## When and How to Draw Diagrams

Drawing UML diagrams can be a very useful activity. It can also be a horrible waste of time. A decision to use UML can be a very good thing, or it can be a very bad thing. It depends upon how, and how much, you choose to use it.

### When to draw diagrams and when to stop.

Don't make a rule that everything must be diagrammed. Such rules are worse than useless. Enormous amounts of project time and energy can be wasted in pursuit of diagrams that no one will ever read.

*When to draw diagrams:*

- Draw diagrams when several people need to understand the structure of a particular part of the design because they are all going to be working on it simultaneously. Stop when everyone agrees that they understand.
- Draw diagrams when two or more people disagree on how a particular element should be designed, and you want team consensus. Put the discussion into a time box, then choose a means for deciding, such as a vote or an impartial judge. Stop at the end of the time box, or when the decision can be made. Then erase the diagram.
- Draw diagrams when you just want to play with a design idea, and the diagrams can help you think it through. Stop when you've gotten to the point that you can finish your thinking in code. Discard the diagrams.
- Draw diagrams when you need to explain the structure of some part of the code to someone else, or to yourself. Stop when the explanation would be better done by looking at code.
- Draw diagrams when it's close to the end of the project and your customer has requested them as part of a documentation stream for others.

*When **not** to draw diagrams:*

- Don't draw diagrams because the process tells you to.
- Don't draw diagrams because you feel guilty not drawing them or because you think that's what good designers do. Good designers write code and draw diagrams only when necessary.
- Don't draw diagrams to create comprehensive documentation of the design phase prior to coding. Such documents are almost never worth anything and consume immense amounts of time.
- Don't draw diagrams for other people to code. True software architects participate in the coding of their designs, so that they can be seen to lie in the bed they have made.

### CASE tools

UML CASE tools can be beneficial, but they can also be expensive dust collectors. Be *very* careful about making a decision to purchase and deploy a UML CASE tool.

- ***Don't UML CASE tools make it easier to draw diagrams?***
  No, they make it significantly harder. There is a long learning curve to get proficient, and even then the tools are more cumbersome than white boards. White boards are very easy to use. Developers are usually already familiar with them. If not, there is virtually no learning curve.
- ***Don't UML CASE tools make it easier for large teams to collaborate on diagrams?***
  In some cases. However, the vast majority of developer and development projects do not need to be producing diagrams in such quantities and complexities that they require an automated collaborative system to coordinate their activities. In any case, the best time to purchase a system to coordinate the preparation of UML diagrams is when a manual system has first been put in place, is starting to show the strain, and there is no other choice but to automate.
- ***Don't UML CASE tools make it easier to generate code?***
  The sum total effort involved in creating the diagrams, generating the code, and then using the generated code is not likely to be less than the cost of just writing the code in the first place. If there is a gain, it is not an order of magnitude, or even a factor of two. Developers know how to edit text files and use IDEs. Generating code from diagrams may sound like a good idea, but I strongly urge you to measure the productivity increase before you spend a lot of money.
- ***What about these CASE tools that are also IDEs and show the code and diagrams together?***
  These tools are definitely cool. However, I don't think the constant presence of UML is important. The fact that the diagram changes as I modify the code, or that the code changes as I modify the diagram, does not really help me much. Frankly, I'd rather buy an IDE that has put its effort into figuring out how to help me manipulate my programs rather than my diagrams. Again, measure productivity improvement before making a huge monetary commitment.

In short, look before you leap, and look very hard. There *may* be a benefit to outfitting your team with an expensive CASE tool, but verify that benefit with your own experiments before buying something that could very well turn into shelf ware.

**But what about documentation?**

Good documentation is essential to any project. Without it the team will get lost in a sea of code. On the other hand, too much documentation of the wrong kind is worse because then you have all this distracting and misleading paper, and you still have the sea of code.


Documentation must be created, but it must be created prudently. Often the choice *not* to document is just as important as the choice to document. A complex communication protocol needs to be documented. A complex relational schema needs to be documented. A complex reusable framework needs to be documented.

However, none of these things need a hundred pages of UML. Software documentation should be *short and to the point*. The value of a software document is inversely proportional to its size.

For a project team of 12 people working on a project of a million lines of Java, I would have a total of 25 to 200 pages of persistent documentation, with my preference being for the smaller. These documents would include UML diagrams of the high-level structure of the important modules, ER diagrams of the relational schema, a page or two about how to build the system, testing instructions, source code control instructions, and so forth.

I would put this documentation into a wiki[1] or some collaborative authoring tool so that anyone on the team can have access to it on their screens and search it, and anyone can change it as need be.

[1] A web based collaborative document authoring tool. See: http://c2.com and http://fitnesse.org

It takes a lot of work to make a document small, but that work is worth it. People will read small documents. They won't read 1,000-page tomes.

### And Javadocs™?

Javadocs are excellent tools. Create them. But keep them small and focused. Those that describe functions that others will use should be written with care, and should contain enough information to help the user understand. Javadocs that describe private utility functions or methods that aren't for wide distribution can be much smaller.