We’re going to talk about the process we went through re-architecting the SpringSource Tool Suite, what we learned from it and share some insights.
Here's a simple architecture diagram showing what STS looked like when it began in 2007. We have the Spring IDE which is an open source (EPL) IDE for developing web applications using the Spring Framework. Spring IDE builds off of Eclipse Web Tools, with integrations with Mylyn and AJDT and GEF for diagrams. On top of this stack we built the SpringSource Tool Suite, which began as commercial closed source extensions and tools for Spring Framework web development. Product based on Eclipse JEE package.
Architecture Diagram (cont.)

But over the years the scope of the project expanded, and we added several server adapters (today the dm Server tools continues as the tooling component for Eclipse Virgo), a Gradle integration and notably we expanded to Groovy and Grails tooling. All this building upon and building out the same technology stack that we had in place in 2007.

The result of this is that the distribution had grown larger, and some users wanted to use certain components without all the other baggage. Are we supporting them optimally?

Of course since this is Eclipse everything is already plugins and features. So we’ll just rearrange the pieces...
Architecture Diagram evolved

Last year we set out to re-architect the tool suite into something more versatile and we ended up with a structure that looks like this. The purpose was to decouple all of our feature components from Spring IDE and from each other so that they each exist standalone, installed into an Eclipse JEE. The Spring tooling that had been developed closed source would be pushed into the Spring IDE open source project, and in fact all of the components would be open sourced (EPL) and have their own sub-projects on Github. Tying them together would be a thin layer we call eclipse-integrations-commons, consisting of common frameworks and dependencies. Some utilities that we had come to rely on in Spring IDE were pushed down into this level as well.
Architecture Diagram evolved (cont.)

And from the SpringSource Tool Suite, we created 2 new tool suite products. The Spring Tool Suite built around Spring IDE and the Groovy & Grails Tool Suite built around Grails IDE.
Conceptualizing the split across logical features. Figuring out which STS features to push back into Spring IDE, which to drop support for.
Beginning to identify dependencies across component boundaries.
So we set out to work on this re-architecturing. It was a collaborative, iterative process. This involved identifying which components each bundle was going to be moved into, and which bundles were being dropped altogether. Then once we had this separation in place, discovering where our dependencies were crossing component boundaries so that we could refactor them out or push them down into our commons component. But while it might have looked complicated, it was programmers work like we've all done, even if not to this scale. It was after we had torn everything apart and put it back together that the real fun came, just making sure that everything still worked.

There were 3 areas that presented unique challenges that we had to consider along the way; IDs and how their used across extensions and in Java, workspace migration (workspace metadata), and how to provide an upgrade path to existing users.
Extension Points and IDs

• Extension point example
  • Plugin com.sts.config.ui, defines extension point formPages
  • Resulting extension point is referenced as com.sts.config.ui.formPages
  • Rename plugin to org.springide.config.ui and plugin.xml can’t find definition for org.springide.config.ui.formPages

• Same principal for editor IDs, view IDs, builder IDs in plugin.xml
• Fully qualify your extension IDs, then plugin ID prefixes don’t apply

Tuesday, March 26, 13

Just about every single bundle and package that wasn’t already a part of Spring IDE was renamed & refactored. The Eclipse refactoring tools are great; renaming plugins, and packages works well across Java files, manifests and plugin.xmls. But there are some areas where you can’t lean on the tools. When we rename a bundle it can percolate through to parts of your code and plugin.xml which you have to be aware of and catch.

Typically when you create a new extension point definition, you give it a simple name which gets postfixed to the plugin ID to give you the extension point ID. Your plugin.xml will have a reference to the .exsd file. When you refactor your bundle ID, it doesn’t actually touch the .exsd file, so the extension point ID remains the same, but plugin.xml tools will complain that the reference to your .exsd file is broken. You have 2 options: update the .exsd file and break all your extension point implementors (bad), or update your plugin.xml reference to the .exsd file (good).

Even if you aren’t creating extension points of your own, many will be implementing extension like project builders, editors, or views which are given an ID that might be referenced by other extensions at some point. Give them fully qualified IDs to avoid this problem.
This is a snippet from the activator class of a plugin in the midst of being ported from STS to the Spring IDE project. We’ve all seen this pattern before, where we prefix some property with the plugin ID. In this case, we’ve just updated the plugin ID to the Spring IDE namespace, but right away that nature ID and property ID change along with it. I know that I don’t want the project nature ID to change, I don’t know yet if it matters whether that install property changes, and I have no idea yet how that plugin ID constant is being used outside of the activator. So I need to correct that nature to remove the plugin ID reference, and I need review all the other uses of PLUGIN_ID.

