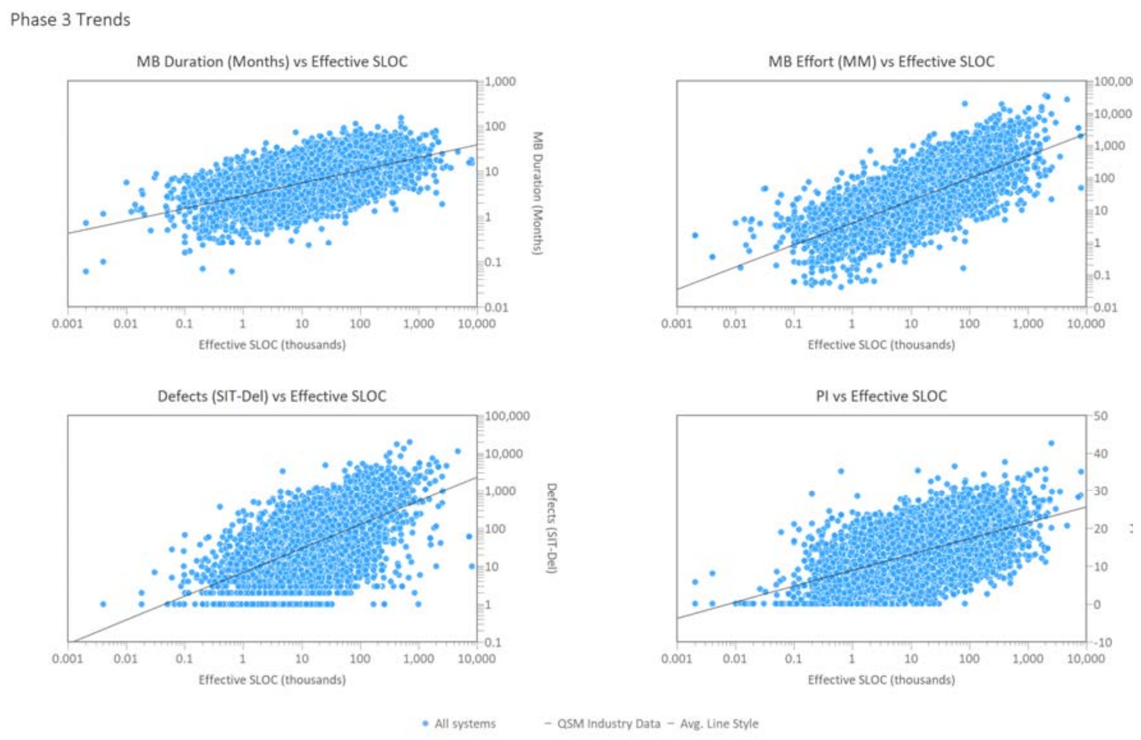
GEARING FACTORS

A FLEXIBLE SIZING APPROACH

MEASURING SOFTWARE SIZE

Estimating the scope of a proposed system is one of the most challenging aspects of software project planning. Size estimation is often viewed as a difficult and thankless job, so it's hardly surprising that many project managers skip size-based software estimates in favor of level of effort or task-based estimation. This is unfortunate, because decades of empirical data show a strong correlation between the **delivered size** of a software project (whether captured in logical lines of code, function points, story points, requirements, or use cases) and its **final schedule, effort, and quality**. The graphs below plot each of the major management metrics against delivered system size for 13,000 completed software projects. All – without exception – increase with software size.



Intuitively, it makes sense that a project's schedule, effort, cost, and defects should be directly related to the quantity of delivered features. High-level and detailed design, coding, package configurations/customizations, quality assurance, documentation, and release management comprise the "work" required to design, implement, test, and deliver each individual feature. As more and more features are added to a release, the total effort and time required to complete the project increases as well. So do the average number of defects!

Size isn't of interest only to estimators. Software estimation, productivity measurement, and benchmarking all rely on the same well established set of software metrics. For decades, these core measures - size, time, effort, and defects - have been used to support a broad range of management decisions. Organizations measure their projects to better predict and control the costs (in time, effort, and money) associated with various management tradeoffs. But there are also **dramatic quality consequences** associated with compressing schedules and piling on staff to meet market deadlines.

