RELEASING THE POWER OF OBJECTS

FUSION NEWSLETTER

Hewlett-Packard Laboratories Volume 4.2 April 1996

DO WE HAVE THE TIME?

by Kris Oosting

This article d escribes what time metrics we hav e found worthwhile collecting on Fusion projects and why they are useful.

Time metrics help us monitor project progress and provide better estimates in fu ture projects. We want t o understand where developers spend their time, so that we can use these metrics as an indicator of trouble on the project. We also want to kn ow how mu ch time each phase is consuming, and gain more insight into the way Fusion is being used in the development process.

Time Metrics

We collect time spent for the following items:

- analysis
- design
- build
- test
- repair: time needed to find, repair and test defects
- research: e.g., trying out alternative design ideas, prototypes, etc.
- other (project related): e.g., project meetings and administrative tasks
- non-project related, like meetings, system admin, vacation, illness, other.

Each person has to deliver certain work products during a phase and is responsible for certain tasks within that phase. F or example, a developer had to create t he Image Handling module. This is a co llection of objects that provide methods for performing image handling. The developer spent time on analysis, design, build, etc. In table format, the hours spent were as follows:

	Analysis	Design	Build	Test	Repair	Research
Image Handling	123	65	41	10	5	16

You can keep two tables. One with es timated values and one with the real values. This is very helpful if you want to collect the EQF (Estimation Quality Factor) metric to measure the quality of th e estimate. This gives you an indication of the percentage of schedule under- or overrun, and where and why this happened.

The numbers are p resented with th e help of pie charts. We use a chart for the who le project, and a chart for every subproject.

IN THIS ISSUE

The charts are produced every week, after the time forms are entered in the system—in this case a spreadsheet. Every project member has his/her own sheet. A summary sheet is produced for the weekly progress meetings.

Figure 1 shows the pie chart for the total project. It is good practice to show the non-project-related activities in a separate pie chart.

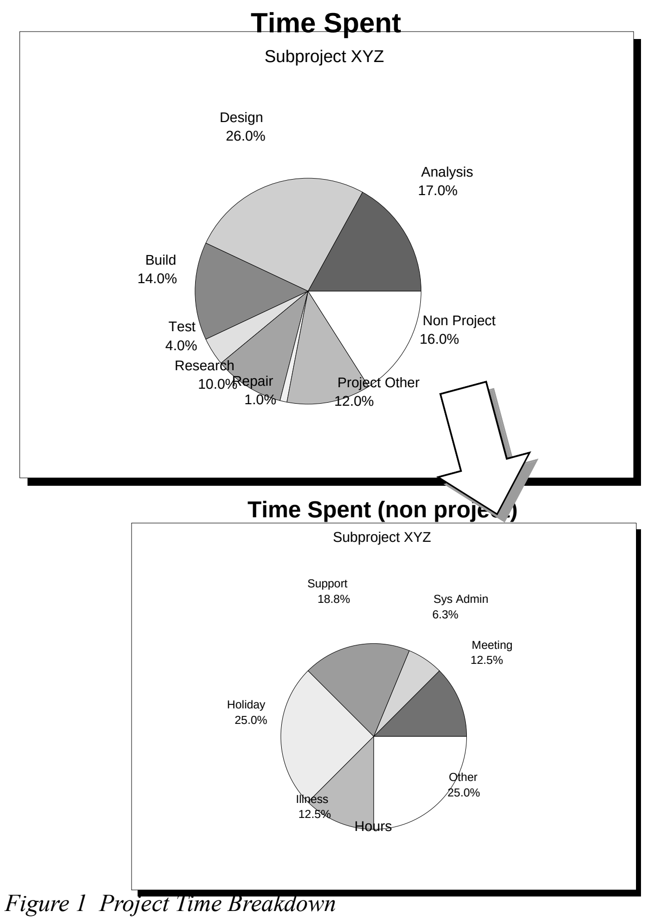

Figure 1 Project Time Breakdown

What can we learn from these charts?

Progress tracking. Of course, we can see how much time we spent where. If these numbers are reported every week, a project manager can see how a project is progressing, and where the project members spent their time.

Indicate trouble spots. The charts can also be used as a sort of early warning system. For example, a project has just moved into design. On the time reports we can see that for 3 weeks in a row the percentage of time spent on the build phase increases rapidly. This can indicate that the developers are first working on building the system and then later documenting it with Object Interaction Graphs, or they just "prototype," etc.

We actually had this occur on one project (a "let's document it afterwards" approach), and this was exactly what showed up in the time reports. As a result we made the reviews less free and code metrics had to be added to design documents after the code was tested. We did this to guide developers to build better systems.

To summarize, first the design (including the documents) is reviewed, then the code with the code metrics are added to the design document. These code metrics can be lines of code (NCSL, comment, empty, preprocessor), complexity, etc.

Now we can check if design complexity is in sync with code complexity. This means that simple designs should not result in complex code and visa versa.

Understanding the way Fusion is being used. If you place the numbers, time spent per project/method phase per week, in a line diagram, then you can see what happened over time. It is like a Method Jump Diagram [1] for time spent.

