The Science of Packages

by Benny Bottema - Friday, March 21, 2008

http://www.bennybottema.com/2008/03/21/the-science-of-packages/

Some time ago I came across a very interesting article by Robert C. Martin (1996 column in pdf) about how packages are supposed to be made up and how it helps building a release, debug and shared-workload strategy. I don't think it all applies to modern package management, especially for web applications, but it is an interesting perspective that deserves some attention.

granularity.pdf (original source offline: www.objectmentor.com/publications/granularity.pdf)

My summary of his article (as I understood all of it):

<summary>

- Packages should have a one-way dependency graph. A cyclic dependency leads to having to import code from other packages you don't always have to use. If an error is made in one of those packages, your code breaks. This can be avoided by removing all cyclic references; in the article he states two interesting solutions (of which one is applying the dependency inversion principle, which is commonly used with classes regardless of any package).
- Packages are release-units. With version control, you can work on a package level that way. Developers or teams are distributed to work on the packages rather than distributed classes. This is also an important reason why not to have cyclic dependencies in packages.According to The Reuse/Release Equivalence Principle Packages are only allowed to (re)use released packages. This is again conform version control principle. This way developers/teams can decide for themselves when they are ready to use a new release knowing it won't change after they started using it.
- The Common Reuse Principle states that classes that are likely to be used together should be packaged together. An example Robert mentions is a "container and its iterators". These classes are reused together because they are so tightly coupled and therefore should be packaged together. Again, this seem to be in favor of the version control principle of to affect other packages as least as possible.
- Classes should be packed together according to the The Common Closure Principle. Which states that classes that are likely to change at the same time because of their high dependency-rate should be packaged together.
  Note:
  In OOP there is a class-level principle called the The Open Closed Principle*, which states that classes should be closed to change but open to extension. This advocates that you should design a class in such a way that you can change its behavior by extending it rather than editing the source itself. The Common Closure Principle makes sure that when you ever do need to edit a class, any other class that is likely to be changed according to that should be packaged together. Again, this is to improve on package-level version control where you want to minimize the effect on other*

packages when changing a class in a package.

Also, Robert mentions packages are designed bottom up, meaning you can’t know or approximate packages forehand; packages are *not* indicative of responsibilities or relationships within an application, which is why you can’t design them forehand. They are a building ‘map’ as he calls it to map out how classes should be grouped together in such a way that changes within packages have minimum impact on other packages. Indeed the UML modeling language gives no more information about packages than that (as of UML 0.9, but I don’t think this changed since). This also means packages grow as classes are added. ‘If you do try to design a package diagram forehand without knowledge about the classes, you will likely fail’ as Robert states and end up in badly maintainable classes, lots of “morning after syndromes” and cyclic dependencies.

</summary>

General impression

I had never thought of packages in such a ‘considerate’ way, but it mostly makes sense to me. I guess that’s because some of these things go natural and you do them without rationalizing everything. People generally seem to design packages pretty neatly (in small applications at least) and I think that’s because there’s an intuitive side to it; You just smack together the classes that you somehow feel belong together. That’s why the *Common Reuse Principle* is probably the most widely used principle, because people subconsciously realize that classes that seem to ‘belong together in a class’ often are classes that adhere to this principle: they are likely to affect each other when changing them.

All in all I think this view rings true, which is confirmed by the fact that you can ‘design’ a package layout in UML. Robert says if you try to design a package without knowledge about the classes, you’ll fail. Obviously if you design an application you get some understanding of its structure and therefor knowledge of the classes. In UML I imagine you do the package diagram just after you did the class diagram, without having to do a bit of coding at all.

There are many guidelines and motivations ofcourse as to how and why package things a certain way, but Robert seems to be on the right track when considering large scale applications, and then desktop applications more so. If you take his advice as a guideline, you could do worse so to speak.