Feature relationships - a review

Leif Frenzel
leiffrenzel@googlemail.com

February 2007

Abstract

This is an informal paper about some of the more arcane details of Eclipse technology, related to its update/install story. The purpose of this is to collect some thoughts that I have posted to mailing lists and points I made in technical discussions, mainly to have them all in one place. There is an ongoing discussion at the Eclipse project how the approach discussed here (the approach based on so-called *features*) might be replaced in future versions of Eclipse, so hopefully this is somewhat useful for summarizing a few lessons learned. I'm confident that the problems described here will soon be a thing of the past.

Contents

1 Feature relationships **1**
1.1 Dependency 2
1.2 Includes 4
1.3 Mixed dependency and include 5

2 Problems caused by feature includes **5**
2.1 Asymmetry of install and uninstall operations 5
2.2 Direct reference to versions in includes 7
2.3 Hidden dependency cycles 7
2.4 The meaning of terms 8
2.5 The right granularity 9

3 Conclusion **11**

1 Feature relationships

The update/install story in Eclipse distinguishes between *plugins* (functional units) and *features* (deployable units). A feature is not much more than a lightweight descriptor that basically states a list of plugins. To install a feature means to copy the software from the plugins enumerated in that list into the

Eclipse installation and register these plugins so that they can be run as part of that Eclipse installation. The feature is then no longer needed (except for uninstalling).

Typically, both features and plugins reside on an update server, but the plugins themselves need only be downloaded when an actual installation is requested. For preliminary tasks (such as determining whether dependencies are fulfilled, or what the file size of the required download would be), only the feature must be accessed, which is much smaller (this distinction is sometimes referred to as one between a 'meta data server' and a 'byte server').

However, this means that all information necessary for validating an installation attempt must be available from features. In particular, the dependencies to other features of a given feature must be known from just reading the feature contents, without examining the plugin contents (which would require the plugin to be downloaded). Thus, all dependency and other feature-to-feature relationships must be knowable from what is stated in the feature manifest.

This paper gives an overview over the different relationships that can obtain between features, and exposes some problems in the logical system that emerges from these relationships. I shall also note some problems that have arisen in the real-world use of the features approach.

To have a shorthand notation, let us refer to features using uppercase letters, and to plugins using lowercase letters. To express that a feature **F** provides the plugins **p**, **q** and **r**, I shall write

```
F[p,q,r]    -- feature F has p, q and r in the list
            -- of provided plugins
```

(I mostly ignore the implications of different plugin and feature versions.) I will introduce more shorthand notation for some of the notions introduced further down.

1.1 Dependency

Dependency is the most straightforward relationship between features, not just because it reflects well-known technical facts about the software provided by these features, but also because users have an understanding of this relation without necessarily knowing about the implementation details that let the dependency emerge. It is easy to understand for a user that she can install some features only if some given other features are installed too, and it is equally obvious that some features can only be uninstalled if some others are removed with them.

Let us use the following notation for the dependency relationship:

```
A -> B      -- feature A depends on feature B
```

All dependencies (and in general, all feature-to-feature relationships) must be declared in the `feature.xml`, the manifest file that is contained in a feature

© 2007 by Leif Frenzel. All rights reserved.

archive. There are two different ways in which a dependency of a feature on another feature can be expressed.

The first way is to directly declare the dependency in **A**. The code snippet from the `feature.xml` in **A** would look like this:

```
<feature id ="A"
  ...
  <requires>
    <import feature="B" />
  </requires>
  ...
</feature>
```

The second way would be to declare a dependency from **A** to a plugin, say **p**. This would not mention **B**, but since **B** happens to provide **p** (which is declared in the feature manifest of **B**), the dependency from **A** to **B** would still hold.

Note that the second way has an advantage over the first one: it is no longer necessary that someone looks at the features and decides that **B** has to be a dependency of **A**. Therefore, there is no danger that the declaration gets out of sync - if **B** no longer provides **p**, but another feature, **C** provides it instead, there is no manual work needed to update the dependency declaration, and everything is still correct.

There is also a disadvantage, however: the Update Manager now needs to download all features in order to determine which of them provides **p** (and if there is none to be found on the update site, that's even in vain and it has to issue an error).

The `feature.xml` code looks like this:

```
<feature id ="A"
  ...
  <requires>
    <import plugin="p" />
  </requires>
  ...
</feature>
```

How does a feature, namely **A**, know to which of the plugins it has to declare the dependencies?

