The Estimation of Effort Based on Use Cases

John Smith

Rational Software White Paper

TP 171, 10/99

Table of Contents

The Problem

Intuitively, it seems as though it should be possible to form estimates of size and effort that development will require based on characteristics of the use case model. After all, the use case model captures the functional requirements, so should there not be a use case based equivalent of function points? There are several difficulties:

- There are many variations of use case specification style and formality which makes it very difficult to define metrics—one might like, for example, to be able to measure the length of a use case.
- Use cases should represent an external actor's view of a system, and so a use case for a 500,000 software lines of code (sloc) system is at a quite different ***level*** to a use case written for a 5,000 sloc subsystem (Cockburn97 discusses the notion of levels and goals).
- Use cases can differ in complexity, both explicitly as written, and implicitly in the required realization.
- A use case should describe behavior from the actor's point of view, but this can be quite complex, especially if the system has states (as most do). So to describe this behavior may require a model of the system (before any realization is done). This can lead to too many levels of functional decomposition and detail, in an attempt to capture the essence of behavior.

So, is some kind of use case realization necessary to make estimation possible? Perhaps expectations about estimation directly from use cases are too high, and drawing parallels between function points and a notion of use case points misguided. The calculation of function point counts requires a model of the system anyway. The derivation of function points from use case descriptions would require a uniformity of level in use case expression and it is only when the realizations start to emerge that one would have much confidence in a function point count. Fetcke97 describes a mapping from use case to function points, but again the level of the use case has to be appropriate for the mapping to be valid. Other methods use class/object-based metrics as a source, PRICE Object Points, for example (Minkiewicz96).

Other Work

There is a fair amount of work on describing and formalizing use cases—Hurlbut97 has a good survey. There is a lot less on deriving estimation metrics from use cases. Graham95 and Graham98 contain quite severe criticism of use cases (but I do not fully understand why he believes his ideas and use cases are so far apart), and propose the idea of 'task script' as a way to overcome the problems with use cases, including their varying length and complexity. Graham's 'atomic task script' is the basis for collection of a 'task point' metric. The problem with an atomic task script is that it is very low-level: according to Graham, it should ideally be a single sentence, and is not further decomposable using only domain terminology. Graham's 'root tasks' contain one or more atomic task scripts, and each root task corresponds "to exactly one system operation: in the class that initiates the plan."(Graham98). These root tasks seem very much like low-level use cases to me, and the atomic task scripts like steps in such a use case. Still, the problem of level remains.

Other work has been done by Karner (Karner93), Major (Major98), Armour and Catherwood (Armour96) and Thomson (Thomson94). The Karner paper posits a method for calculating use-case points, but again assumes that the use cases are expressed in a way realizable by classes (i.e., at a finer level of detail than subsystems).

So, should we avoid use cases for estimation and rely instead on the analysis and design realizations that emerge? The problem with this is that it delays the ability to make estimates and will not be satisfactory for a project manager who has chosen this technology—early estimates ***will*** be required and other methods would then have to be used. It is better for the project manager to be able to obtain estimates early for planning purposes, and then ***refine*** them iteration by iteration, rather than delaying estimation and proceeding in an unplanned fashion.

What is described in this paper is a framework in which use cases at any level can be used to form an effort estimate. To present the ideas, some simple canonical structures are described, with associated dimensions and sizes that have some basis in experience. The paper is full of bold (or should that be bald) conjecture because I can see no other way forward given the lack of work and data in this area. I have drawn on the 'systems of interconnected systems' idea in the formulation.

Next, I'll digress briefly to set down some background thoughts that set me going down this path.

Avoiding Functional Decomposition?

The idea of functional decomposition seems to be an anathema to many in software development. And my personal experience of functional decomposition taken to an extreme (three thousand primitive transforms in a very large data flow diagram, five or six levels deep, done with no thought to architecture, except at the infrastructure level) didn't leave me feeling sanguine about it either. The problem in this case was not just with functional decomposition though, but also with the idea of not describing a process until the functional primitive level is reached, at which point the specification should be less than one page in length.

The result is very hard to understand—how desired behavior required at a higher level emerges from these primitive transforms is difficult to discern. In addition, it is not obvious how the functional structure should map to a physical structure that will meet performance and other quality requirements. So the paradoxical thing was that we decomposed and decomposed until we reached the level at which we could 'solve the problem' (the primitive level), but it was not clear or demonstrable that the primitives working together actually met goals at higher levels. There was no way in this method to take account of non-functional requirements. The architecture, in its totality, not just the infrastructure (communications, operating system, etc.) should have been evolving alongside the decomposition and each should have influenced the other.

What about the Bauhaus edict that 'form follows function'? Well there were many good things that flowed from their functionalist approach to design, but some bad ones too, such as the use of flat roofs everywhere. If you have regard only to the function of a roof and subordinate design totally to the roof being a cover for the inhabitants, then the result, at least in certain areas will be unsatisfactory. Such roofs are difficult to waterproof; they will collect a lot of snow.

Now these problems can be solved***, but at greater expense than if you had chosen a different design***. So although it seems trite to say it, form should follow *requirements*—all of them, functional and non-functional, and these latter may include aesthetics. The problem for the architect will often be that non-functional requirements are often poorly stated and much reliance is placed on the architect's experience of 'the way things should be'. So functional decomposition is bad if it solely drives the architecture—if decomposition proceeds several levels down and the functional primitives map one-to-one with 'modules'—and define their interfaces.

Considerations like this convinced me that it would not make sense to decompose use cases either down to some normalized level (that could be realized by a collaboration of classes) ***in advance of architectural work.*** That decomposition will occur is certain if the system is of some size (see Jacobson97) but the criteria and engineering process for decomposition are important—ad hoc functional decomposition is not good enough.