Improved estimates. Many of the projects we work with are delivered in incremental stages. For example, in 4 incremental delivery cycles, the first cycle delivers the basic system with limited functionality, but is fully tested and can go to customers for acceptance test. If there are defects, they will be solved in the second cycle. Cycle 2 delivers the basic system, repaired defects and additional functionality, and so on. By tracking the time spent in each phase of a cycle, you improve your ability to predict and plan the subsequent cycles.

At the end of the project you will be rewarded with a pie chart that tells you where the project members spent their time. These numbers can then be used for estimating (or checking the estimates for) the next Fusion project.

[1] Oosting, Kris. "Method Metrics and Defect tracking: Measuring and Improving the Fusion Development Process", Chapter 7. *Object-Oriented Development at Work: Fusion in the Real World.* Malan et al. eds . 1996.

Kris Oosting

Analysis in the Real World, Part II

by Kris Oosting

Part I o f this article, in the previous Newsletter, described what the impo rtant models are, ho w we used the metho d, what metrics to collect, and how we can test the models.

This article will continue with the questions raised in part I. These questions are:

- Were the analysis metrics useful?
- Did the analysis metrics provide us with information we can use for design?
- What did the scenario test offer us?

So far, a smooth interaction between analysis and design is vi sible. We still see a strong interaction between analysis models and the ob ject interaction graphs. This means that developers start to form the application and discover that assumptions made during analysis were not always the most optimal ones. This is ok—better now then after release. We are h appy to report that the metho d supports this process in a good way.

Let us s ee if the metrics were as useful as we had hoped for

Were the analysis metrics useful?

Number of Classes. Yes, we could measure growth of the models. After the last measurement we discovered that the number of classes increased by 40%. The question that arose directly was: Did we miss that much? No, just one developer forgot to enter his part, and used his own naming convention. No problem here, because the metrics did indicate that something was going on. The problem could be solved easily. So the metric is useful.

Number of system operations. This one also increased by 12. This means that design will take more time because there is more functionality to support. In this case the impact on the current planning was acceptable.

Number of Relationships. The number of relationships also increased. After checking, it turned out that the developers made the relationships more precise. So, for example, one relationship became three, expanding from conceptual to more detail.

For us the analysis metrics are sufficient. They work as an early warning system, to focus attention on the right spot.

The number of classes still doesn't give a lot of information for estimating design effort. On the other hand the system operations do.

Did the analysis metrics provide us with information we can use for design?

The system operations, or the amount of functionality we have to support with the application, can provide us with information to estimate the time we need to design them.

As described in "Method Metrics and Defect Tracking" (chapter 7, Object-Oriented Development at Work: Fusion in the Real World, 1996), to design one system operation as described in its operation model it takes an average of between 4 to 8 hours. This depends, of course, on the amount of reusable components and the complexity of the system operation.

At this time we are testing the idea of *categories* of system operations and output events.

What does this mean? For example, you can have a system operation that is responsible for grabbing some data and popping up a window to present it. Say there is no checking involved. This system operation is very simple and doesn't require much time to design.

In another case, we can have a system operation that is responsible for combining complex graphical structures, loading data from a database, and as a result (output event) displaying the resulting image. This one takes more time to design.

We saw that in the real world developers need a reasonable amount of time to get their alternatives on the table. Sometimes they need to try out a few of their ideas (yes they code), and then finalize the design. We had a case where it took a developer 4 days to design the system operation to get it right. After this process, coding was very straightforward.

So far the formula for calculating design effort is simple:

(# system operations * weight for this category) * hours per system operation

For example:

(10 system operations * 1.0 (GUI Category)) * 4 = 40 hours to design it.

Long, long, long ago when the Yourdon/De Marco method (SA/SD) was popular, Tom DeMarco wrote about the concept of Function Weight in his book *Controlling Software Projects*. He introduced Process Categories which have different weighting factors depending on their complexity. We used them then, so why not using them today?

In a subsequent article I will describe this in more detail.

What did the scenario test offer us?

Defects! Or, a little more precisely, we found defects. Most of them falling in the category of "missing information" or "unclear." Because we were still in analysis, they were easy to repair.

What was more important for us, was the fact that the test is useful. We remedied the defects and had a better problem specification (analysis) on entry into design.

The other advantage is that you can report early in the process what the status of the quality of the system is. "You can't control what you can't measure," so make sure you can measure it—then project managers will really be in control again.

Method metrics are essential. Using a color project planning package doesn't mean the project manager controls the project. If a project manager doesn't use any metrics, he or she can only guess what is going on. Then you'd better make sure the developers have sufficient coffee so they'll be nicer to the project manager if a crisis situation occurs :-)

The Fusion method offers the possibility to be measured, not only during analysis, but also during design. The Fusion models are easy to understand and do what they have to do. The process is described well, so use it and you'll be happier!

The Scenario Test is summarized below:

Name. Scenario Test.