Dependencies are in the end always plugin-to-plugin dependencies. (Remember, plugins are functional units, but features are only deployment descriptors, they are no longer needed once the plugins have been installed - in particular, they are not needed when running an Eclipse installation.) So the statement that feature **A** depends on plugin **p** is really just short for: At least one of the plugins provided by **A** requires plugin **p**. [1] The set of required plugins

[1] Note how this gets interesting when there is more than one plugin provided by **A** that requires plugin **p**, and the dependencies are specified by giving different version ranges. There is also a certain fuzzyness about cases where a plugin is provided by multiple features.

© 2007 by Leif Frenzel. All rights reserved.

for a feature **A** is therefore the union of all plugin requirements of the plugins provided by **A**. (It's a list only because it is written in sequence in the manifest file, but nothing depends on the order here, so from a logical point of view, it can be seen as a set.) Thus, if

```
A[a,b,c]
```

and **a** and **b** require a plugin **p**, whereas **c** requires **q**, and both **p** and **q** happen to be provided by a feature **B**, then

```
A[a,b,c] -> B[p,q]
```

The Plugin Development Environment (PDE) in Eclipse, which can be used to create features, determines the dependencies that must be written into a feature automatically from the dependencies in the provided plugins. (Unfortunately it does that not always automatically, which means that there are many features out there which declare an incomplete list of dependencies.)

1.2 Includes

A feature can *include* another feature. In the `feature.xml`, such an inclusion relationship (sometimes called 'nested features') looks like this:

```
<feature id ="A"
  ...
  <includes id="B" version="1.2.3" />
  ...
</feature>
```

Let's write this as

```
A{B} -- feature A includes feature B
```

There can be arbitrarily nested features, i.e.

```
A{B{C}}
```

is fine.

A feature inclusion can be seen as an alternative way to specify that a plugin is provided (one could say: it is in this case indirectly provided). This means that

```
A[a,b] {B[c,d]}
```

is roughly equivalent to

```
A[a,b,c,d]
```

with the difference of course that there is a certain 'ownership' information (**B** provides **c** and **d**) which is lost in the second formulation. However, if the Eclipse update mechanism encounters a feature that includes another feature, the included feature is silently installed. The update mechanism therefore treats included features indeed as if the inclusion were equivalent to direct providing, i.e. it assimilates the first formulation above to the second one.

© 2007 by Leif Frenzel. All rights reserved.

1.3 Mixed dependency and include

The dependency and inclusion relationships can (and actually do) obtain in any mixture in the real world. A feature included by another feature may depend on a third feature:

```
A{B -> C}
```

Or a feature that includes another feature may itself depend on a third feature, although the included feature doesn't:

```
A{B} -> C
```

There may be a dependency from a feature to another feature that includes a third one:

```
C -> A{B}
```

or to a feature that itself is included by a third one:

```
C -> B, A{B}
```

A mix of dependency and inclusion relations between features, in addition to the opacity that stems from the fact that some (actually, most) dependencies are just re-declared dependencies from provided plugins, can make the actual connections within a given set of features quickly too difficult to clarify without using an analysis tool.

What is worse, though: it introduces a number of potential installation problems that are hard to understand for an end user. These are discussed in the following section.

2 Problems caused by feature includes

2.1 Asymmetry of install and uninstall operations

As mentioned above, the Eclipse Update Manager silently installs and uninstalls features that are included by other features. A problem with this approach is that it makes the installation and de-installation operation asymmetric, which is very irritating for users.

Here are two different example cases.

Suppose there is a feature **J**, which is included by a feature **V**. Let's also assume that there is a feature **W** which depends on **J** (but not on **V**).[2]

```
V {J}, W -> J
```

[2]This was actually one of the first examples of project interdependencies at Eclipse.org. If you substitute the Eclipse WebTools for **W**, the Eclipse Visual Editor for **V**, and the Java EMF Model (JEM) for **J**, you get the exact real-world equivalent of this example. (The letters are chosen deliberately to reflect this, of course.)

© 2007 by Leif Frenzel. All rights reserved.

Now suppose a user has installed **W** successfully. Since the installation is valid and **W** depends on **J**, we can assume that this installation contains at least these two features, i.e. **W** and **J**, together with probably some others. Suppose further the user wants, after a while, to give **V** a try. So she installs **V** (successfully). After playing around with it a little, she decides that she doesn't really need **V** after all, and tries to un-install it, thus reverting the installation to its previous state. But the Update Manager refuses to do so. It issues an error that this operation "would invalidate the configuration". This is surprising: adding a single feature to a valid installation and then removing it, without modifying the installation otherwise, should be possible; and it should lead to the exact situation as before the operations were performed. It is not at all plausible for a user that the configuration after uninstalling the feature **V** should be any different from what it was before installing it - and it was valid then.

