceptable practice.
- They propagate change immediately. As soon as you check in, all other dynamic views on the stream will see your work. This is very useful when working with source code that builds, as only one person needs to perform a build. For models, it can be equally useful if the model is fragmented into very small pieces and reserved checkouts are enforced. This removes the possibility of file-level conflicts (and the resulting need for merges), except in the rarest of circumstances (for example, a race condition).
- When refactoring models, it is possible to get part way through and be forced to stop and reload the entire model. This happens when someone else changes a needed file, and it suddenly appears in the workspace while the older version is loaded into Rational Software Architect and is about to be operated on. The only choice in this case is to cancel the whole operation and roll back all changes.
• They require no space on the local hard disk. This is very useful when you have very small workstations. Coupled with fragmentation’s reduced memory footprint, it is possible to use quite small machines effectively.

Snapshot view

Some useful properties of snapshot views:

• Snapshot views provide excellent performance. They work as fast as your hard disk and are never impacted by network issues.
• Snapshot views provide the ultimate level of isolation. They are stored on your local machine in a folder on your disk sandbox. You can work on these without feeling the impact of other peoples' changes. While this is not an issue with private development streams, it can be a huge issue when you work directly on an integration stream. Snapshot views allow you to control exactly when you take the latest artifact versions.
• You can take your work with you on your laptop. Dynamic views require a fast connection to the server, while snapshot views require no such connection.

Governance

UCM optionally offers integration with IBM® Rational® ClearQuest®, an industrial-strength problem and activity management tool. It too has multi-site support. When you use these integrations, ClearCase asks you to select an activity whenever an artifact is checked out. This ties all related artifact versions to a specific activity, which in turn is tied to a baseline. All this provides a high degree of visibility to change sets and baselines. ClearQuest provides a large number of built-in queries, and more importantly a powerful query builder feature. Although beyond the scope of this article, you should keep model governance in mind when you set up the tools and environment, and UCM provides a significant set of features oriented to governance through tracking and queries.

Clients

ClearCase offers three clients, each of which has different strengths and weaknesses, and each supports different view types and model types. They are:

• **ClearCase client applications.** These include the ClearCase Explorer, the version tree, the history browser, the ClearTool command line and so on. They are implemented as native applications on Windows and Linux and use the "type manager" interface to perform compare and merge operations. Rational Software Architect has full support for the type manager for Eclipse based models, implementing a protocol that allows ClearCase to communicate with any running instance(s) of Rational Software Architect, Rational Systems Developer or Rational Software Modeler and to choose the best match to handle the request (based on the workspace contents and the included support for model merging). Rational Software Architect is capable of launching an instance of itself when ClearCase calls and there are no running instances. ClearCase client applications all support UCM or base ClearCase with snapshot or dynamic views. Rational Software Architect works with all versions of ClearCase from 5 onwards. Rational Software Architect will warn the individual users if the ClearCase patch level is not adequate to support modeling in a team.
• **ClearCase Software Configuration Management (SCM) plug-in for Eclipse.** This is the original Eclipse client and it works extremely well, supporting all operations on snapshot and dynamic views. It does require the presence of the ClearCase client applications, as it uses shared libraries to communicate with the ClearCase servers. SCM works with all supported versions of the ClearCase server.

• **ClearCase Remote Client (CCRC) plug-in for Eclipse.** This newer plug-in is the future of ClearCase on Eclipse. It supports a special "Web view" type that can interoperate with other streams and views. CCRC works on ClearCase 6 and newer versions only. CCRC supports logical models in Eclipse as of version 7.0.1, thus making it possible to merge fragmented models within base ClearCase. However, CCRC cannot merge logical models during rebase or deliver operations. This has significant implications for partitioning options, as will be discussed in a later section.

**Team or group configuration options**

ClearCase streams and views are almost infinitely configurable. A good place to start, though, is to choose how your modelers will be expected to interact. Will they work together so tightly that even their mistakes are immediately propagated to all the other desktops? Or will they work somewhat isolated from each other and only bring their work together when they deliver to the integration stream?