System Considerations

Systems engineers do functional analysis, decomposition, and allocation (when synthesizing a design)—but function is not the only driver for the architecture—teams of specialty engineers will contribute in assessing alternative designs. IEEE Std 1220, the Standard for Application and Management of the Systems Engineering Process, describes the use of functional decomposition under section *6.3, Functional Analysis* in subsection *6.3.1 Functional Decomposition*, and system product solutions under section *6.5 Synthesis*. Of particular interest are subsections *6.5.1 Group and Allocate Functions* and *6.5.2 Physical Solution Alternatives*. In section *6.3.1*, it says that decomposition is performed to ***understand*** clearly what the system must accomplish, and ***generally one level of decomposition is sufficient.***

Note the purpose of functional decomposition is not to shape the system (synthesis does that) but understand and communicate what the system must do—a functional model is a valid way to do this. In synthesis, the subfunctions are allocated to solution structures and then the solution is evaluated—taking into account all other requirements. The difference between this approach and multi-level functional decomposition is that at each level you try to describe the required behavior and find a solution to implement it, before deciding whether the behavior at the next level needs to be further refined and allocated to lower level components.

One conclusion from this is that it is not necessary to have hundreds of use cases to describe behavior at any one level. The number of external use cases (and associated scenarios) that will adequately cover behavior of the thing described—system, subsystem, class—can be quite small. I should say what I mean by external use case. Take the example of a system composed of subsystems that in turn are composed of classes. The use cases that describe the behavior of the system and its actors, I've called external use cases. The subsystems may also have their own use cases—these use cases are internal to the system, but external to the subsystem. The total number of use cases, ***external and internal***, ultimately used to construct a very large (say 1,000,000+ lines of code) system could be in the hundreds, because systems of that size will be constructed as systems of systems, or at least systems of subsystems.

Assumptions about Structure and Size

Number of Use Cases

At Rational® Software, we have generally taught that the number of use cases should be small (10–50, say) and observed that a large number (over 100, say) of use cases may indicate a lapse into functional decomposition, where the use case is not delivering anything of value to an actor. Nevertheless, we do find large numbers of use cases in real projects, and not all are 'bad'—they cover a mix of levels—for example, in a Rational internal email, the author quotes an example from Ericsson:

> Ericsson, modeling large portions of a new generation of telephone switch, estimated to be +600 staff years (at peak, 3–400 developers), 200 use cases ***(using more than one level of use cases, refer to "Systems of Interconnected Systems")*** (my italics)

For a system of 600+ staff-years (how big is this? 1,500,000 lines of C++ code?), I suspect that the use case analysis stopped one level above the subsystem (that is, if one defines a subsystem to be 7000–10000 lines of code), otherwise the count would have been higher still.

Therefore, I'll stay with the notion that a small number of *external* use cases is adequate. To match the structures and dimensions I've proposed, I'm asserting that ***10 external use cases,*** each with ***30 associated scenarios***[1] are adequate to describe behavior[2]. If in a real example, the number of use cases exceeds 10, and they are genuinely external at that level, then the system being described is larger than the corresponding canonical form. I'll try to provide some supporting reasoning that these numbers are sensible later in the paper.

Structural Hierarchy

The structural hierarchy proposed is:

4 — SystemOfSystems

3 — System

2 — SubsystemGroup

1 — Subsystem

0 — Class

Class and Subsystem are defined in UML; the larger aggregates are subsystems (containing subsystems) in UML. I've named them differently to make discussion easier. The aggregate subsystemGroup is a CSCI-like size, for those who know the terminology from military standards like 2167 or 498 (which would make a subsystem a CSC and a class a CSU). As I recall, after the arguments in the 2167 days over what Ada construct should be mapped to what level, when the dust settled, the Ada package was usually mapped to CSU. I'm not suggesting that systems must rigidly conform to this hierarchy—there will be mixing between levels—but the hierarchy allows me to reason about the effect of size on the effort per use case.

There will be use cases at each level (although probably not for an individual class), but not a single mass of incredible detail, rather use cases for each component (i.e., subsystem, subsystemGroup, etc.) at that level[3]. I've asserted above that there should be 10 use cases for each component at each level. If the use-case descriptions average 10 pages, this gives a potential specification document length of 100 pages (plus a similar or smaller number more for non-functional requirements). This is a

[1] In UML1.3 a scenario is described as: "**scenario:** a specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction or the execution of a use case instance". It is used here in the second sense of illustrating the execution of a use case instance.

[2] Note that this number (of scenarios) is intended to reflect the complexity of a use case—it is not suggested that a developer ***must*** produce and write down 30 scenarios for every use case—rather that 30 scenarios captures most of the interesting behavior for a use case, even though there may be many more paths through the use case.

[3] Some reviewers expressed alarm at the prospect of use cases at four levels, but note that this would only be for a system of systems, which will typically be very large. In such cases, I would not be surprised to see use cases at four levels, particularly if the work is done by a prime contractor (for the system of systems), subcontractors (for the systems) and maybe even sub-subcontractors for the subsystems.

number favored by Stevens98, and is close to that suggested in Royce98. But why 10 use cases? To arrive at this, I reasoned from the bottom up, based on what I thought were reasonable sizes for number of classes per subsystem, class size, operation size, and so on. These are collected together for reference with the other assumptions in the following table.

Operation size	70 slocs
Number of operations per class	12
Number of classes per subsystem	8
Number of subsystems per subsystemGroup	8
Number of subsystemGroups per system	8
Number of systems per systemOfSystems	8
Number of external use cases (per system, subsystem, etc.)	10
Number of scenarios per use case	30
Pages per use case description[4]	10

I do not have a great deal of empirical data—there are bits and pieces scattered throughout the texts. Lorentz94 and Henderson-Sellers96 have some data and I have some data from projects in Australia, mainly in the mil-aerospace domain. In any case, it was important at this stage just to get the framework positioned more or less in the right place.

Size of Components in the Hierarchy

I should say here that I have used lines of code knowing that some folks don't like the measure. These are C++ (or equivalent level language) lines of code, so it would be easy enough to backfire to function points.

There must be some relationship between the number of classes in a container and the richness of the behavior that can be expressed. I chose eight classes/subsystem[5], eight subsystems/subsystemGroup, eight subsystemGroups/system, and so on. Why eight?