What happened is this: since the installation already contained **J** when **V** was installed, there was no need to silently install **J** (but that would have happened if **J** wouldn't have been already there). After the installation of **V**, however, an operation that involved **V** would automatically also affect **J**, which was considered to be included by **V**. When the uninstallation was to happen, the Update Manager deemed it necessary to remove both **V** and the included **J**, instead of just removing **V** and thereby restoring the previous state. But **J** could not be removed because there was a dependency on it (outgoing from **W**). Thus the Update Manager concluded that this particular uninstall operation would not be permissible.[3]

The second case is somewhat related to the first. Suppose there is a feature **A** which includes a feature **B**; and a user has installed the latter one, making some use of its functionality standalone. In order to try it out, he installs **A** too. A while later he wants to uninstall **A** again. But, as he will be thrilled to find out, he cannot just uninstall **A**, thereby restoring the previous state. Since **A** includes **B**, after the uninstall operation **B** will be gone too. (It's not just that this will happen as if per accident or so - there is nothing that the user can do; it is not possible to uninstall **A** and retain **B** - unless of course **A** (including **B**) is first uninstalled and then **B** is indeed re-installed.)

Note that the installability and un-installability of a feature is therefore subject to conditions that cannot be controlled or even detected by just looking at that feature alone. It is enough that an arbitrary other feature happens to be added to an installation and declares an inclusion relationship - this makes the former feature practically uninstallable without anything it could do about it. Conversely, a feature might be subject to an inclusion relationship and just be uninstalled by virtue of that relationship, again without anything it can do about it. (Suppose I supply a fancy J2EE IDE for download and just for fun declare to include JDT - you install my stuff, you uninstall it, and whoops! - you

[3]Note that there is a difference between this install-and-uninstall use case and a use case 'revert to previous configuration'. The former is described here as a sequence of operations that follow each other directly; but nothing depends on this: there might have been other changes to the installation between them which are not to be 'reverted'.

© 2007 by Leif Frenzel. All rights reserved.

have an IDE without the Java Development tools ... Actually this won't happen, since - guess what - JDT includes the Platform, and the Update Manager won't uninstall that. But you get the point.)

2.2 Direct reference to versions in includes

Another difficulty stems from the fact that an inclusion must be specified by referring to a feature version. There is no such thing as a feature including another feature *as such* - it can only include a specific version of that other feature.[4]

But now suppose you have an installation with a feature **A** and a feature **B**, where **A** includes **B**. Suppose further both features have version 1.0.0. When you have made some changes to the plugins in feature **B**, you might want to increase its version to, say, 2.0.0, and update it. But that's not possible. An installation with **A** in version 1.0.0 (which must also have **B**, and in version 1.0.0), cannot be updated to contain **B** in version 2.0.0. The Update Manager refuses to do so. Thus your only way to update feature **B** is to also increase the version of **A** and have that second version of **A** include **B** in version 2.0.0.

Sounds not a big deal? Perhaps, but remember from the discussion above that any feature may declare an inclusion relationship to **B**, in version 1.0.0 of course. You might not even provide feature **A**, or any other feature at all, yourself. Still, if the user has your feature, **B**, and some other feature declares to include it, in its version 1.0.0, the user won't be able to update to your version 2.0.0 any more.

2.3 Hidden dependency cycles

Since the feature relationships discussed so far are directed binary relations, there is a possibility to have cycles. The most simple case is a pure dependency cycle.

```
F -> G -> H -> F
```

The upshot of a dependency cycle is of course that all features involved in a cycle cannot be installed (or uninstalled) separately - they must always be added (or removed) as a group. (One cannot select any combination of the features without ending up selecting them all, after dependencies are resolved.) This makes their splitting up into different features pretty pointless, but apart from that it is harmless.

Similarly to dependency cycles, there may be pure include cycles.

```
F {G {H {F}}}
```

[4]Often enough feature authors try to declare a version range in an `include` statement in their `feature.xml`s, using a `match` attribute (e.g. `match="compatible"`). But that attribute is only valid in dependency declarations, not in inclusion declarations.