Once that decision is made, the rest follows in a straightforward manner. The following describes these two as opposite extremes.

Note that each of the following methods starts with you creating a team integration stream. All modelers on the team will contribute to the single integration stream, and the integration stream will eventually be rolled up to a project stream, an enterprise stream, or any other integration stream depending on the chosen stream hierarchy for the enterprise.

**Tightly integrated team**

A small team may choose to interact in real time. If you use this method, it is recommended that you fragment the model as completely as possible. All classes and packages become individual artifacts in the repository. Diagrams are fragmented into their own package as a group or individually. (This overhead is expected to disappear in a future release, when you will be able to fragment diagrams without a wrapping package.)

It is very important that your team use only reserved checkouts for this method. Small context merging can cause difficulties because of the lower level of model integrity protection that is available, so it is important that checkouts be prevented when someone else already has the class or package checked out.

The potential to block others’ access to an artifact is not a significant issue when your team practices strong ownership. With strong ownership, each modeler is responsible for a specific area of the model, and collisions at the artifact level are significantly reduced. When collisions do occur, the modeler simply waits until the artifact becomes available.
A disadvantage of dynamic views is that every change is immediately propagated to all desktops in the group – including bad changes. The method is excellent, though, for teams that have a quality network, and that do not take the models home at night.

Figure 2 shows how one of these teams is configured:

**Figure 2. Configuration for a tight team**

<table>
<thead>
<tr>
<th>Tightly integrated team integration stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeler 1 dynamic view</td>
</tr>
<tr>
<td>Modeler 2 dynamic view</td>
</tr>
<tr>
<td>Modeler 3 dynamic view</td>
</tr>
</tbody>
</table>

**Loosely integrated team**

A loosely integrated team prefers to take advantage of the isolation afforded by private development streams. This method relies on the model manager to verify the quality of deliveries into the integration stream, and to set baselines and recommend these baselines so that the team can update to the latest work as often as possible.

An obvious advantage for snapshot views is the lack of impact in real time from bad changes, and the high performance that is possible when you work from the local disk.

Figure 3 depicts the typical loosely integrated team:

**Figure 3. Configuration for a loose team**

<table>
<thead>
<tr>
<th>Loosely integrated team integration stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeler 1 private development stream</td>
</tr>
<tr>
<td>Modeler 1 dynamic view</td>
</tr>
<tr>
<td>Modeler 2 private development stream</td>
</tr>
<tr>
<td>Modeler 2 snapshot view</td>
</tr>
</tbody>
</table>

Note that, with private streams, the practitioners can use either type of view. The private stream isolates any changes from the integration stream until delivery. Only local policy regarding disk space or the need for disconnected work has any impact on the choice of view types for practitioners.

It would appear that there is nothing to stop each team from choosing its own operating methods. Any one team can theoretically choose to operate as tightly or loosely integrated. The problem, of course, is that the model partitioning strategy that best suits each of these options is different. Classic model partitioning into larger chunks using scoping as suggested in the hub and spoke method described in Part 3 of this series works well for loosely integrated teams. The new fine-grained fragmentation that is available in Rational Software Architect V7.0.0.2 and later works well for tightly integrated teams. So it is a very good idea for these decisions to be made at the highest level of model and project management that would control a set of models in ClearCase.

Of course, local policy could allow the proliferation of these differences, with the attendant difficulties. And they do make it more difficult to follow what is happening.
Roll-up

In a governed, multi-stream environment, each team will eventually roll up its changes to the next level integration stream, which requires a standard UCM delivery. Theoretically, this makes it possible for teams to independently choose their operating mode, as shown in Figure 4:

Figure 4. How teams roll up changes

![Diagram showing team roll-ups](image)

The streams used for teams 1 and 3 imply private development streams that roll up to the integration stream, while team 2's stream implies shared dynamic views on a single integration stream. Each method works best with a different partitioning strategy – reasonably large partitions for the loosely integrated teams and small fragments for the tightly integrated team. Unfortunately, the previously mentioned inability of CCRC to provide merge support for logical models restricts us to one extreme or the other.