- It's within 7, plus or minus 2.
- Because at 850 slocs of C++ per class (12 operations of 70 slocs each), it gives a subsystem size of ~7000 slocs—a chunk of functionality/code that is deliverable by a small team (say, 3–7 staff) in 4–9 months, which should harmonize with the iteration length of systems in the range 300,000–1,000,000 slocs (RUP99).[6]

So, what is the number of use cases that would express the behavior (externally) of eight classes, which are cohesive and have been co-located in a subsystem? It is not simply the number of use cases but also the number of scenarios for each use case that determines the richness. Now there is not much in the way of guidelines for scenarios/use case expansion—Grady Booch indicates in Booch98 that: "There's an expansion factor from use cases to scenarios. A modestly complex system might have a few dozen use cases that capture its behavior, and each use case might expand out to several dozen scenarios…", and Bruce

[4] Later in the paper, this is refined for different classes of systems.

[5] I believe that this sort of count is representative of analysis—there will be an expansion and refactoring through design and implementation, and the number of classes increases by a factor of three or more, whereas the operation size and class size decrease correspondingly.

[6] For smaller systems (shorter iteration times) the subsystems may be planned to be smaller or it is always possible to plan for partial delivery for each iteration—although this needs careful control and may require the delivery of 'stubs'.

Powel Douglass says in Douglass99, "…. many scenarios are required to fully elaborate a use case—typically one dozen to several dozen". I've chosen 30 scenarios/use case—that's on the low side of 'several dozen", but Rechtin (in Rechtin91) says that engineers can handle 5–10 interacting variables (which for the purposes of this argument I interpret as 5–10 classes in a collaboration) and 10–50 interactions (which I've interpreted as scenarios). Interpreted this way, multiple use cases are multiple instances of this variable space.

Therefore, 10 use cases, each with 30 scenarios, says that 300 scenarios total (which will later lead to ~300 test cases) are sufficient to cover the interesting behavior of eight classes. Is there any other indication that this is a reasonable number? If the 80–20 rule of Pareto applies, then 20% of the classes will deliver 80% of the functionality and, similarly, 80% of the functionality will be delivered by 20% of the operations in each class. Let's be conservative and say that we need 20% of classes, etc., to reach 75% of the capability, and construct a Pareto distribution through this point (Figure 1).

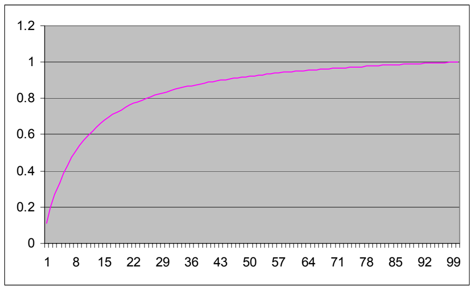


Figure 1: A Pareto-like distribution

If we want 80% coverage of the behavior overall and the Pareto rule applies to number of classes, operations, and scenarios, then we need 93% (0.93^3 is 0.8) behavioral coverage from each—that is requiring 50% of each; i.e., 4 classes and 5 operations (= (12 less 2 constructor/destructor)/2). The number of different traversals of the tree of nodes constructed to represent the execution patterns of four classes with five operations each could run to many thousands. I constructed one with up to three links from each node, assuming a hierarchy, with 10 operations (interface operations) at the top, and forming a tree of three levels. This gives close to 1000 paths or scenarios. So 500 scenarios should give 93% coverage. With 300 scenarios, (using the same assumptions) we should get about 73% coverage. Examining how the tree might be pruned, to eliminate redundant behavioral specification, suggests even smaller numbers may be adequate, depending on the algorithm chosen.

Another way of approaching this is to ask how many test cases (derived from scenarios) would one expect for 7000 slocs of C++. These tests would be anything beyond the unit test level and there is some evidence from Jones91 and the Boeing 777 project (Pehrson96) that this number is safe, at least in that it represents practice. These sources suggest that between 250–280[7] are about right. At a completely different level, the Canadian Automated Air Traffic System (CAATS) project uses 200 system tests (private communication).

Use Case Size

How 'big' should a use case be? Big enough to present enough detail so that the desired behavior may be realized—this will depend on its complexity, internal and external, which will be related to the type of system. Here we run into the problem of how much of the internal action of a system should be described. To build a system from a description of its external behavior requires, obviously, that outputs be related to inputs. Now if, for example, the behavior is history sensitive and complex, it will be very difficult to describe it without some conceptual model of the inside of the system and the actions it takes. Note though that this does not necessarily describe how the system is to be constructed internally—any design that satisfies the non-functional requirements and which matches the behavior of the model will do.

[7] From feedback I received from reviewers inside Rational, it is felt that this is more than enough for most non-critical systems; that these systems will have fewer than 30 scenarios per use case. It would be interesting to have more data on this and the relationship between numbers of test cases and the number of defects discovered in use.

The definition offered in UML1.3 is: "**use case [class]:** the specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system". For complex behavior, this definition can reasonably be taken to include internal actions—unless this is to be postponed until realization—which is a further step away from the end-user. Business rules should also be incorporated into use cases to constrain the actors' behavior; for example, in an ATM system, a bank may have a rule that no more than $500 may be withdrawn in a single transaction, no matter what the balance of the account.

With this kind of interpretation, the use-case flow of events description can vary between 2–20[8] pages. Algorithmically simple systems with simple behavior will obviously not need lengthy descriptions. Perhaps we can say that simple business systems are characterized at 2–10 pages with a mean of 5. More complex systems, business and scientific at 6–15 pages with a mean of 9, and complex command and control at 8–20 pages with a mean of 12 (these ratios reflect the non-linear relationship of effort to system type for systems of the same size) although I have no data to back this up. More expressive descriptive forms, state machines, or activity diagrams, for example, may take less space. We still tend to emphasize text, so I'll ignore the others for now—there is little or no data anyway.

Developments that differ systematically from these sizes should apply a multiplier to the hours per use case derived from these heuristics (I suggest adding a COCOMO-style cost driver, which is the observed mean size/suggested mean size for the system classification—simple business, more complex, command and control, etc.).

Another aspect of use case size is the scenario count; for example, a use case that is only 5 pages long may have a complex structure that allows many paths. Again, the number of scenarios needs to be estimated and the ratio of this to thirty (my initial guess at a number of scenarios per use case) used as a cost driver.