Purpose. To test the scenario as described in the user requirements against the user interface. To find possible defects.

Test Method. Walkthrough, simulation.

Models Involved. User Requirements Document, (Graphical) User Interface, System Object Model, Operation Models, and Life-cycle Model.

When to Test. During or at the end of the analysis phase.

What to Test.

- Does the life-cycle reflect the scenario the user is expecting?
- Are parts of the GUI not implemented?
- Is a system operation from an agent followed by the right output event(s) in the GUI?
- Are the right classes in the System Object Model used? (if you test a system operation or functionality it's like you 'send' a query to the system object model)
- etc., etc.

Metrics.

- Number of System Operations generating an incorrect result (check with life-cycle).
- Number of incorrect Output Events. Check with life-cycle model and GUI layout.
- Number of incorrect classes.
- Number of incorrect relationships.
- Number of Defects.

Comments.
CASE tools can perform standard syntax and method checks.

Defects in the requirements can be found during this test and should be submitted and corrected.

Our report from the real world will be continued.

Kris Oosting

Does the Fusion Method Work?

by Kris Oosting

This is a question a lot of developers have. Many people see a method as extra work to document their code. Well, as most of you Fusion users know, this is not the case. The Fusion method documents the system, but isn't extra work. We had to find a way to convince a few developers, and I just want to share it with you. It's called "The Proof."

In order to demonstrate that a method is working, you have to talk the "victim's" language. We decided to use control complexity (McCabe) as an indicator. Four systems were measured. Three of them were already finished, and one is still under construction but is useful for the comparison.

All projects use a windows environment (OSF-Motif, MS-Windows), and C++. Project A has also parts written in C.

The following metrics were presented:

- Highest complexity—to show the limit
- Average Complexity—to give an indication of the whole product
- Percentage of modules with complexity above 40—an indicator for maintenance and defects to expect
- Complexity increases by 1 for every x NCSL—to show how many lines of code are needed to increase the complexity by one.

The results of the measurements are:

Project A: bits and pieces of OMT and OOA (Coad/Yourdon) used, 85K lines of code
Highest complexity: 320
Average Complexity: 68
Percentage of modules with complexity above 40: 26%
Complexity increases by 1 for every: 4 NCSL

Project B: OMT used, 35K lines of code
Highest complexity: 140
Average Complexity: 20
Percentage of modules with complexity above 40: 20%
Complexity increases by 1 for every: 7 NCSL

Project C: OMT used, 18K lines of code
Highest complexity: 240
Average Complexity: 40
Percentage of modules with complexity above 40: 30%
Complexity increases by 1 for every: 10 NCSL

Project D: Fusion used, 15K lines of code (70K estimated)
Highest complexity: 21
Average Complexity: 16
Percentage of modules with complexity above 40: 0%
Complexity increases by 1 for every: 30 NCSL

If we had moved the maximum complexity to 20, then projects A, B, and C result in about 50% to 60% of their modules are too complex. But because we're dealing with a complex environment and problem, a complexity of 40 is allowed.

Project D already had good design documentation in place (part of the Fusion method), the other projects have no documentation concerning object interaction, object visibility, and sometimes not even inheritance graphs.

Project D produced code with a lower complexity and is easy to maintain. We see that developers now start to look at object interaction graphs to find defects instead of "browsing" through the code.

The Result?

Again, from these few metrics we can conclude that there is a strong indication here that the Fusion method does work!

There are other metrics one can measure to understand the usefulness of a method. One of them is called "Frustration Level."

Frustration Level

The "frustration level" metric has to do with a more human aspect of software development—frustration!

The amount of frustration the developers feel can often be found in code. To measure the frustration level of the developers when coding, we introduced the "Dirty Words Scan." This scanner searches in the source code for words that can indicate frustration, irritation, and potential problems. If the source code contains a lot of "dirty" words, we often measured a high complexity in code and found a less than optimal design.

These words can be:

- *side*, to find words like "side effect"
- *work*, to find words like "doesn't work"
- *problem*, to see if there are still unsolved problems
- *crash*, to find cases where the programmer indicates a potential crash
- *never*, like "this never works," "never do this," etc.
- *hack*, to see if the code was hacked together. e.g., "Hopefully this hack will work..."
- *illegal*, too see if programmers did "illegal" programming
- *maybe*, like "maybe this will work"
- *later*, like "added for later use"
- *future*, like "in the future this might work"
- *trick*, to see if a special coding trick used
- *fix*, to locate bugfixes

This is a handy tool when you have to perform an audit. Once the developers know you're looking for these words, they will change the way they document their code. And that's what you want if you deliver source code to your customers.

Don't forget to ask the developers why they used the words in the first place. We discovered that when asking these questions you're touching an emotional chord, and they will probably tell you all the things that frustrate them. This may not be nice to hear, but good to know if you want to improve your process.

In practice, during audits of commercial software we found that some interesting phrases come up. The number of dirty words increases if the project becomes late and the pressure gets high. If the method doesn't support the developers well at that point, they will start to hack and express this with their "favorite" words.

Kris Oosting