It is therefore not possible to mix team operating modes using a common partitioning strategy. With high context partitioning, merging and UCM are the choices. With fragments, shared dynamic views on a single stream (for the whole organization) is appropriate.

Summary of issues with mixed team styles

So what can you expect if you choose to mix these team styles anyway? The two scenarios that have serious issues are:

- Finely-grained model fragmentation can work with a loosely integrated team so long as you practice strong ownership. That is, so long as there are not a large number of conflicts at the artifact level. In cases where there are significant conflicts, each conflict will be processed in the very small context of a single classifier. Since the compare support does not see all of the related conflicting changes at once, no model integrity protection will apply, and the practitioner performing the merge is free to resolve each conflict to the local artifact, the latest checked-in artifact (often called the remote artifact), or the common ancestor artifact. Since each of these is processed individually, it is quite possible to mix resolutions across several related changes, thus creating a new state that never existed in either contributing change set. This is a subtle form of corruption that cannot be prevented when mixing fragmentation with unreserved checkouts and merging. To be clear, the risk of this kind of corruption is very high when using a lot of small fragments with UCM and merging.

- High-context (large) model partitions do not work well with a tightly integrated team if the intent is to avoid merging. Larger artifacts will guarantee collisions in the repository, which will hurt productivity as practitioners wait for access to the model partition. The alternative is to allow unreserved checkouts and accept merges when needed, which does in fact work. The main issue with this, though, is that frequent updates of the larger files may force frequent
reloads on all other desktops, because the larger models are likely to be loaded into the application all the time. No physical separation means that every check-in intrudes on the whole group. This behavior will get old in a hurry.

**Model manager role**

Regardless of which integration style a team chooses, there is the need for a model manager to watch over the model itself. Each level of integration requires a separate model manager role (although obviously a single person could be the model manager for any group of streams, or for the whole enterprise.)

The model manager is responsible for reviewing changes to each integration stream, and for periodically validating and correcting the model. The model manager also sets baselines and recommends them so that they become the default when the next layer below wants to pull in the latest changes.

This can work well within a team hierarchy: each integration point maintains separate baselines, and accumulates changes from child streams at different rates. It is preferred that each team have its own scope, and that they operate on different artifacts (or at least on different parts of the model).

The model manager will typically have either a dynamic or snapshot view on the integration stream. At some point, preferably once per day, the model manager will:

- Lock the stream and update the view
- Open and validate the model
- Correct any errors
- Declare and recommend a baseline
- Unlock the stream
- Send a message to the team that there is a new baseline

The team members should immediately update their private streams and views so that they can resolve any conflicts as soon as possible. The shorter the time between new baselines, the better everything functions. There are fewer merges, because people are working on the latest versions of the artifacts. When merges do happen, then, there are fewer changes to merge, which reduces the likelihood of conflicts at the element level. Therefore, user intervention is rarely required.

**Refactoring**

Refactoring your models (that is, changing the structure of the model, logically and physically; moving things around; or renaming elements and packages) is a necessary evil. Why consider refactoring evil? This is because physical changes to the models’ structure interact with ClearCase and with other modelers’ changes. Refactoring changes are therefore particularly violent with artifacts, and with the repository.

There are several ways in which a team can protect itself from the risks associated with these changes:
Lock the entire integration stream and have one person perform the refactorings on a dynamic view. When the stream is reopened, ensure that all modelers rebase to the latest models. The advantage here is zero interactions with other changes. The disadvantage is that it only works for major refactorings, because minor ones would not normally be performed while the stream is locked. They simply happen too frequently. However, their scale is usually small enough to be much less risky.

Perform all refactorings as part of the normal work. The advantage here is that the stream does not get locked out. The disadvantages depend on the style of partitioning and work flow that you have chosen.