The consequence is that we are asserting that a use case based specification of ~100 pages should be enough for an external specification at any given level, in addition to the supplementary specification. The range is from 20–200 pages (these limits are fuzzy). Note though that the total for a system (of subsystemGroups) ***at the lowest level*** is 3–15 pages/ksloc (simple business system)—12–30 pages/ksloc (complex command and control). This seems to explain the apparent contradiction between Royce98 Table 14-9 where the page counts for artifacts are quite small and observation of real projects, which, particularly in defense have produced large amounts of paper. This paper comes from a level of specification which need not be committed to paper—Royce is right, the important things, like the Vision Statement should be of the order indicated in the table—200 pages, for large, complex systems.

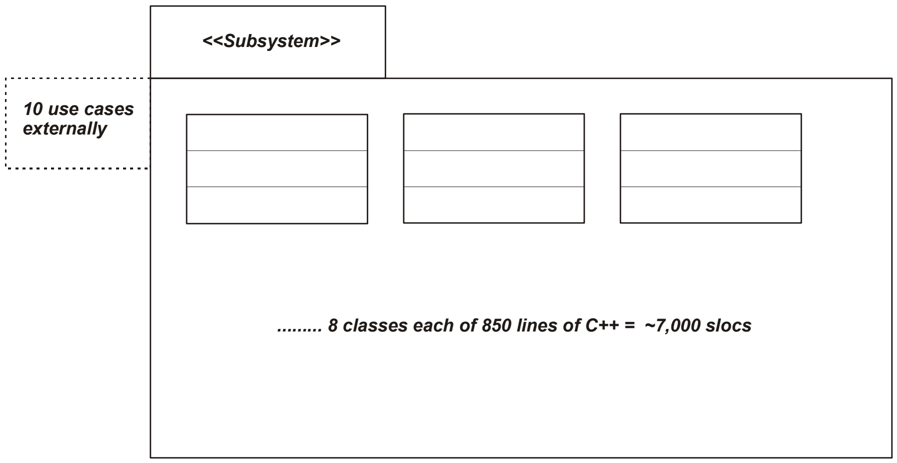
The Subsystem Hierarchy

What does this look like as a subsystem hierarchy? Here are the simple 'standard' forms I have used. Note these are the conceptual forms used to realize a system. The actual system boundary is outside a collection of these forms, and the sum of the external use cases for each is the total of external use cases for the system; thus a real system may have more than ten external use cases, but the upper bound is not unlimited as we see later. Note that it is not suggested here that all developments must use four levels of use case in their description. Smaller systems (<50,000 slocs) will likely use only one or two.

[8] Note that this is not intended to be a hard upper boundary; the length of a use-case description will follow some kind of statistical distribution, where the extremes have a lower probability of occurrence.

Level 1

At Level 1, we have use cases realized by classes in zero or more subsystems:



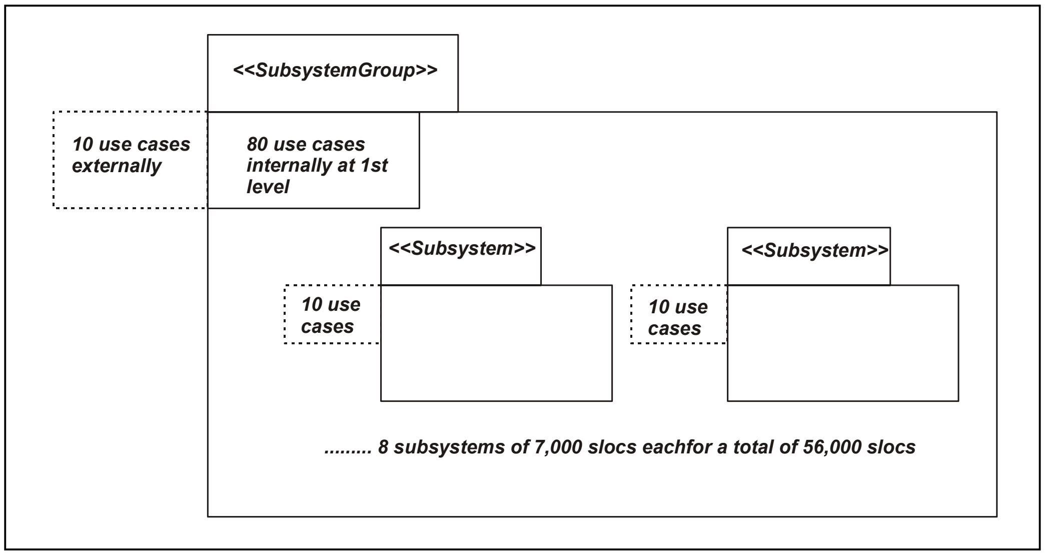
Estimating ranges of size for systems at this level (using the notion of 7 plus or minus 2):

- from 2 to 9 classes (not formed into subsystems)—1700 slocs to 8000 slocs, or
- 1 subsystem of 5 classes totaling 4000 slocs up to
- 9 subsystems of 7 classes totaling 53,550 slocs,

with the use cases expressed to be realizable by class instances. That's a range of 2–76 use cases. These are fuzzy limits, at least the upper limit is—the probability of building a system this way (at this size), never expressing desired behavior in some higher level form, should decline to zero at this limit. A larger use case count may indicate some pathology.

