Dear Dr. Use Case:
What About Function Points and Use Cases?

by **Leslee Probasco**
Rational Software Canada

Note: This is a summary of a recent discussion on the chat_rup forum. My thanks to the main discussion contributors: Davyd Norris, Pan-Wei Ng, and John Smith.

Dear Dr. Use Case,

Recently a customer asked me the following question: "If we could estimate the functional complexity of a use case (e.g., hard, medium, or easy), is there a way to then estimate the number of function points those use cases might have?"

Of course, I had a little trouble with this question. My gut reaction was that use cases and function points do not play in the same space (or, at least, they "play the game" differently). Have you ever dealt with this issue? Does Rational have any documentation on using function points versus use cases?

Please point me in the right direction.

Signed,
Pointless About Use Case Estimations

Dear Pointless,

At first glance, estimating use cases (UCs) using Function Points (FPs) might seem like comparing apples with oranges, because we work so hard to avoid functional decomposition with use cases. Function points rely heavily on the physical layout of the system (for example, numbers of tables and fields) and are therefore predominantly data driven. The goals

of the two methods have some obvious similarities; as with UCs, FPs are defined from a user perspective. The International Function Point Users' Group (IFPUG, at http://www.ifpug.org/) defines an FP as "ýmeasured from a functional, or user, point of view. It is independent of the computer language, development methodology, technology or capability of the project team used to develop the application." But that's where the similarities end. To compare the two methods directly, you would have to base your UC grading on the number of tables, functions, and so on, as per the IFPUG standard.

The Rational Unified Process® (RUP®) contains a paper by John Smith called "The Estimation of Effort and Size Based on Use Cases," which looks at some important techniques for estimating development effort and includes an estimation framework based on use cases. This framework considers the idea of *use-case level*, size, and complexity for different categories of systems. Along with other approaches, it discusses a Use-Case Point (UCP) method based on Function Point Analysis (FPA), referencing Gustav Karner's 1993 M.Sc. thesis on this topic (written while Karner worked at Objectory AB, under the supervision of Ivar Jacobson).

Using Karner's Estimation Technique

Several folks at Rational have been using Karner's technique for a number of years now, with good results. Sun and IBM have also publicly posted that they use this technique and have revised the "fudge factors" (described below) based on their experience; in addition, the technique has been documented in several books. *The technique's main benefit is that it can be performed in your head from a use-case model survey* (i.e., *very* early in the lifecycle, with *very* low precision), before the use cases have even been written, provided you have some idea of how many scenarios are contained within each use case (I always include a list of key scenarios in my brief use-case description).

Basically, Karner's technique is similar to FP techniques in that you:

1. Count key aspects of your requirements to form an *unadjusted point count*.
2. Use several sets of questions about your team and their environment to *create a fudge factor.*
3. Multiply your original count by the fudge factor to come up with *an adjusted point count*, which then translates into a person-hour LOE (Level of Effort) estimate.

Karner proposes using a fudge factor set very similar to the FP method factors but with slightly different weightings, and he proposes 20 person-hours/UCP for LOE estimates. By looking at Actors and use cases for input, you can derive a point count as follows:

1. **Rank Actors** as simple (1 point), average (2 points), or complex (3 points):
    - *Simple*: a machine with a programmable API

- *Average*: either a human with a command line interface or a machine via some protocol (no API written)
- *Complex*: a human with a GUI

2. **Rank use cases** as simple (5 points), average (10 points), or complex (15 points):
   - *Simple*: fewer than 4 key scenarios or execution paths in the UC
   - *Average*: 4 or more key scenarios, but fewer than 8
   - *Complex*: 8 or more key scenarios
3. **Calculate unadjusted use-case point (UUCP) count, a fudge factor, and an adjusted use-case point (AUCP) count.**

   For the system under scrutiny, add up all the points to get the *unadjusted* (UUCP) count.

   Then, multiply by the technical and environmental fudge factors to get the *adjusted* (AUCP) count.

