UML and the Cost of Defects

Stephen J Mellor

stephen_mellor@mentor.com

It is common knowledge that software defects, especially in embedded systems, are expensive to repair; less well appreciated is just how very expensive it is, especially for requirements defects. This paper outlines these costs and how they depend on the development process. We describe three approaches to software development starting with UML models, add verification through execution, then add translation, and examine how each affects the cost of defect repair.

Costs

The Software Engineering Institute estimates that defects are injected, per hour worked, at these percentages in the lifecycle: 7% during requirements, 7% during high-level design, 21% detailed design, 55% coding, and 10% thereafter.

Barry Boehm [1] showed that the cost to repair an error, increases exponentially the later it is found. The stages of the lifecycle differ from those of the SEI; moreover, costs differ according to the size of the project. Extrapolating across the missing phases, we assume here a doubling for each stage that an error persists.

The table below combines the data from Boehm and the SEI. The first data row copies the SEI data above; the second shows the relative cost to fix the error, derived by doubling the cost for each of five phases, moving backwards from deployment; the third multiplies the two together to arrive at some absolute cost. The final row normalizes against the total of absolute costs (372) to arrive at the percentage of the total cost of rework for each type of error.

Phase	Requirements	High-level Design	Detailed Design	Coding	Test and Deployment
Percentage	7%	7%	21%	55%	10%
Relative Cost	16	8	4	2	1
Absolute Cost	112	56	84	110	10
Normalized Cost	30%	15%	23%	30%	3%

The table shows that 30% of defect repair time goes into fixing requirements errors, even though only 7% of the errors come from requirements.

Standish Group reports (2000) that 60-80% of the cost of software development goes to rework, which seems a little high. If we assume about 50%, then 15% of all project costs through initial deployment come from requirements errors alone! It is therefore of some considerable value if these defects can be removed early.

Development Processes

To elucidate the cost of defects, we examine three approaches to software development. First, we consider a process in which we build a UML-based specification, then code from that. Second, we build an *executable* model, verify that the behavior is as desired, then write code from that. Third, we consider the construction of an executable model that is *translated* into code.

Documentation and Visualization

Many people use UML models for documentation and visualization purposes. At the beginning, requirements are exposed and discussed with clients and experts; defects are found and fixed. The models are then grown organically to describe the high-level design. Changes are sometimes made to the requirements—both deliberately and inadvertently—as design issues come to the fore. In turn, more detail is added to capture the detailed design. At some point, victory is declared and code is written.

Execution

Execution is the idea that we can describe the behavior of a system without describing its implementation. This is achieved by defining a UML-based language that thinks in terms of sets of data and communicating state machines. The sets of data can be implemented by lists, arrays, trees or whatever, and the communicating state machines can be implemented in tasks, processors, hardware, or even sequentially. The model can execute, but its implementation is (initially) unstated, just as a statement such as x = y; can execute, even though it does not specify the implementation in terms of registers and memory. See [2] for details of the language.

The language abstracts away details of data structure, tasking structure, and control structure, just as third-generation languages abstract away details of register and memory allocation.

Translation

An executable model can be translated into an implementation using a *model compiler*, which consists simply in a set of rules. For example, there may be a rule that transforms sets of data into lists or arrays, and another rule that allocates state machines to tasks and processors, along with the infrastructure code required for communication. This code will be regular and uniform across the entire application.

Model compilers add in this information depending on the type of system to be built. The code generated for an embedded system, for example, would be different from real-time transaction processing, or an IT system.

Cost Reduction

Documentation and Visualization

The data quoted in the first section comes from a variety of sources spread over thirty years, from around the beginning of structured analysis and design to the forefathers of UML. We assume that a textual requirements specification, supported by some informal graphics such as SA/SD, is the basis for the data shown above.

It would be uncharitable to conclude that UML has had no effect on reducing defects, though there is little supporting data. Given the nature of UML, one would certainly expect to see reductions in requirements defects, in high-level design defects, and—since UML is often used to visualize software structure—detailed design. UML does not help directly with coding, testing and initial deployment.

The first row of the table below shows the normalized cost from the table in the first section. The second row is my (generous?) assessment of how much using UML for documentation and visualization can help with defect reduction. The final row applies those assumptions to arrive at a revised normalized cost for defect reduction using this approach.