Level 2

At the next level, we have a subsystem group of eight subsystems. I think this is equivalent to a computer system configuration item (CSCI) in defense terminology. At this level, use cases are realized by collaborations of subsystems:


<<SubsystemGroup>>
10 use cases externally
80 use cases internally at 1st level
<<Subsystem>>
10 use cases
<<Subsystem>>
10 use cases
.... 8 subsystems of 7,000 slocs eachfor a total of 56,000 slocs

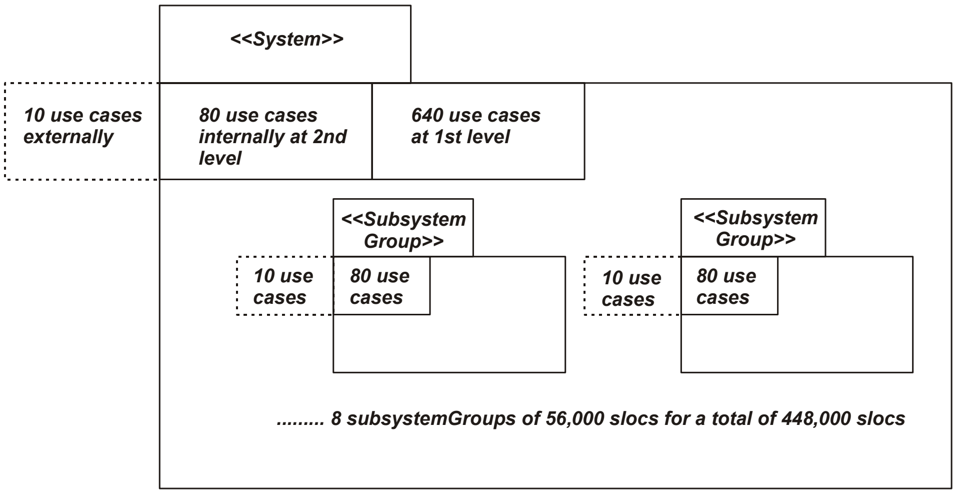
Estimating ranges of size for systems at this level (using the notion of 7 plus or minus 2):

- from 1 subsystemGroup of 5 subsystems of 5 classes totaling 22,000 slocs, to
- 9 subsystemGroups of 7 subsystems each of 7 classes, totaling 370,000 slocs

That's a range of 4–66 external use cases. Again, these are fuzzy limits.

Level 3

At this next level, we have a system (of subsystem groups). At Level 3, use cases are realized by collaborations of subsystem groups:


<<System>>
10 use cases externally
80 use cases internally at 2nd level
640 use cases at 1st level
<<Subsystem Group>>
10 use cases
80 use cases
<<Subsystem Group>>
10 use cases
80 use cases
.... 8 subsystemGroups of 56,000 slocs for a total of 448,000 slocs

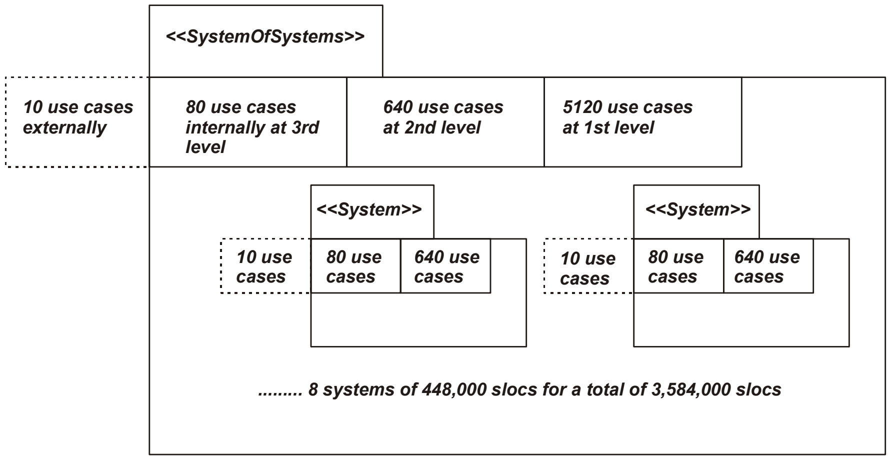
Estimating ranges of size for systems at this level (using the notion of 7 plus or minus 2):

- from 1 system of 5 subsystemGroups of 5 subsystems of 5 classes totaling 110,000 slocs, to
- 9 systems of 7 subsystemGroups each of seven subsystems each of 7 classes, totaling 2,600,000 slocs.

That's a range of 3–58 external use cases. Again, these are fuzzy limits.

Level 4

At the next level, we have a system of systems. At Level 4, use cases are realized by collaborations of systems:


<<SystemOfSystems>>
10 use cases externally
80 use cases internally at 3rd level
640 use cases at 2nd level
5120 use cases at 1st level
<<System>>
10 use cases
80 use cases
640 use cases
<<System>>
10 use cases
80 use cases
640 use cases
.... 8 systems of 448,000 slocs for a total of 3,584,000 slocs

Estimating ranges of size for systems at this level (using the notion of 7 plus or minus 2):

- from 1 system of systems of 5 systems of 5 subsystemGroups of 5 subsystems of 5 classes totaling 540,000 slocs, to
- 9 systems of systems of 7 systems each of 7 subsystemGroups each of 7 subsystems each of 7 classes, totaling 18,000,000 slocs.

That's a range of 2–51 external use cases. Again, these are fuzzy limits. I suppose larger aggregates are possible, but I don't want to think about them!

Effort per Use Case

We can get some insight into effort per use case, by estimating the effort for these nominal sizes at each of the levels. Using the Estimate Professional™ tool[9] (based on COCOMO 2[10] and Putnam's SLIM[11] models), setting the language to C++ (other cost drivers set to nominal) and calculating effort for each of the example system types at each nominal size point (assuming 10 external use cases), gives the results found in Table 1.

Table 1: Effort per Use Case for Various Sample Types

Size (slocs)	Effort hrs/use case simple business system	Effort hrs/use case scientific system	Effort hrs/use case complex command and control system
7000 (L1)	55 (range 40-75)	120 (range 90-160)	260 (range 190-350)
56000 (L2)	820 (range 710-950)	1700 (range 1500-2000)	3300 (range 2900-3900)
448000 (L3)	12000	21000	38000
3584000 (L4)	148000	252000	432000

[9] Software Productivity Center Inc, http://www.spc.ca/ supplies the Estimate Professional tool.

[10] See Boehm81 and http://sunset.usc.edu/COCOMOII/cocomo.html.

[11] See Putnam92.

The ranges shown in Table 1 for Level 1 (L1) and Level 2 (L2) take account of the complexity of an individual use case—estimated by analogy with COCOMO's code complexity matrix. At L2, I believe the variation with complexity will start to be subsumed into the characterization by system type, so that a higher level complex command and control system use case, say, will contain a mix of complexities at a lower level. Plotting these on a log-log scale yields Figure 2.