   Note: These counts are COCOMO[1]-like; when using this approach with COCOMO, use the *unadjusted* count (based on the assertion that UUCP has the same "weight" as unadjusted function points, which are fed into COCOMO).
4. **Convert the totals from Step 3 to an LOE estimate** based on calibrations of your team/organization. Use 20 person-hours/AUCP as a start. *Note*: The folks at Sun report that in their experience the rate should be closer to 30 person-hours/AUCP; I have found it's somewhere in the middle, but highly organization dependent.

Step 1 above gives straightforward definitions for ranking Actors, but for UCs (Step 2) you need to apply discrimination to determine what constitutes a "key scenario." A key scenario in this case is the major way a use-case instance can be executed. For the most part, this would correspond to a major alternate flow, but not always. It could be that several alternate flows combine into one key scenario, or that a particular exception flow is very complex and so becomes part of a key scenario. The instruction in Step 2 also assumes that your use cases are leveled (with respect to level of detail) in similar ways to other projects. As mentioned in the RUP, a midsize project of about 10 developers over 6-8 months should have about 30 use cases. This fits with the idea that *an average UC has 12 UCP, and each UCP requires 20-30 hours. That means a total of 240-360 person-hours of effort per use case*. So 30 use cases would require approximately 9,000 staff hours (10 developers for 6 months). Note, however, that a very large project with 100 staff for 20 months would NOT start with 1,000 use cases (pro rata), because of the level issue.

It is important to make sure the UCs are not too decomposed, and, equally important, not too high level. Make sure you are dealing with a *system* use case, not a business use case. The test question I ask is: "Can the key scenarios be realized by collaborations of 7ý2 classes? "This

applies to the analysis level; there could be several more classes when you look at a fully elaborated design-level collaboration. If the number of classes starts to explode, and you start to aggregate the classes into subsystems, then your use cases may be at a different level. Vastly different numbers of use cases will skew your results one way or another, but the method can be recalibrated as described above (by reapplying the estimation to similarly leveled use cases) to take this into account for each organization's style.

Estimates for Simple and Complex Systems

Using Karner's technique to estimate effort for simple and complex systems (as shown below) yields a range of values that correlate well with empirically based figures given in the RUP of about 150-350 hours per use case.

- **Example: Simple System**
  The simplest system (UC rank = 5) would be a human initiating a simple use case driven by a command line interface (Actor rank = 2). Based on the formula specified in Step 3 above, this would give a UUCP count of 2 + 5 = 7 points. Using the formula in Step 4, at 20 person-hours per UCP, this yields about **140 person-hours**.
  Note: A typical fudge factor for a new team would add between 10 and 20 percent to the effort estimate.
- **Example: Complex System**
  A complex system (UC rank = 15) would be a human initiating a complex GUI-driven (Actor rank = 3) use case; this would add up to 18 UUCP or about **360 person-hours**.

Do It in Your Head

As you can see, you can apply this technique in your head as you walk into a project. I find it particularly useful when I get thrown into a problem project headfirst. I do a quick mental check of how many people should be on the team and where in the schedule they should be.

Others have calibrated the technique for their teams with very good results (search the Sun and IBM developer sites to see papers on the use of UCPs). The key point, however, is that *with very little effort, you can use this technique to get a very early gross estimate*. And it will be just as accurate (or inaccurate) as any other method you could use at this early stage in the project.

The best way to use the technique is to do a quick calculation and then move on to more effective methods of estimation, such as actually doing some useful work with your team and seeing how long it takes. Your initial estimates can then be calibrated against these findings and refined as you move further along.

Compensating for the Technique's Deficiencies

Most of the problems I have seen in understanding and applying Karner's

UCP technique revolve around evaluating the complexity of use cases, or rather, defining what is a key scenario. For example, should a use case that allows me to do CRUD (Create, Replace, Update, Delete) be considered as 1 UC with 4 key scenarios, or is it actually 1 use case with 1 key scenario, as the other scenarios are so similar? When such questions arise, I turn to Larry Constantine's idea of an "essential use case." In this context, *essential* does not refer to the crucial use cases in your system, but rather to the essence that defines what each use case is about. You should be able to look at a use case and determine which scenarios shape its very essence, as opposed to those that fill out and complete it. (See Larry Constantine and Lucy A. D. Lockwood's book, *Software for Use*[2], for a further discussion of what constitutes an essential use case.)

