Modularizing a Cairngorm Project - A Debriefing

Posted At : April 27, 2011 8:45 PM | Posted By : Josh
Related Categories: Adobe Flex/AIR

I recently took on the task of a major refactor of an AIR project. The application had grown into a monolithic beast with lots of conceptually related but logically disparate functionality. Management of the code was becoming a challenge. Packages were poorly structured. It was a mess.

The real catalyst for change, though, was the compile time. It had gradually crept up from a few seconds to 30+ seconds. In an effort to return to acceptable levels, I broke out chunks of UI into modules within the main project. I pulled graphics and sounds assets into a separate library SWC. But there was no appreciable effect.

The plan of attack was to break out at least two sections of the application, which are essentially mini-applications on their own. They share only a few common aspects of the main application, such as a common model and server connection manager, and are otherwise self-sufficient. Luckily, the project is built on Cairngorm principles, meaning that these items are already more or less detached from the view components.

What follows is a set of principles for successfully breaking down a large Flex application into separate, manageable module-based projects. These techniques seemed to have worked for me, although of course it is still a work in progress and there is much optimizing left to do. Really, this post is probably more for me as a future reference than anything else, but if it can help someone else, that would be terrific. Your mileage may vary, and if it does, leave me a comment to let me know what you've done differently!

1. **First, you'll need a "common library" project.** There are going to be many classes that will need to be accessible from both the "main" project and the secondary module project. This may include DTOs, some Cairngorm events, utility classes, specialized interfaces for the model and server connection manager singletons, and common building-block-style view components. It will also include an interface for the module itself, as detailed in the next item.

2. **Create an interface for the module.** This should generally consist of getter/setter pairs for the model and the server connection manager interfaces. This will allow the main application to inject these items into the module as soon as the module finishes loading. If there is anything else to inject, this provides an easy way to do it.

Before the next item, I have to mention source code versioning. If you are not already using some kind of tool for this, start now. I cannot emphasize this enough. I use SVN for my repositories, TortoiseSVN for the client. You may want to use a different product, which is just fine. But make no mistake, you are about to make massive changes to your project, and it probably won't go exactly according to plan the first time through. You'll be extremely glad a few hours from now when you need to scrap everything you've done so far, and it's as simple as hitting "Revert" and letting the source code versioning tool work its magic.

- **Move the code!** This is an obvious step and sounds simple, but in a complex application it can be tricky. In my first attempt at the refactor, I was totally daunted and afraid to break the original project. So instead of properly moving the code, I made copies of all the classes I wanted to move and left the originals in place. In retrospect, I'm not sure what my plan was. Suffice it to say, the plan (non-plan?) failed miserably and I had to revert hours worth of work. If you're using Flash Builder as your IDE, take advantage of its refactoring tools. Moving a class to another package, or another project, is as easy as right-clicking the file, selecting "Move...", and following the dialogs. This will maintain referential integrity throughout the process.
- **Don't panic when your code breaks, but fix it before moving on.** When moving dozens of classes around between projects, it's very likely that you will introduce a few compiler errors as you move along. It can be unnerving to suddenly see those little red X's all over your pristine project, but it's part of the process. However, you will want to tidy these up as you go. Flash Builder's refactor tools are tremendously helpful, but if your code contains errors, all bets are off. If you ignore the warnings and proceed, package names and import statements will likely be left unchanged and essentially orphaned, since the IDE will no longer have any idea what they are supposed to point to.

After moving things around, I saw tremendously improved compile times, and the application is now much easier to manage. There are still a number of items I want to revisit, and if you're in the midst of a refactor of your own, you may want to consider the following todos:

1. I think I read somewhere that there is a way to exclude common classes from the compiled SWF of a Flex module. This should help to minimize the size of the module SWF. This in turn will optimize the application in one of two ways. For AIR applications, the executable or .AIR file will be smaller. In the case of a web application, there should be significant performance increases when loading the module, since there is less data to download from the web server.

2. I now have a number of classes that live in common, but perhaps shouldn't. Ideally, I think all Cairngorm events should live in their respective module projects alongside their matching commands. This isn't always the case for me, because I've failed to properly separate all of the logic. This means that sometimes two different modules are dispatching the same event, which breaks encapsulation.

Project planning is an important part of development, but unfortunately it sometimes takes a backseat to coding. If you're stuck with a project that has outgrown itself, modularization after the fact might be a viable way to get back on track.