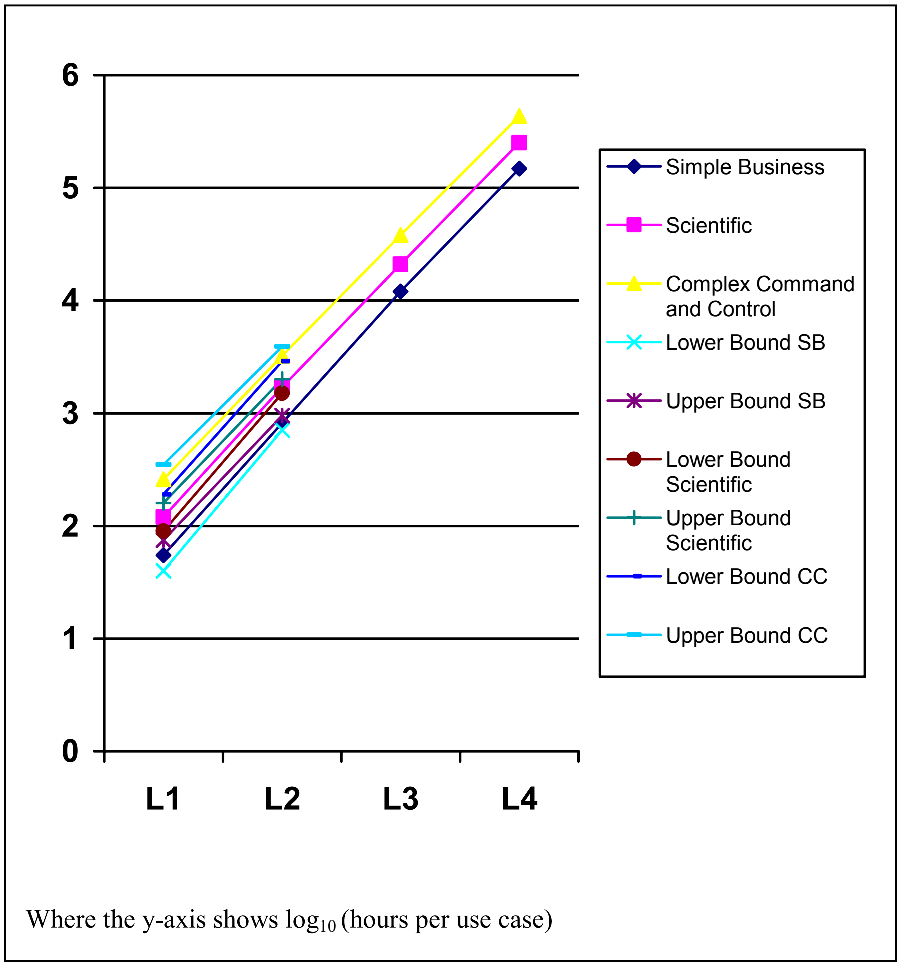


Figure 2: Use Case Effort by Size

It can be seen from this that the old Objectory number of 150–350 hrs/use case ($10^{2.17}$–$10^{2.54}$) fits nicely at L1, i.e. these are use cases that can be realized with collaborations of classes—so there is some justification for this number after all. However, it is not adequate for characterizing all projects during analysis—as a colleague said in an email communication, "it's too 'flat'".

Effort Estimation

Now real systems will not fit into these convenient slots, so to help reason about how a system should be characterized, we can use the fuzzy limits derived along the way and plot them as illustrated in Figure 3.

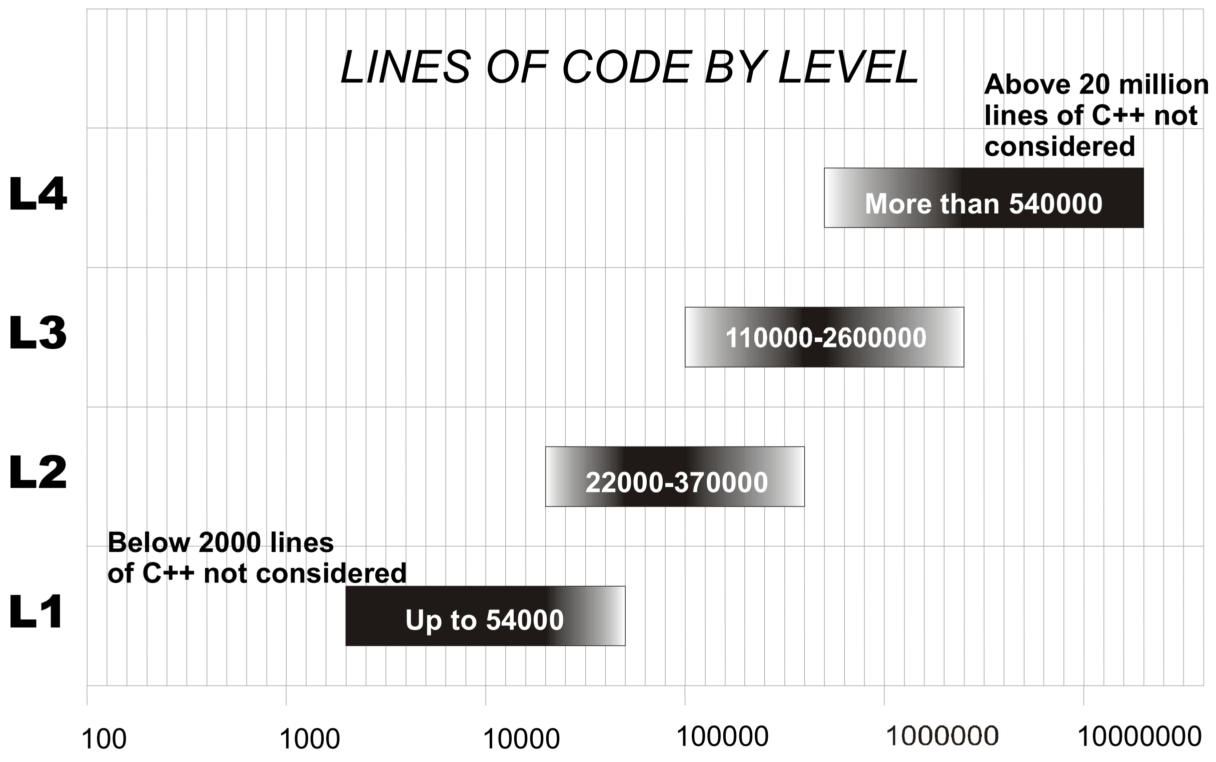


Figure 3: Size Bands for Each Level

From Figure 3, we see that systems up to 22000 slocs are most likely to be described at Level 1, with a use case count of between 2–30. Higher use-case counts at this size may indicate that the granularity of the use cases is too fine.