- For private streams, the integration will only occur when all the refactorings are completed successfully. In addition, UCM delivery can always be rolled back should a merge become too difficult. This is in some ways superior to shutting the stream down, as there is no time pressure during the refactorings. You can test the change sets repeatedly against the latest versions by simply rebasing in the latest changes. Furthermore, you can deliver and roll back the models as often as necessary in order to get a clean model in the integration stream before committing. Note, however, that the presence of undelivered structural changes does increase future risk levels, and you can run into great difficulty if two refactorings collide later on.

- For shared dynamic views, things are trickier. If there is refactoring on the live integration stream, all modelers will see the changes in real-time, and the changes may be considerable. If the fragments are already in memory, the other modelers will constantly be refreshing the model. If the fragments are checked out, the modeler performing the refactoring will be stopped, and all changes rolled back forcibly. The practitioner will then have to retry later. This can be quite slow and painful, basically eliminating all the potential advantages of using streams in the first place.

The bottom line is that refactorings are best performed on locked integration streams when fine-grained fragmentation is in use. When large context models are in use, refactorings can be performed within the private stream and merged during rebases until the changes are completed and ready to be delivered. In a hybrid mode, where high context fragments are used, the private stream mechanism is the best approach.

Refactorings should not be left undelivered for a long period of time, as this will increase the risk of eventual collision with other refactorings. Merging many structural changes to the repository with many other structural changes would be a very unpleasant way to spend a day.

How to partition a model

ClearCase's excellent automated merge support makes it possible to use a single model artifact, even when it is shared across multiple teams. Far more important than physical partitioning is the logical partitioning inside the model artifact. Packages should be structured in a logically separate fashion, and they should be as cohesive as possible. Diagrams should be coupled only with the minimum set of shared packages, and never with each other. In Part 3 of this series, the hub-and-spoke logical partitioning method is described. This technique is necessary in order to enable later physical partitioning for smaller memory footprints.
Well-formed models

Spaghetti models are as bad as spaghetti code. Isolate the separate parts of your model from each other to every extent possible. Keep shared elements together, and have diagrams point only to the shared elements. Shared elements may exist in layers, like an onion, with fewer spokes interested in higher layers and more spokes interested in lower layers. All the usual best practices (such as high cohesion and low coupling) are as important during design modeling as they are during implementation. You may refer to any number of excellent books on software architecture to ensure the application of these concepts into your models.

Figure 5 shows a version of the hub-and-spoke partitioning model that looks more like onion layering for shared packages. You can view the two layers between the outer spoke layer and the inner common package layer as shared portions of multiple spokes or, perhaps better, as specialized portions of the hub. The key point is that any variation of these techniques will work very well so long as the references to elements always flow inwards, and never sideways or outwards.

Figure 5. Layering shared packages

Once the logical partitioning is completed, you can begin the physical partitioning with the chosen team workflows in mind. Following are three options for physically partitioning your models.
Classic partitioning by models and packages

Model partitioning is classically performed by carving up the logical model into several stand-alone physical models. Each model can be opened independently, but its internal elements will reference elements in other model files. The hub-and-spoke partitioning method as described in Part 3 of this series points out that the memory footprint can be reduced to the spoke in which the modeler needs to work and any hub models that the spoke model(s) reference. The ideal here is to open a single spoke model and a single shared packages model.

Advantages to the classic model partitioning include:

- Speed is good when you perform repository operations, because the number of files is minimized
- Organization on the disk is very clear, and it is easy to find the file that you want to open
- Merge context is high, which helps protect model integrity

Disadvantages include:

- Transfer times, load times, and other processing times will be slow if the workstation is memory-constricted and the partitions are large
- There is no root model with the complete hierarchy, as Rational Software Architect implements logical models of this type as a mesh

This form of partitioning lends itself particularly well to the UCM workflows with private development streams and merging.