In a very real sense, a project's effort outlay, schedule, and reliability depend upon assigning the right resources to the project before it begins missing deadlines. But how can managers plan efficiently if they don't know how much work is needed to translate a given set of requirements into executable code? It is tempting to think of software development work in terms of the effort or resources applied to the project, but this formula puts the cart before the horse. From an estimation perspective, effort (or staff over time) is an *output*. It can help predict how much the project will cost, but not the amount of work needed to implement the requirements (or the speed with which those requirements can be converted to software).

This is why measuring project size is so important.

Without a notion of functional size, it is difficult to negotiate realistic schedules consistent with your organization's proven ability to deliver software. Over four decades of collecting and analyzing completed software projects show that most software metrics increase exponentially as the volume of delivered functionality grows. Unfortunately, human beings think and estimate using simple, linear rules of thumb while software data exhibits strikingly non-linear relationships between size and effort, schedule, and quality. With a little history (and the ability to place an estimate in the correct size regime) managers can empirically show how unlikely it is that a 10 person team that successfully delivered a 150,000 ESLOC project over six months will deliver half the functionality in half the time. *The data makes their argument for them.*

Data driven estimation allows managers to sanity check current plans against past performance or industry trends and negotiate achievable outcomes based on a realistic assessment of how much functionality can be built with a given time frame and resource profile.

ESTIMATING SIZE

It's all very well to say that project managers should measure size. But without a method that is simple to use, repeatable, and above all *practical*, size estimation is unlikely to gain widespread acceptance within an organization. With the text based programming languages of the past, measuring system size was a fairly straightforward process. Source Lines of Code (or SLOC) were easily measured at the end of a project via text export and automated code counters. The downside of using SLOC as an estimation measure is that code counts have little meaning to nontechnical personnel and customers. Without an empirical baseline, it can be difficult to draw connections between the business requirements (often, the only convenient size measure at the time of estimation) and final code counts.

This translation problem has only been exacerbated by the move from text- and procedure-based programming languages to today's object oriented, package-driven, and GUI design environments. Nth-generation development tools don't always lend themselves readily to SLOC-based sizing methods. These days, developers may never write a single line of code. They create software by configuring objects and fields or diagramming relationships with sophisticated graphical tools. Bridging the gap from more abstract software components to finished application size is best accomplished by breaking the work to be accomplished into a series of steps and then relating each set of steps back to a known quantity.

Depending on the technology chosen (or how the project team solves technical issues associated with the project) the "steps" needed to implement a given set of business or technical requirements can be represented by a variety of size measures: function points, epics/stories/story points, web pages, reports, configurable database fields, scripts, diagrams, or SQL queries. Some steps involve writing actual code while others require development staff to drag and drop elements or set properties via a graphical interface.

Depending on the information available, estimates can be detailed or high level. A detailed size estimate might decompose the work into groups of related tasks:

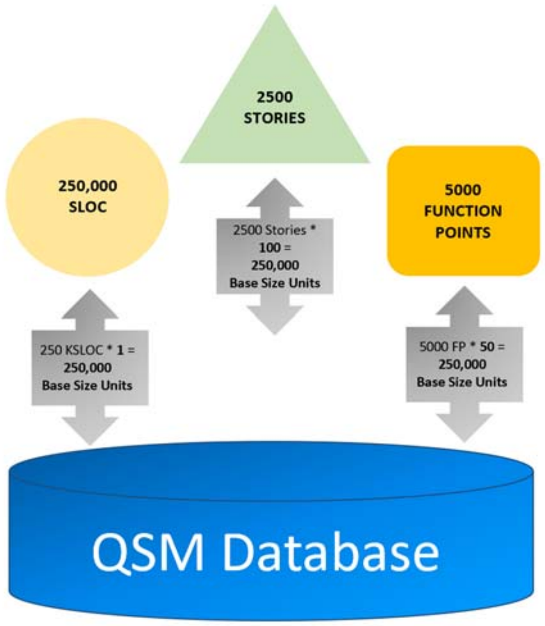
- A set of scripts that migrates data from an existing application to the new platform and performs needed transformations.
- A GUI front end designed by dragging, dropping, and configuring screen elements (screens).
- A set of business rules, reports, and queries.

Another estimator might size the same system using a single abstract size measure that maps to the entire system. Function points, requirements, or Agile epics/stories/story points are often used for this purpose. The estimator is free to choose the method that best suits the information he or she has on hand at the time the estimate is compiled.

