PROBLEM? WHAT PROBLEM?

by Kris Oosting,

There are problems everywhere! We encounter them during software development and even beyond. We use methods to tackle these problems—to find solutions.

In our experience in the software development world, there are a lot of ways of going from "the problem" to "the solution." But most of the time, developers try to deal with all the details right away.

Here are some observations we made working with Fusion projects developing all kind of systems.

For simplicity, lets say there are four views of the problem:

- *Problem Definition*: a clear definition of what the problem really is in its frame of reference.
- *Problem Specification*: specifying the problem in a way that software developers can work with it.
- *Problem Solution*: the creative phase—finding a solution to the specified problem!
- *Problem Implementation*: translating the designs into something a computer can handle.

Problem Definition, or the requirements phase, defines the problem (or the challenge) as well as possible. There are modeling techniques that can be used to do this, though they will not be discussed here. I am however very pleased that Edward Swanstrom of Arthur D. Little (Fusion Newsletter, April 1996) uses Mind Mapping as a technique to capture candidate classes and relationships. We use the same technique, and also use it to check the requirements for redundancy, consistency, and completeness. During our courses we always get positive feedback that this technique is very useful. It is true that, as a nice result, a Mind Map "gives" you the candidate classes and relationships.

This fits into how we use Fusion quite well. Fusion provides a "to-do list" and uses the concept of "work-preparation." Every technique is preparing the work for the next modeling step. For example, the object interaction graph is preparing the work for the visibility graph and the code. If you do the object interaction graph well, coding is peanuts. So one technique is "transferring" something to the another.

A "To-do list"? Yes. What is the list of System Operations? It's a big to-do list of the amount of functionality you have to design and implement or, better yet, deliver to the user. We noticed that a lot of developers use the list of system operations in this way. They also use it for the famous "percentage done" estimate.

Now back to requirements. It's very important that these are complete and well-defined. Without a good definition of the problem, it's very difficult to build a system that meets the users needs—even with a good method.

We still see a lot of defects being created due to the fact that the problem was not described well enough.

Problem Specification, or the analysis phase, specifies the problem in a way that software developers can do something with it. The challenge here is to find the right object models.

We discovered that when you let people construct an object model from the same problem definition, they find different ways of specifying it. How can you experience this? Well, capture the list of candidate classes from the requirements and put one (important) class in the middle of a piece of paper. Let people then create the object model around it. Every person creates his/her own model.

The models are different, because every person has different experience, providing another frame of reference to the problem domain. This is logical, because an object model looks very much like a Mind Map, and similar processes are going on in one's brain. It is making associations.

To get all of the frames of reference aligned, a lot of discussion is necessary during the Problem Specification phase. Mind Mapping can help to create an overlapping "frame of reference."

We also "measured" that communication is very intensive during this phase. Creativity is lower, as shown in the Communication-Creativity curves in Figure 4.

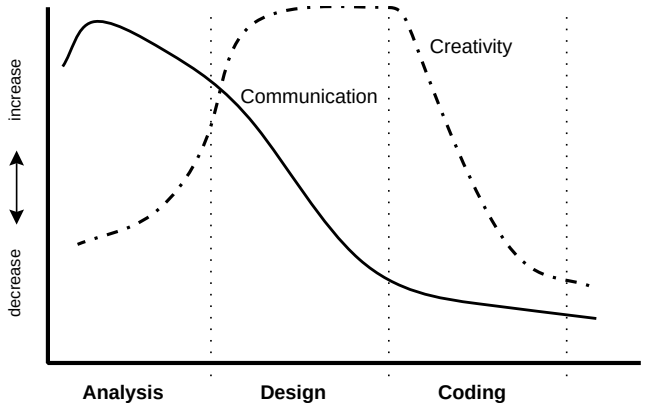

Figure 4 Communication-Creativity curves

What developers are doing is creating associations in their brains, and then trying to convince each other that their model is the best "solution" (you have probably encountered this as well ;-). But, as the first Fusion book (Coleman et al, 1994) already mentioned, Object Modeling is not an exact science. It is specifying the problem definition as well as possible so it is useful for software development.

System Operations are the contractual bit. That's where you specify the conditions—very useful.

Problem Solution, or the design phase, delivers a solution to the specified problem. A lot of creativity is required here, while the amount of human interaction is lower than during analysis.

When one is creative, one can make mistakes, called defects—although making mistakes doesn't completely rely on creativity. It also relies on how the problem was specified, of course.