Another problem I have with the UCP technique is its vagueness about how many use cases you will have and how granular they should be. There are plenty of debates about what constitutes a use case, and how much it should be refined. For an appropriate use-case count, I typically go by "gut feel" and averages I have cultivated over the years, based on my own and others' experiences. The RUP says that an average IT project (business, not technical) of about 6-8 months and 10-15 staff will consist of somewhere around 30 use cases, and what I see in practice confirms this. One of the largest projects I worked on was a 4-year, 300-person project consisting of around 280 use cases (FP estimates of this project put it at around 9,000-14,000 FPs!).

With respect to the issue of use-case granularity, the smallest useful use case I have seen was only a half page in length, and the largest was more than 120 pages! Before you start yelling, this huge use case was actually very simple: It had a main flow that was 2 pages long and 50 alternate flows that started and ended at exactly the same points. Basically, the use case documented the management of company rules and restrictions, and there were 50 or so different types of rules a user could choose from; average reviewers read the 2-page main flow and then picked a couple of rule types that interested them. During development, the rule types were prioritized, and a few new rule types were added in each iteration (in fact, some low-priority rules never did get implemented).

Yet another problem I have with Karner's technique is that estimates change based on the type of project. For example, the GUI for a Web application causes related Actors to be ranked as complex, as would the geographic mapping GUI in a command and control (C2) project. However, it could be argued that the internals of a Web application are based on well-known component infrastructures (such as .NET or J2EE), so its implementation would be trivial compared to the extremely complex inner workings of a C2 system. In this situation several things come into play. Compared to a Web application, a C2 system will have many more use cases, and each of these will tend to have more key scenarios than Web application uses cases; this increases the UCP for C2 systems. In addition, the technical and environmental complexity of a C2 system greatly increases the fudge factors for the project, making them much higher than those for a corresponding Web application.

Beyond Early Estimates

Although the early estimates you get with this technique can give you a good start, the most important thing to remember is that they are only gross estimates. As you proceed with the project, you need to:

- Factor your own experiences into the mix.
- Start refining your figures as soon as you have more information.

Once you have an Analysis Model available, it is possible to identify boundary, control, and entity classes, or, even better, design subsystems. You can estimate the effort required to implement these by using analogous figures from past projects.[3] (At this level of analysis detail, you can also start employing the techniques presented in a paper entitled "The Estimation of Effort Based on Use Cases"[4] by Rational's John Smith, or use well-known techniques such as those in COCOMO II.[5]

On the other hand, rather than coming up with a better way to nail down costs before you start the project, perhaps you should spend most of your effort on changing the organization's/team's underlying attitude, so you can avoid premature estimation in the first place. Premature estimation is a very nasty condition that leads managers to commit the team to unrealistic budgets, which results in everyone on the project becoming hot under the collar, resources being spent before completion, and requirements that are only half satisfied. It is possible and potentially useful, however, to make estimates at any time, provided you recognize the attendant error bounds on your estimate. Also, the project manager should -- if required to make budgetary projections-- either provide for contingencies or establish a scope management regime that will prune functionality to fit the budget. This is the real world, after all.

Hope this helps.

Usefully yours,
Dr. Use Case

Notes

1 COCOMO is the Constructive Cost Model, originally developed by Dr. Barry Boehm and described in his classic work *Software Engineering Economics*, published in 1981 by Prentice-Hall.

2 Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison Wesley, 1999.

3 See Joe Marasco's article on "Commitment" in the May 2002 issue of *The Rational Edge*.

4 Available online at
http://www.rational.com/products/whitepapers/finalTP171.jsp

[5] See http://sunset.usc.edu/research/COCOMOII/.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!