Between 22000 and 54000 slocs, there could be a mix of Levels 1 and 2 use cases, with a use-case count between 4 (all Level 2) and 76 (all Level 1). As the chart tries to show, these extreme values have low probability.

Between 54000 and 110000 slocs, it's possible that a well-structured system could be described entirely at Level 2, with a use-case count of between 10 and 20; the mix may be L1/L2/L3 (1–160 use cases, with these extremes having ***extremely*** low probability).

Between 110000 and 370000 slocs, there's possibly a mix of Level 2 and Level 3, with a use-case count between 3 (all Level 3) and 66 (all Level 2).

Between 370000 and 540000 slocs, if described entirely at Level 3, there would be a use-case count of between 9 and 12; the mix may be L2/L3/L4 (1–100 use cases, with these extremes having ***extremely*** low probability).

Between 540000 and 2600000 slocs, there is possibly a mix of Level 3 and Level 4, with a use-case count of between 2 (all Level 4) and 60 (all Level 3).

Above 2600000 slocs, the use-case count at Level 4 should rise from ~8.

How Many Use Cases are enough?

Some interesting observations flow from this that support some of the rules of thumb. The question is often asked: "How many use cases are too many?" This question usually means how many are too many ***during requirements capture.*** The answer seems to be that more than ~70, even for the largest system, possibly indicates too fine a granularity prior to design. Between 5–40 is comfortable, but the number by itself, without consideration of level, cannot be used to estimate size and effort. This is the ***initial*** number, appropriate to a particular level. The hundreds of use-case counts will come if a large supersystem is decomposed into systems, and then subsystems, and so on. If use cases were developed until the class level was reached, then the final count could be hundreds or even thousands (say, ~600 for a 140 staff-year project, or something like 15 function points per use case). However, this will not occur as a pure use-case decomposition, independent of design. These use cases arise from the process described in Jacobson97—where use cases at a system level are partitioned into behavior allocated across subsystems, for which lower level use cases can be written (with other subsystems as actors).

Effort Estimation Procedure

So how do we proceed in making an estimate? There are some prerequisites: an estimate based on use cases cannot be made without some understanding for the problem domain, and without ***already having an idea of the proposed system size, and some idea of the architecture, appropriate to the stage at which the estimate is being made.***

This first rough cut at an estimate can be done using expert opinion or slightly more formally by the Wideband Delphi technique (this was invented by the Rand organization in 1948, see Boehm81 for a description). This will allow the estimator to place the system in one of the size bands in Figure 3. This placement will suggest a range for the use-case count, and indicate the level of expression (L1, L1/L2, and so on). The estimator must then decide, based on the current knowledge of the architecture, and the vocabulary of the domain, whether the use cases nicely fit one level, are split discretely, or are a mix of levels (in the way the flow of events is expressed).

From these considerations it should also become apparent if the data is possibly pathological; for example, if the Delphi estimate is 600,000 lines of code (or function point equivalent) and there has been little architectural work, so that not much is known yet about the system structure, Figure 3 suggests that the use-case count should be between 2 (all Level 4) and 14 (all Level 3). If the use-case count is actually 100, then the use cases may have been prematurely decomposed or the Delphi estimate is a long way out.

Continuing this example: if the actual use-case count is 20, and the estimator decides that these are all L3 and, further, that the use-case length is 7 pages on average, and the system is of the complex business type, then the hours per use case (from Figure 2) is 20,000. This has to be multiplied by 7/9 to account for the apparent lower complexity (based on use-case length). So the total effort by this means is 20*20000*(7/9) = ~310,000 staff-hours, or 2050 staff months. According to Estimate Professional, 600,000 lines of C++ code, for a complex business system, requires 1928 staff months. Therefore, in this concocted example there is good agreement.

If the actual use-case count was 5, and the estimator decides these are split 1 at L4 and 4 at level 3, and, further, that the L4 use case is 12 pages and the L3 use cases average 10 pages, then the effort is 1*250,000*12/9+4*21000*(10/9) = ~2800 staff months. This seems to suggest the Delphi estimate perhaps needs to be revisited, although given that a major piece of the system is still only understood at a very high level, the error bounds are greater anyway.

If the original Delphi estimate had been 100,000 lines of C++, the indication from Figure 3 is that the use cases should be at L2 and there should be about 18 of them. If there were actually 20, as in the first example, application of the method without considering the actual use-case level will give a badly flawed result, if the Delphi estimate is badly wrong.

The estimator must check, therefore, that the use cases are really at the suggested level of abstraction (L2) and can be realized by a collaboration of subsystems, and the use cases are not all really at L3—although the Wideband Delphi method isn't usually quite that bad (i.e., predicting 100,000 when the actual is closer to 600,000). The point is, though, that this method of estimation cannot proceed with confidence without the construction of some notional or conceptual architecture, which aligns with the use-case level. For an estimator very experienced in the domain, the model may be a mental one that enables a judgment of level to be made; for a less experienced estimator and team, it is wise to do some architectural modeling to see how well the use cases can be realized at a particular level.

The count for a mixed expression use case (that is, a mix of Level N and Level N+1) should be counted as $n=8^{(\text{fractional distance between the two levels})}$ of the lower bound use-case type. Therefore, a use case assessed at 50% L1 and 50% L2 should be counted as $8^{0.5}$ = 3 L1 use cases to get the overall count. A use case assessed at 30% between L2 and L3 should be counted as $8^{0.3}$ L2 use cases = 2 L2 use cases. A use case assessed at 90% of the way between L2 and L3 should be counted as $8^{0.9}$ = 7 L2 use cases.