This brings me to another surprise, at least from what I see in the real world. Some companies still choose methods like OMT to do their strategic development. They also mention that design is very important. Well, as you know OMT has good points and some less good points. Design is one point where it is weaker. There is less possibility to be creative, for developers are not given the means to model object communication or object (class) referencing. It is no wonder they start to hack!

We measure and see all the time that Object Communication is essential to model before it is coded. But it needs a lot of creative thinking. You would not start to build a house before you know if will satisfy your needs, would you?!

During reviews we see that most of the time developers do some simple things wrong. Very often the sequence numbering in the object interaction graphs is wrong, or more than one system operation is modeled in one object interaction graph. Not that serious, but makes it difficult for others to code from it.

Another point is that some developers are thinking and working according the straight-jacket mindset of rush-to-code. One can conclude that they worked too long with a programming language so their creativity, or their will to do exploratory solution finding, has disappeared.

This phase takes a lot of time, but is a lot of fun. Just do it, and you'll find coding less interesting, and get better applications ;-)

Problem Implementation, or the coding phase, delivers the implementation of the found solution.

As you have probably guessed by now, it is the least interesting phase of all. Unfortunately we have to translate the designs to something the poor computer understands, namely code.

In this phase we "measured" low levels of human interaction and creativity. Everybody was caught within the "walls" of the programming language.

Developers that design completely according the method say that coding goes very much faster, and they do not compile as often to see if a little part is working (this they say with a big smile on their faces).

Because we look at the code metrics for every design, we can measure an improvement in complexity (lower), and amount of code. This shows that creativity works when given the right tools to stimulate it!

If you use the method as it is supposed to be used, the problem will turn out not to be that big of a problem at all. In fact, there is always a way to simplify the problem by working with higher-level design abstractions during the problem solution phase.

Kris Oosting,

DO WE HAVE THE TIME? PART II

by Kris Oosting,

In the previous newsletter we described basic time metrics and what can be learned from them. This article describes different views on the time metrics collected and why they can be useful.

All the information is based on Fusion Projects. Fusion allows to us measure the method itself as well as the development process that comes with it.

We collect time metrics in order to understand our development process. The results can then be used for estimating new projects.

In the previous article there were pie charts that showed the amount of time spent on each phase, like analysis, design, build, test, etc. If we take the collection of pie charts that were produced every week and place them in a nice row, we can then draw a line chart that indicates the progress flow. See Figure 7 for an example of a Progress Flow Chart. The x-axis shows the week number, and the y-axis the percentage of time.

Here you can see that Analysis dropped off after week 5, while Design and Coding grew steadily. In week 5 the “Other” activities started to increase as well. This category includes project meetings, admin, holidays, etc. In this case, “Other” increased because of the amount of time the

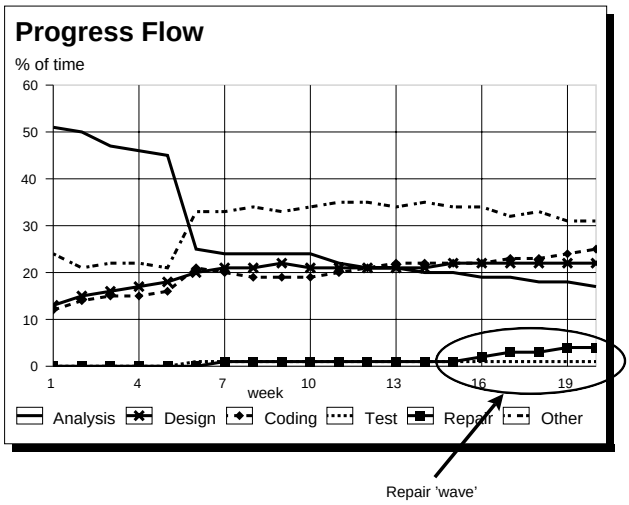

Figure 7 Chart of Progress Flow

developers spent negotiating about the underlying architecture of the framework from the technical point of view. You can also see that Repair followed after Test, but probably only essential repairs where done. From week 15 we see an increase in repair time. People "found" the time to repair the defects.

The chart in Figure 7 shows the relative proportion of cumulative time spent on each kind of activity. Thus, for example, we see that over the weeks the proportion of accumulated time spent on analysis decreases. When the project is finished, this chart will show the percentage of time that was needed for each phase. That is what we always want to know up front.

We also use non-cumulative charts. These charts show the percentage of time spent every week. It like "zooming in" on the first chart. These are more useful for direct control.

We used the chart for the following:

- to support estimates
- to predict what will happen (e.g., when certain resources are needed)
- to monitor the project and take preventive actions when something tends to go wrong

To Support Estimates

This actually happened more or less by accident. Every so often we ask the developers to review their estimates and to give a new estimate based on the information they know. This is done to get the EQF (Estimation Quality Factor) metric.