Regardless of the method chosen, comparing or combining different sizing components is impossible without first identifying some sort of common denominator (a 'conversion' or 'gearing' factor) that tells the estimator how big they are, relative to each other. Decomposing system size into smaller, abstract size chunks and using a single conversion unit to "gear" these differing size units to a common point of reference allows estimators and project teams the flexibility to *describe the project in terms of the work they will perform* rather than dictating a rigid, one-size-fits-all approach. Once the project is completed, the conversion (or "gearing") factor facilitates meaningful comparisons between projects measured in different functional size units.

The Base Size Unit

QSM calls this common denominator the Base Size Unit. In the past, SLOC was a nearly universal measure of work for software projects spanning different technologies, languages, and development paradigms. But the advent of modern diagramming tools, GUI languages, and programming environments makes lines of code less useful as a measure of work performed. Developers who use diagramming tools may find that a combination of GUI actions better represents the work needed to translate a given set of requirements into software. Those who spend their time configuring database tables can identify the smallest unit of work applicable to database construction and build from there.


2500
STORIES
250,000
SLOC
5000
FUNCTION
POINTS
2500 Stories * 100 = 250,000 Base Size Units
250 KSLOC * 1 = 250,000 Base Size Units
5000 FP * 50 = 250,000 Base Size Units
QSM Database

It doesn't matter what the Base Size Unit is called. What matters is that estimators identify the high level programming tasks (or steps) to be performed, then decompose each step until it requires *approximately* the same amount of time and effort as writing an executable line of code. This is an idea most developers understand intuitively, since even in GUI or package implementation environments some code must still be written by hand. The goal is to preserve a common frame of reference while allowing users to choose the sizing method that most accurately reflects the actual work being performed: translating *abstract* requirements into a *concrete*, functioning software system.

Calculating Gearing Factors from Completed Projects

One of the best ways to derive gearing factors is from completed software project data. If a single size component is used, gearing factors can be **calculated** at the end of a project and the resulting factors used to estimate new projects. If multiple components are used, gearing factors can also be **estimated** during the sizing process and confirmed by **sampling** while the project is underway or once it finishes.

For completed projects sized using a single size component (examples: Function Points, stories, requirements, or use cases), the gearing factor is best determined by running an automated code counter on the software product and dividing the LOC count by the number of size components delivered in the final product. For **Stories**, if your Base Size Unit is SLOC, the gearing factor is the average number of lines of code per Story. This figure is obtained by dividing the effective (new_LOC + modified_ LOC) count from a few comparable completed projects by the delivered/final story count from each project.

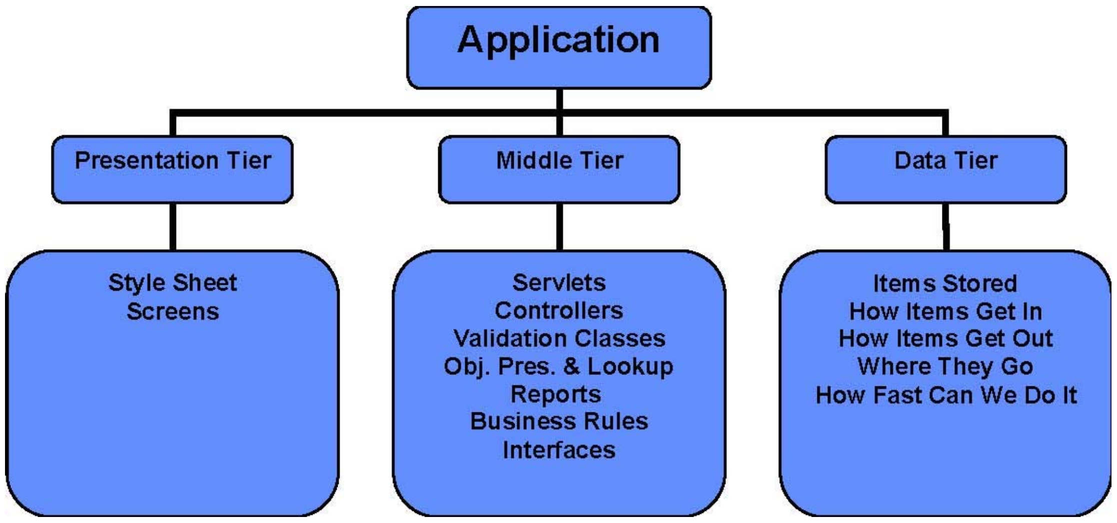
For a single language project sized in **function points**, the calculation would be similarly straightforward. The calculation for a 100,000 line of code project with a final function point count of 2500 would look like this:

Final effective SLOC count/ Final effective function point count
= 100,000/2500
= 40 SLOC/FP

For mixed language projects, this process is a bit more complex. When calculating gearing factors to be used for future estimates, care should be taken to use only projects written solely or primarily (85% or more) in the language of interest. For obvious reasons, calculating a function point language gearing factor for Java from a project that is only 5% Java will not result in an accurate gearing factor for that language. For mixed language projects in which the mix of languages is well known, relative percentages can be applied to the overall gearing factor to yield estimated language gearing factors for each constituent language. It is obviously preferable, however, to use single language projects to calculate language-specific gearing factors.