Size Adjustment of Table

There is actually a further adjustment that needs to be made to the individual hours/use figures to take account of the overall size—the effort figures are appropriate at each level ***in the context of systems of that size.*** Therefore at L1, in Table 1, 55 hrs per use case will apply when building a system of 7000 slocs. The actual number will depend on the total system size so if the system to be built is, say, 40,000 slocs and there are 57 Level 1 use cases describing it, the effort will not be 55*57 hrs for a simple business system, but $(40/7)^{0.11} * 55 = 66$ hrs/use case. This is based on the COCOMO 2 relationship of size to effort. According to the COCOMO model, Effort = A * $(\text{Size})^{1.11}$, where:

- Size is ksloc
- A will have cost drivers factored in
- Project scale factors are nominal (giving 1.11 for the exponent)

Note that these calculations could be factored into a tool like Estimate Professional to eliminate the calculation burden; they are shown here for completeness.

Therefore the effort per ksloc, or per unit if you will, equals A* $(\text{Size})^{1.11}$/Size, which gives A* $(\text{Size})^{0.11}$, and the ratio of effort/unit at size S1 to the effort/unit at size S2 is $(S1/S2)^{0.11}$.

In addition to the Delphi estimate, the system size can be calculated roughly from the use-case count at the various levels: if there are N1 use cases at Level 1, N2 at Level 2, N3 at Level 3, and N4 at Level 4, then the total size is [(N1/10)*7 + (N2/10)*56 + (N3/10)*448 + (N4/10)*3584] ksloc. And so we can calculate the effort multipliers for each of the effort per use-case figures in Table 1, by dividing this total size by the size for each level (in ksloc) shown in column one of Table 1.

- Therefore, at Level 1 $(0.1*N1 + 0.8*N2 + 6.4*N3 + 51.2*N4)^{0.11}$
- At Level 2 $(0.0125*N1 + 0.1*N2 + 0.8*N3 + 6.4*N4)^{0.11}$
- At Level 3 $(0.00156*N1 + 0.0125*N2 + 0.1*N3 + 0.8*N4)^{0.11}$
- At Level 4 $(0.00002*N1 + 0.00156*N2 + 0.0125* N3 + 0.1*N4)^{0.11}$

Clearly, at Level 4 for example, the number of Level 1 use cases has a tiny effect compared with the number of Level 3 or Level 4.

Summary

A framework for estimation based on use cases has been presented. To make the presentation more concrete, some values were chosen for the framework parameters, which, it is argued, are not wildly in error. As always, such conjecture should be tested against reality and the parameters re-estimated as data is gathered. The framework takes account of the idea of use-case level, size, and complexity for different categories of system, and does not resort to fine-grained functional decomposition. To ease the burden of calculation, it is possible to construct a front end to a tool such as Estimate Professional that provides an alternative method of inputting size, based on use cases.

For comments and feedback on this white paper, please contact John Smith, jsmith@rational.com.

References

1. Armour96: Experiences Measuring Object Oriented System Size with Use Cases, F. Armour, B. Catherwood, et al., Proc. ESCOM, Wilmslow, UK, 1996
2. Boehm81: Software Engineering Economics, Barry W. Boehm, Prentice-Hall, 1981
3. Booch98: The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, 1998
4. Cockburn97: Structuring Use Cases with Goals, Alistair Cockburn, Journal of Object-Oriented Programming, Sept-Oct 1997 and Nov-Dec 1997
5. Douglass99: Doing Hard Time, Bruce Powel Douglass, Addison Wesley, 1999
6. Fetcke97: Mapping the OO-Jacobson Approach into Function Point Analysis, T. Fetcke, A. Abran, et al., Proc. TOOLS USA 97, Santa Barbara, California, 1997
7. Graham95: Migrating to Object Technology, Ian Graham, Addison-Wesley, 1995
8. Graham98: Requirements Engineering and Rapid Development, Ian Graham, Addison-Wesley, 1998
9. Henderson-Sellers96: Object-Oriented Metrics, Brian Henderson-Sellers, Prentice Hall, 1996
10. Hurlbut97: A Survey of Approaches For Describing and Formalizing Use Cases, Russell R. Hurlbut, Technical Report: XPT-TR-97-03, http://www.iit.edu/~rhurlbut/xpt-tr-97-03.pdf
11. Jacobson97: Software Reuse – Architecture, Process and Organization for Business Success, Ivar Jacobson, Martin Griss, Patrik Jonsson, Addison-Wesley/ACM Press, 1997
12. Jones91: Applied Software Measurement, Capers Jones, McGraw-Hill, 1991
13. Karner93: Use Case Points - Resource Estimation for Objectory Projects, Gustav Karner, Objective Systems SF AB (copyright owned by Rational Software), 1993
14. Lorentz94: Object-Oriented Software Metrics, Mark Lorentz, Jeff Kidd, Prentice Hall, 1994
15. Major98: A Qualitative Analysis of Two Requirements Capturing Techniques for Estimating the Size of Object-Oriented Software Projects, Melissa Major and John D. McGregor, Dept. of Computer Science Technical Report 98-002, Clemson University, 1998
16. Minkiewicz96: Estimating Size for Object-Oriented Software, Arlene F. Minkiewicz, http://www.pricesystems.com/foresight/arlepops.htm, 1996
17. Pehrson96: Software Development for the Boeing 777, Ron J. Pehrson, CrossTalk, January 1996
18. Putnam92: Measures for Excellence, Lawrence H. Putnam, Ware Myers, Yourdon Press, 1992
19. Rechtin91: Systems Architecting, Creating & Building Complex Systems, E. Rechtin, Prentice-Hall, 1991
20. Royce98: Software Project Management, Walker Royce, Addison Wesley, 1998
21. RUP99: Rational Unified Process, Rational Software, 1999
22. Stevens98: Systems Engineering – Coping with Complexity, R. Stevens, P. Brook, et al., Prentice Hall, 1998
23. Thomson94: Project Estimation Using an Adaptation of Function Points and Use Cases for OO Projects, N. Thomson, R. Johnson, et al., Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA ’94, 1994

Rational

the software development company

Dual Headquarters:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail: info@rational.com
Web: www.rational.com
International Locations: www.rational.com/worldwide