Two developers delivered their estimates for the second phase of their small project. They estimated they needed 3 times more time to code the design than before for similar functionality and complexity.

For example: they estimated 12 days for design and 60 days for coding. Before the metrics showed, over a period of about 20 weeks, that design time was about 60% of the time needed for coding. The metrics we had collected gave us a frame of reference for new estimates in the same project.

After reviewing their estimates and work breakdown structure with the developers, the revised estimate was 12 days for design and 26 days for coding. You'll probably now ask: what time did they spend for real then? Well, 10 days for design and 31 days for coding.

The first estimate was 31 days too high, but after reviewing and revising the estimate, it was only 3 days higher than the actual. Not so bad!

Conclusion: the chart can be used to check new estimates to a certain extent.

To Predict What Will happen

Another discovery from the real world is that this chart can indicate a change in effort after a certain event. In our case; what happens after Testing has been done?

In the chart you can see an increase in time spent testing. After the test period is finished, we asked ourselves which line will follow first, second, etc., and by how much?

For example, after testing was done, the list of defects was available and had to be solved. Logically after Test the Repair line will increase, because people are repairing the application to get rid of the defects found. The question is: how much time will people spent on repair, and where are the most defects? This information can be deduced from your Defect Tracking System.

To capture it in the Progress Flow Chart, we have to zoom in a little and replace the Repair line with Repair Analysis, Repair Design, Repair Coding, etc. lines. If this becomes to messy, than we can put it on a separate chart.

If the Repair Analysis line increases first, then one can expect an increase in Repair Design and Repair Coding as well. Defect Tracking Information will assist you in estimating the effort you need to support defect repair. The information can be collected per developer, per subproject, or for the total project. Just try it.

With these charts we can make the Fusion development process more visible. Use the results right away, or use them for later projects.

Figure 8 shows an example of the Repair Flow. This "snapshot" shows that at the beginning of the chart, fixing the code takes a lot of time. From week 14 on parts of the analysis have to be fixed. This means that a defect's origin was analysis. After week 18 the analysis defects are

repaired and the design needs to be changed as well. If we now need to predict the flow, we would say that the analysis and code wave decrease for a while. Then the design wave will decrease, and the coding wave will increase. The challenge now is to predict when and by how much. Well, that's the reason we collect all this information now, so we can predict better in the future.

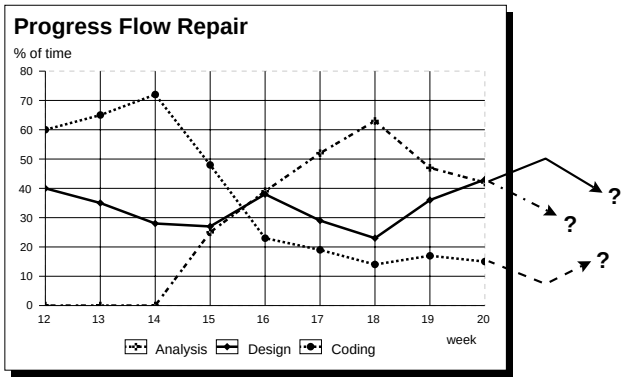

Figure 8 Progress Flow Chart for Repair

To Monitor The Project

The Progress Flow Charts show you what happens on a weekly base (or daily, monthly).

As a Project Manager plans the project, he or she will have an expectation of what will happen during the project. The project planning tool will assist one, but these time metrics will as well. (You probably understand that there are more metrics than just these ones.)

We found that project teams that are new to object technology, and do not have a clear idea what OO will bring them, found these charts very useful. People almost start to bet what will happen next. This shows that the collection of these metrics and producing these charts increases involvement in the project. This is very important. A project should not be a nightmare, but a challenge!

After these time metrics, which are easy to collect and to understand, it is time to collect more metrics of a different nature—but not too many.

Conclusion

The conclusion we come to is that we are always interested in how much time we spent where and why we spent it there. We collect this information in order to better understand what a typical Fusion project is. Or better, to find norm values for the metrics. Of course, projects depend on the experience of the developers, environment used, politics, etc. But even so, one can get a second opinion from the norm values that are found. The result is that projects can be estimated and guided more effectively.

In our experience, the Fusion Method and its process is measurable—and therefore controllable. As Tom DeMarco put it: "You can't control what you can't measure", and now we *Do Have The Time*! From yesterday, from today, and for tomorrow.

Kris Oosting.