© 2007 by Leif Frenzel. All rights reserved.

Unlike a dependency cycle, an include cycle is not accepted by the Update Manager. Features that are involved in such a cycle cannot be installed at all.

The possibilities for cycles are not restricted to pure dependency or inclusion cycles. Imagine the following situation involving four features **F** to **I**, each of which provides a plugin (**a** to **d**):

```
F[a], G[b], H[c], I[d]
```

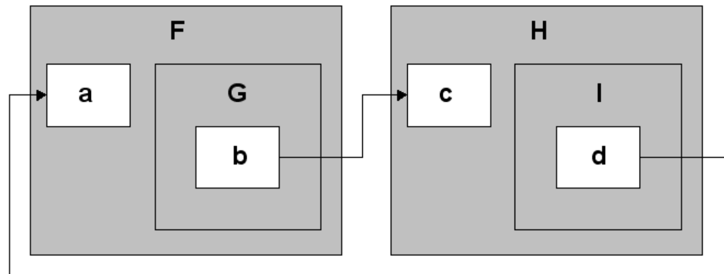
where **d** requires **a** and **b** requires **c**. These plugin relationships result in feature **G** depending on feature **H**, and **I** depending on **F**:

```
G[b] -> H[c], I[d] -> F[a]
```

Suppose further that **F** includes **G** and **H** includes **I**. This is a full cycle; no feature can be selected for installation without forcing all three others to be selected with it.[5]

```
F[a] {G[b] -> H[c] {I[d] -> F[a]}}
```

An illustration might help to disentangle this complicated case:



2.4 The meaning of terms

It is sometimes said that the inclusion metaphor is meant to the effect that included features should be invisible at all to the end user - they are just a sort of 'implementation detail' that is meant to be encapsulated by the including feature and thus of no interest. This is consistent with the behaviour of the Update Manager and its underlying mechanics of silently installing and uninstalling included features together with their includers.

However, it is not consistent with the actual practice in most projects at Eclipse.org (and, following them, projects outside Eclipse.org, too). Consider the Eclipse SDK itself. The Eclipse SDK feature (`org.eclipse.sdk`), at the

[5] Suppose **F** is selected first, then **G** is also selected because it is included, **H** is selected because it is depended on by **G**, and **I** is selected because it is included by **H**. Suppose **G** is selected first, then **H** must be selected because **G** depends on it, **I** is included by **H**, and **F** is selected as dependency of **I**. Similarly for selecting **H** first or **I** first.

© 2007 by Leif Frenzel. All rights reserved.

time of writing, as of version 3.2.2, includes among others the Java Development Tools feature (`org.eclipse.jdt`) and the Eclipse Platform feature (`org.eclipse.platform`), both of which may well be installed as top-level features in their own right. For instance, users who want a Java IDE and don't want to do Eclipse plugin development will typically download the Java IDE runtime download from Eclipse's download server, which contains just the Platform and JDT features (without sources and PDE, the Plugin Development Environment). Similarly, people might be interested in just having the Eclipse Platform feature installed, presumably in order to use some development tools for other languages than Java on top of them. A popular combination would be the Eclipse Platform plus the Eclipse C/C++ Development Tools (CDT). Far from being an encapsulated 'implementation detail' here, the Platform feature is arguably more a piece of software in its own right than the Eclipse SDK feature is. (The main *raison d'etre* of the SDK feature seems rather to be to bundle the RCP, Platform, JDT and PDE features together with their sources, thus being a mere wrapper.) Very similar patterns of usage have arisen at many other Eclipse.org projects, such as the Test and Performance Tools Platform Project (TPTP), or the Business Intelligence and Reporting Tools Project (BIRT).

This brings out an important difference. A feature, as a deployable unit, can be seen from two different points of view: that of an end user, who wants, after all, to use the functionalities it provides and installs or uninstalls accordingly, and that of the feature provider, who wants to split up functionality in a way so that commonly used code can be installed separately - and this means not only code that directly has some end user functionality, but also re-usable components, libraries, frameworks and so on.

2.5 The right granularity

When it comes to the decision what the granularity of features from a project should be, there is a conflict between optimizing code re-use and keeping things user-friendly. Providing a piece of software in an own feature makes it installable separately. From a developer's perspective, it makes sense to break up software into small bits that can be combined freely. (After all, its modularity is a big strength of the Eclipse Platform technology.) From a user's perspective, however, features are the fundamental unit to deal with, and it is therefore important that each feature can be understood by a user. Since features are what users choose from in the Update Manager UI, it should be clear what functionality they provide, and how they interrelate with other features. Dividing software into too small units is not of much help then.