Estimated or Sampled Gearing Factors

If gearing factors are not available from your completed projects, they can be derived during the estimation process by breaking the work to be accomplished into discrete steps or components and dividing each step or component into complexity bins (each with an associated gearing factor that ties it back to the basic unit of work). For a COTS package implementation, this process might begin with a high level architecture view that flows into a more detailed breakdown of how requirements are implemented in each tier of the finished application.



In the presentation tier of the example pictured above, the presentation layer items are identified (a **style sheet** that standardizes the look and feel of each user page and **update screens** that allow users to log in, display and update their user profile, and manipulate complex data arrays). These "steps" or size components are then logically decomposed into **complexity bins (low, average, high)** and mapped to their **associated technologies**. In the first example (a simple login screen implemented in JSP) the Base Size Units represent the work required to configure fields on a login screen.

Example 1: Simple Login screen implemented in JSP

# Fields (2) *

- **# User Actions (1 per field)+**
- **# Checks (Low 2, Average 3, High 5)**

The second example (a style sheet design element), looks more like what we might expect for traditional lines of code, though it may in fact have been produced in a GUI environment:

Example 2: Average 'look and feel' style sheet design element (HTML or XML):

Average or Expected Sizes
(to get Low/High, add +/- 20%)

- **Header = 30 HTML**
- **Footer = 8 HTML**
- **Navigation = 12 HTML**

When sizing the middle tier, a variety of different components requires a flexible approach. A **complex report** that allows users to drill down to derive information is sized as follows:

Tables:

- **6-10 Tables * 6 Definition Steps + Calculations & Control Breaks:**

- 12-20 Fields * 6 Definition Steps

Sizing a **complex controller** yields the following calculation:

# of Fields (Low 5 Fields, Average 7 Fields, High 10 Fields), *

- **4 Methods per Field**
- **up to 2 Java Beans**

The work involved in **data persistence and lookup** is estimated by breaking this task into its simplest form:

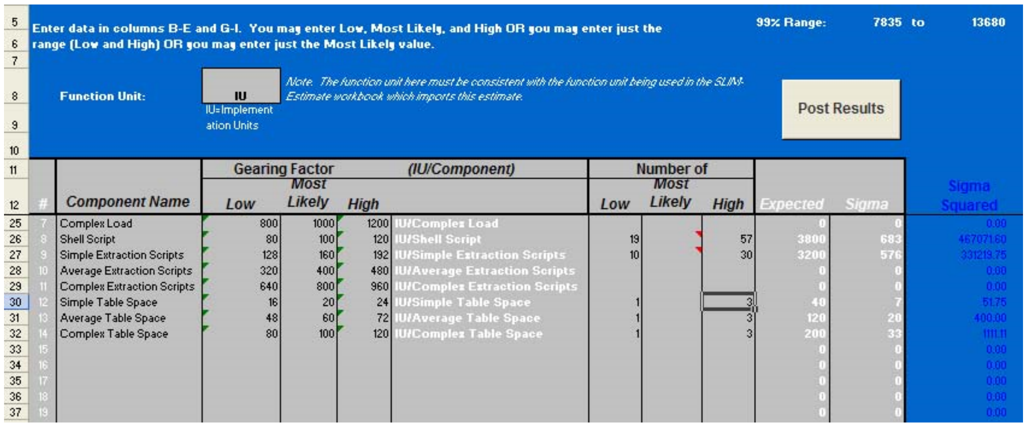
- **1 Lookup Statement +**
- **# of Fields to Map in Single Table**
  - **– Low 15, Average 100, High 200**

More complex lookups are accommodated by adjusting the number of lookups and multiplying the average number of fields in a table by the number of tables to be queried. The beauty of tying the Base Size Unit to a rough line of code equivalent is that it allows development teams the freedom to ignore code, use code exclusively, or combine GUI and SLOC estimates, as in this **interface** example:

- **13-50 Data Elements +**
- **(6-20 Data Translations * 10 ESLOC per Translation) +**
- **150-300 ESLOC (Code to filter the data)**

Regardless of the task, the process of deriving the gearing factor is the same. The estimator begins at the highest level and logically walks through the process of creating each component or "step", asking questions such as, "How do we create a simple login screen?" "On average, how many fields does a simple login screen contain? Do you have to configure each field? If so, how many configurations/properties (on average) must be performed per field?"