If we go back to the formPages extension point from the previous slide, imagine I have some extension loading code that uses this pattern to reference the extension point ID. I’ve gone through all the trouble to maintain consistency in my plugin.xml, but now my extension still isn’t loading because I’ve broken the ID reference on the Java side. So there are a couple of anti-patterns using IDs that we can fall into, and we needed to be rigorous in identifying them, correcting them, and verifying that everything works as expected.
Workspace Migration

- Grails and Gradle project natures changed
- STS 2.8.0 migrated Maven projects from Sonatype m2eclipse tools to Eclipse m2e, updating project natures and classpath container IDs
- Reuse the same mechanism to update Grails, Gradle natures & containers

So we’ve taken care to not break extension points and we’ve verified that the product is functioning correctly. But what about our users and existing projects? Spring IDE project natures didn’t change, but Grails and Gradle natures were updated. We want our users to just pick up the tools and continue to use them on their projects like nothing has changed. But with differing natures, the tools won’t recognize existing projects.

It turns out we had already solved this problem a couple of releases earlier. STS had bundled m2eclipse tools for a long time, and when the Eclipse m2e project reached 1.0 we quickly adopted the new tools. We discovered that there was some configuration required to get existing projects working, so we developed a utility that would detect when a project was configured for the legacy Maven tools, and offered to update the natures & IDs for the Eclipse m2e. Realizing we had this support already in place, we decided to build upon it to migrate Grails and Gradle projects.
Workspace Migration

- Preference locations are tied to plugin ID
  - Workspace preferences at workspace_home/.metadata/plugins/org.eclipse.core.runtime/.settings/my.plugin.id.prefs
  - Project preferences at project_home/.settings/my.plugin.id.prefs
  - More at workspace_home/.metadata/plugins/my.plugin.id/

But we went a step further and extend the tool to migrate preferences too! To do this properly we had to be aware of all the different locations where preferences are stored. These locations are coupled to the plugin ID. For example workspace preferences live under workspace_home/.metadata/plugins/org.eclipse.core.runtime/.settings/my.plugin.id.prefs and project preferences under project_home/.settings/my.plugin.id.prefs and so forth. If the plugin ID has changed then the location where preferences are retrieved from and saved to changes as well! Workspace migration tool copied and renamed preference files to their new locations and updated preference keys within the file.

Workspace migration tool is EPL at https://github.com/SpringSource/eclipse-integration-commons look for legacyconversion packages in org.springsource.ide.eclipse.commons.frameworks.core and ui. It might be useful to somebody out there.
Upgrade Path

- No update from 2.x to 3.0
- Plugin, feature and product IDs had all changed
- m2e-wtp upgrade path from legacy features
- Mylar to Mylyn rebranded in 2007
- Early Mylyn releases installed ‘no-op’ legacy Mylar plugins
- For STS, recommended users start fresh with new STS or GGTS packages

So we’ve taken care of user’s data, but what about their install? How do they get from 2.x to 3.0? Unfortunately we could not support updates. Plugin, feature and product IDs had all changed and we had even removed some bundles and features, so a typical update wasn’t going to be possible or practical.

With Juno SR2, it is possible to provide a p2 update from an old feature/product to the a new one. The m2e-wtp project achieved this, transitioning from the m2eclipse-wtp (JBossTools) components to the Eclipse hosted components. Details available from https://bugs.eclipse.org/bugs/show_bug.cgi?id=394156

What about having the old 2.x tools installed alongside the 3.0 tools? Bad idea. When Mylar project rebranded to Mylyn, the early Mylyn releases installed non-operational legacy Mylar bundles to override the old bundles. So if you had both tools installed, only Mylyn was functional.

A clever solution but we settled on recommending instead that users start fresh with new STS or GGTS packages for 3.0.
Workspace migration
https://github.com/SpringSource/eclipse-integration-commons
Look for legacyconversion packages in *commons.frameworks.core and ui bundles

m2e-wtp migration details
https://bugs.eclipse.org/bugs/show_bug.cgi?id=394156

Twitter
Leo Dos Santos, @leods
Martin Lippert, @martinlippert

Workspace migration tool is EPL at https://github.com/SpringSource/eclipse-integration-commons look for legacyconversion packages in org.springsource.ide.eclipse.commons.frameworks.core and ui. It might be useful to somebody out there.
Give Feedback on the Sessions

1. Sign In: www.eclipsecon.org
2. Select Session Evaluate
3. Vote: +1 0 -1