Here is an example of what has actually happened in the past two years. The Web Standard Tools (WST), which make up part of the Eclipse WebTools Platform (WTP), have several different components, such as

```
org.eclipse.wst.common
org.eclipse.wst.server
org.eclipse.wst.web
```

© 2007 by Leif Frenzel. All rights reserved.

```
org.eclipse.wst.ws
org.eclipse.wst.xml
```

These are again divided up, as it is a best practice in Eclipse development, into headless and UI layers, typically called 'core' and 'ui'.

```
org.eclipse.wst.xml_core
org.eclipse.wst.xml_ui
```

and so on, for each of them (I continue the example just with the xml component here). Therefore, we see primarily two features in each component. They have in turn an associated source feature:

```
org.eclipse.wst.xml_core
org.eclipse.wst.xml_core.source
org.eclipse.wst.xml_ui
org.eclipse.wst.xml_ui.source
```

And each of these (including the source features) have associated features that contain internationalized texts, the so-called 'Language Packs'. Language packs come in four different flavours: NLS1 (German, Spanish, French, Italian, Japanese, Korean, Brazilian Portuguese, Traditional Chinese and Simplified Chinese), NLS2 (Czech, Hungarian, Polish and Russian), NLS2a (Danish, Dutch, Finnish, Greek, Norwegian, Portuguese, Swedish and Turkish) and NLSBidi (Arabic and Hebrew). This means that we end up, for just the XML stuff from the WebTools project, with a whopping 20 features:

```
org.eclipse.wst.xml_core
org.eclipse.wst.xml_core.nls1
org.eclipse.wst.xml_core.nls2
org.eclipse.wst.xml_core.nls2a
org.eclipse.wst.xml_core.nlsBidi
org.eclipse.wst.xml_core.source
org.eclipse.wst.xml_core.source.nls1
org.eclipse.wst.xml_core.source.nls2
org.eclipse.wst.xml_core.source.nls2a
org.eclipse.wst.xml_core.source.nlsBidi
org.eclipse.wst.xml_ui
org.eclipse.wst.xml_ui.nls1
org.eclipse.wst.xml_ui.nls2
org.eclipse.wst.xml_ui.nls2a
org.eclipse.wst.xml_ui.nlsBidi
org.eclipse.wst.xml_ui.source.nls1
org.eclipse.wst.xml_ui.source.nls2
org.eclipse.wst.xml_ui.source.nls2a
org.eclipse.wst.xml_ui.source.nlsBidi
org.eclipse.wst.xml_ui.source
```

© 2007 by Leif Frenzel. All rights reserved.

And we have still left out the user documentation and ISV documentation features here, also 'SDK' features (which are used to group software features and source features), examples and so on. Even if users do understand all this (which is dubious), they will have a tedious job to select all they need. (There is one common user reaction to situations of that sort - just install it all.)

This means that, as far as packaging of software is concerned, the structuring that makes sense from the engineering side has superseeded the one that is more right-grained for users, consequently either frustrating them or forcing them to use the nuclear option of 'just-get-it-all'.

There are also scale problems for the Update Manager technology itself implied. As Update sites get larger, the manifest data (essentially, the feature manifests, which are contained in feature archives, one for each feature) that the Update Manager has to download and analyze for inter-feature relationships (such as dependency and inclusion) grow bigger in number and more difficult to handle. Large Update sites with hundreds of features make the Update Manager user interface sluggish and unwieldy.

3 Conclusion

For future designs of an update/install story, features, and in particular certain relationships between them, such as feature includes, don't seem to be a promising model. Moreover, there seems not to be a real need for feature includes - a single, simple dependency relationship can do all that is needed.

A special effort should be made to ensure that the point of view of the end user is taken seriously. It is the end user who who selects, installs and uninstalls features or whatever the unit of deployment is called. The granularity of features should be made to accomodate that perspective, not that of feature providers, that is, developers. If this leads to more complexity for developers, it will always be possible to address it by supplying better tool support. The same applies to the way the install and uninstall operations behave. They should lead to the results that can reasonably expected, not to suddenly 'invalid configurations' where it is just not clear what might have gone wrong. Equally inacceptable are mechanisms that have the effect that uninstalling one feature removes other features from an installation without the user having intended this.

© 2007 by Leif Frenzel. All rights reserved.