Fine-grained fragmentation

Fragmentation, as introduced in Rational Software Architect V7 and later, creates many very small files that remain hierarchically related and attached. The ideal here is to open only those fragments that you really need by drilling down from the root model. For example, you might open 1 package with a contained diagram and all the packages to the root, then each classifier that is referenced in the diagram and their parent packages to the root.

Advantages to this fine-grained fragmentation include:

- Load times are very fast, because only a tiny root is loaded at first, then each package fragment is opened until the diagram is opened
- Strong ownership can mean that collisions at the file level never happen
- The model's hierarchy remains visible in the project explorer, so there is no architectural ambiguity or confusion

Disadvantages include:

- The context for merges is very small, eliminating any chance to perform model integrity protection
- A file-level collision in shared dynamic views can force you to reload the whole model
• Transfer times can be excessive because of the high overhead from a large number of files.

Fine-grained fragmentation lends itself to the use of shared dynamic views with enforced reserved checkouts. This is essentially the classic database record-locking mechanism. **Note:** In fact, enforced reserved checkouts are "mandatory" in this model because it is far too easy to corrupt a fragmented model by merging many related pieces with no shared context. This eliminates the compare support's model integrity protection mechanisms.

**Coarse-grained fragmentation**

Of course, the two methods can be mixed, and you can break out larger packages or groups of packages as fragments. Advantages include:

- Speed is good when you perform repository operations, because the number of files is minimized
- Organization on the disk is very clear, and it is easy to find the file that you want to open
- The hierarchy remains visible in the project explorer, so there is no architectural ambiguity or confusion
- Merge context is high, which helps protect model integrity

Disadvantages include:

- Transfer times, load times, and other processing times will be slow if the workstation is memory-constricted and the partitions are large
- A file-level collision in shared dynamic views can force you to reload the whole model, and this form of partitioning increases collision risk.

This form of partitioning works best with the UCM workflows with private development streams. Note that, because merging is used when moving from stream to stream, it is important that fragmentation be done in locked streams and not in parallel. This removes the risk of corruption of the model's hierarchy references during merges. Once partitioned, it is perfectly safe to merge individual fragments with reasonably large context. This allows compare support's integrity protection to function appropriately.

**Conclusion**

This introduction to model management with ClearCase has described a couple of best practices. There are many variations on this theme, but in practice it is important to keep things fairly simple. UCM works well and handles most of the heavy lifting of stream management. Shared dynamic views will work for some teams as well.

In summary:

- Choose between: a hierarchical governed model management strategy with private development streams and merging; and a finely-grained fragmentation strategy with a single stream acting as a database-like repository and individual fragments locked completely.
• Use private streams and UCM methods at the team level when you use larger artifacts, either models or fragments
• Use shared dynamic views with enforced reserved checkouts at the team level when you use fine-grained fragments

For a governed model management strategy:
• Structure your packages and diagrams along the hub and spoke model, regardless of which partitioning method you use
• Structure your integration streams into a hierarchy, place all models in the streams, and let each modeler choose which subset(s) to load into Rational Software Architect
  • Use team project sets for scoping
• Use UCM to manage your integration streams
  • Alternatively, you can write your own scripts to mimic the behavior of UCM, but this entails unnecessary development and maintenance cost
  • Another alternative is to use tools like ClearCase's findmerge to simulate UCM-like behaviors, but this is cumbersome for practitioners and model managers alike
• Use activities, integrated with ClearQuest, to provide a significant amount of query capability for model and team governance
• Choose your refactoring methodology carefully based on the analysis above
• Keep your fragments in the same Eclipse project with the main model, as this is the best supported physical architecture in Rational Software Architect V7 and later

Final Words

There are many related articles on ClearCase and model merging. I encourage you to explore these articles and perhaps acquire one of the many books on ClearCase deployment and configuration. Although the up front cost sometimes appears daunting, you will be happy you are using ClearCase and UCM when more and more streams are split off for various versions and releases of the models.

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)