Phase	Requirements	High-level Design	Detailed Design	Coding	Test and Deployment
Normalized Cost	30%	15%	23%	30%	3%
Defect Reduction	A third	A third	A half	None	None
Revised Cost	20%	10%	12%	30%	3%

The total is 75%. In other words, under these assumptions, using UML reduces the costs of rework of defects by 25%. Continuing with a 50% total cost of rework estimate yields a 12.5% reduction in total project cost through initial deployment.

Execution

A UML specification without execution can only be defended by argument or exhaustion—one cannot know it is correct until it executes. However, a (partial) executable model, because it has an execution semantics, can be run with real data, and the results passed back to customers and marketing, thus providing immediate feedback. Defects can be repaired at the beginning of the project instead of waiting for the first system build.

Most defects in requirements can therefore be eliminated during the requirements phase, and the cost of reworking them too. There is also a cost to building the scenarios to run the model, which is equivalent to the work involved in the construction of unit tests, except that it occurs earlier in the lifecycle. Design proceeds exactly as before. When coding, the results of units tests can be compared to those run and the model, and any defects can be repaired then and there.

The second row below records these assumptions. The last row in the table once again shows the reduced cost. The total is 50% for the total cost of rework caused by defects, which is 25% of total project cost.

Phase	Requirements	High-level Design	Detailed Design	Coding	Test and Deployment
Normalized Cost	30%	15%	23%	30%	3%
Defect Reduction	Most	A third	A half	A third	None
Revised Cost	5%	10%	12%	20%	3%

Translation

When hand coding, design decisions are smeared across the system as each developer makes isolated "creative" choices that do not necessarily (more exactly, hardly ever) integrate properly. It is this that is the cause of the majority of the 21% of detailed design errors, and a good number of the high-level design errors too. In theory, it is possible to describe the system architecture and demand that coders follow it. In practice, such documents are rarely more than high-level guidelines, and, without enforcement, are rarely followed completely.

Enforcement can be had through automation. Rather than hoping that programmers follow a set of rules, we can formalize these rules and apply them automatically to the executable model to produce the code for the system. This is the model compiler; it houses the system design and provides automatic translation rules from an executable model to code.

A project using translation requires an executable specification. Hence, the project will derive the benefits from defect reduction during requirements, as described above.

High-level design errors are limited to whether you chose the correct class of model compiler. For example, a fundamentally periodic application would not benefit from a completely asynchronous, event-driven high-level design (and vice versa.).

Detailed design errors will be slashed, and often eliminated completely. If the model compiler yields an implementation that meets all performance requirements, then the design work is complete. If not, then new rules must be written. To test the new rules, compare the output to that from a known-to-be-correct model compiler. Behaviorally, they must be the same, though their performance may differ.

There will be *no* coding errors, *none*, because an out-of-the-box the model compiler will generate correct code, just as a programming language compiler may be expected to generate correct code.

This code will be regular and uniform across the entire application, and it reduces to zero the defects that can be introduced by hand translation.

Phase	Requirements	High-level Design	Detailed Design	Coding	Test and Deployment
Normalized Cost	30%	15%	23%	30%	3%
Defect Reduction	Most	Most	Most	All	All
Revised Cost	5%	5%	5%	0%	0%

UML and the Cost of Defects

Let's face it: There's more precision than accuracy in this paper. The figures we do have are based on incomplete definitions and unknown premises. Moreover, the word "assume" makes more appearances than is comfortable for this engineer. Nonetheless the overall pattern is clear: defects are expensive to remove; they are much more expensive when they occur early in the lifecycle; and the development process has a large effect on these figures—even if they are guesses.

The table (over) summarize the results.

Phase	Cost of Rework	Cost in Project
Original Cost	100%	50%
Documentation and Visualization	75%	37.5%
Execution	50%	25%
Translation	15%	7.5%

In other words, using UML for documentation and visualization can reduce the cost of rework by 25%; using executable UML models can reduce the cost of rework by 50%; using translation can reduce the cost of rework by 85%. If half of project time is spent on rework, that comes to 12.5%, 25% and 42.5% reductions in total project costs.

While the numbers might not be as accurate as we would like, it is worth remembering Boehm's result that the same error, if found in operation, can cost over 100 times more to fix—and that excludes the reputation risk of, say, an automobile or medical device that fails while in the hands, or heart, of the consumer.

---

[1] Boehm and Basili, Software Defect Reduction Top Ten List. *Computer Vol. 34,* January 2001

[2] *Executable UML: A Foundation for Model-Driven Architecture*, Mellor and Balcer, Addison-Wesley, 2002.