Often the answer to these questions will be a range (high, medium, and low) rather than a single number. This is fine, because it allows estimators to determine an expected value and uncertainty range for each size estimate. Once the individual low, most likely, and high estimated gearing factors for each sizing component have been rolled up, they can be loaded into a sizing spreadsheet to speed up future size estimates and encourage standardization across projects. The example below shows the roll up for the data tier of the example shown earlier:



Enter data in columns B-E and G-I. You may enter Low, Most Likely, and High OR you may enter just the range (Low and High) OR you may enter just the Most Likely value.

99% Range: 7835 to 13680

Function Unit: IU
IU=Implementation Units

Note. The function unit here must be consistent with the function unit being used in the SLIM-Estimate workbook which imports this estimate.

Post Results

		Gearing Factor			(IU/Component)	Number of					
#	Component Name	Low	Most Likely	High		Low	Most Likely	High	Expected	Sigma	Sigma Squared
7	Complex Load	800	1000	1200	IU/Complex Load				0	0	0.00
8	Shell Script	80	100	120	IU/Shell Script	19		57	3800	683	467071.60
9	Simple Extraction Scripts	128	160	192	IU/Simple Extraction Scripts	10		30	3200	576	331219.75
10	Average Extraction Scripts	320	400	480	IU/Average Extraction Scripts				0	0	0.00
11	Complex Extraction Scripts	640	800	960	IU/Complex Extraction Scripts				0	0	0.00
12	Simple Table Space	16	20	24	IU/Simple Table Space	1		3	40	7	51.75
13	Average Table Space	48	60	72	IU/Average Table Space	1		3	120	20	400.00
14	Complex Table Space	80	100	120	IU/Complex Table Space	1		3	200	33	1111.11
15									0	0	0.00
16									0	0	0.00
17									0	0	0.00
18									0	0	0.00
19									0	0	0.00

If your programming environment allows text exports of code, you can verify and refine the initial estimates by sampling code for a few completed size elements and comparing these code counts to your estimates. When code counts are not available, another way to empirically confirm your size estimates is to compare your estimated gearing factors to one or more completed components of the same type, using the same "steps-based" decomposition technique. So (for example), you might identify several specific instances of a complex report and compare the steps *actually* required to create each report to your estimated steps. Earlier, a complex report was estimated thusly:

Tables:

- **6-10 Tables * 6 Definition Steps + Calculations & Control Breaks:**
    - **12-20 Fields * 6 Definition Steps**

In this case, you would start with a completed complex report and reconstruct or count the *actual* number of tables and fields involved, verify the *actual* steps required to define each one, and compare the rolled up "gearing factor" for one or more complex reports to your original estimate.

Remember: *the goal is not perfection*, but the creation of *a practical, consistent, and repeatable process* that simplifies future calibration and estimation of similar projects. A line of SQL query code might be more complex than a line written in Python, but when development work is reduced to *the smallest identifiable and practical* unit of work, these differences will be minimized. The important thing to realize is so long as you are consistent in your measurement approach, differences in complexity will be reflected in the Productivity Index. Complex projects of the same size will take longer to build and will exhibit lower average PIs, but when these projects are used to estimate future projects of a similar nature, their calibrated PIs will automatically "build in" the right amount of time and effort going forward since their increased complexity is reflected in the final project PI.

Measurement is never a perfect endeavor but we learn more about the interplay between various project metrics when we measure *all* dimensions of a software project than when we count on incomplete measurement, intuition, expert judgment, or linear rules of thumb for guidance. Armed with a few simple questions and the power of completed project data, project managers will be in a much stronger position when it comes time to negotiate a few extra weeks of schedule flexibility or a reduction in delivered functionality.

Kate Armel is the Director of Testing, Training, and Technical Support for Quantitative Software Management, Inc. She has over two decades of experience in technical writing, metrics research and analysis, and assisting Fortune 1000 firms with software estimation, tracking, and benchmarking.