Reconstructing Completed Projects

Collecting historic data, either for benchmarking purposes or for use in calibrating future software estimates, can seem a daunting task without a clear and easy to follow set of guidelines. Fortunately, if a few simple and practical rules are followed, most organizations find the hurdles to successful reconstruction of completed projects are easily overcome. This document describes a method for jump starting the data collection process with minimum effort to get maximum value.

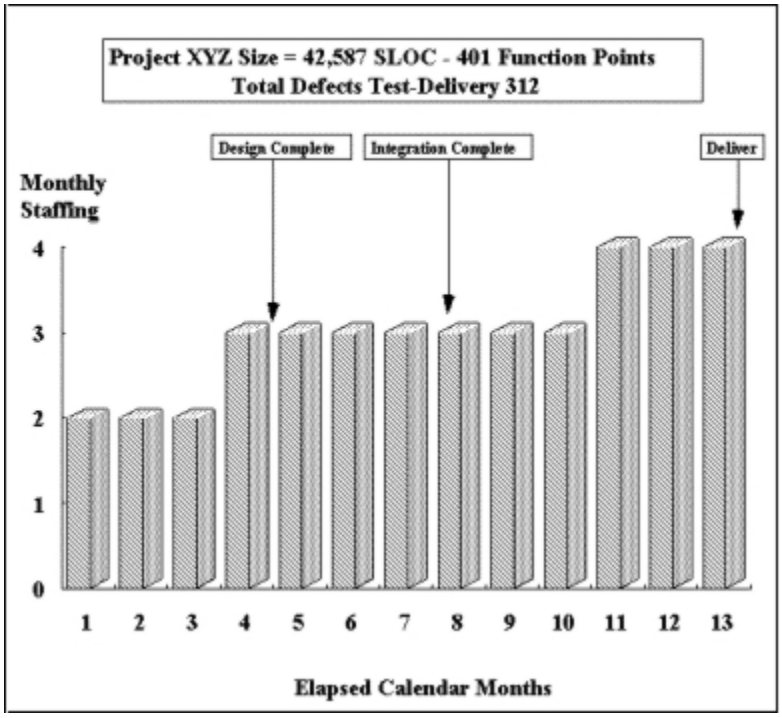
The basic data requirements for **calibrating estimates and assessing productivity** are:

- **Size** (a count of some size unit, accompanied by a gearing factor if needed)
- **Phase 3 Schedule** (start/end dates and/or duration in months)
- **Phase 3 Effort** (total effort in hours, days, weeks, months, or years)
- **Defects**

The best way to derive schedule and effort for completed projects is to reconstruct the project staffing profile. Even if the required data is easily obtainable, the interview process itself lends valuable insight into the data. If effort information is not readily available, a project manager can reconstruct the project's staffing and effort profile in 10-20 minutes via a simple interview process.

The Project Interview

On a piece of paper, draw a simple graph with elapsed calendar time in months on the horizontal axis as shown in the illustration. The vertical axis will be full time equivalent staffing (the total skilled manpower required to design, build, integrate, test, document, and manage the project) per month.



Ask "When did the project start?" and "How many people did you have working on it?" Next, "How long did you remain at that staffing level?" As the interviewee responds, sketch in the first staffing increment, then continue reconstructing staffing increments until the entire profile is complete.

The next step is to ask when major milestone events occurred and mark them clearly on the time line (this will be help you determine where the major phases begin and end).

See the section on QSM Default Phase Definitions for more information on mapping the lifecycle.

Add up the total effort under the staffing profile by integrating the staffing curve. For example, **(2 people for 3 months = 6 staff months)** plus **(3 people for 7 months = 21 staff months)** plus **(4 people for 3 months = 12 staff months).** Add the figures up like this: **(6+21+12) = 39 staff months of effort over (3+7+3) = 13 calendar months.** Your effort and schedule profile is complete. If you are used to seeing effort in person hours rather than person months, simply multiply the total person months of effort by whatever number the project (or your organization) considers to be the average number of person hours in the accounting month.

Next, extract the size of the system by asking for the new and modified source code or function points (or whatever sizing metric is being used). If you are using a function unit (function points, screens, tables, dialogs, etc.) other than lines of code, try to determine the average number of basic work units per size unit. If you need help in determining a gearing factor, see the Sizing section of the User Manual for additional guidance.

Finally, ask for the total number of defects found from the start of system level testing (the point at which all subsystems have been integrated. This typically starts at approximately 71% of the Build and Test phase and continues until the software product is verified as complete by the software developer and delivered to the first external installation. Also, ask for the Mean Time to Defect, or MTTD (average amount of time in days that the software was able to operate until encountering a defect) in the first month of operation. This is all the minimum information needed to calibrate the project.

Mapping Your Lifecycle to the QSM Phase Definitions

There are nearly as many lifecycle methodologies as organizations that embrace them. To make matters even more complicated, even organizations that use the same methodologies often implement them differently; each applying their own terminology and slicing and dicing the work to be done in unique ways. This creates a challenge for analysts seeking to benchmark or measure across an industry that lacks a single, unifying standard or set of measures.

Rather than create yet another level of complexity, QSM has mapped the activities that occur in every software development to four high level sets of activities, or “phases”. At a high level, these phases map to the traditional set of activities common to all software projects:

- **Concept Definition** outlines the project plans and requirements and completes with a go/no-go decision.
- **Requirements & Design** analyzes and fleshes out the requirements.
- **Build & Test** designs, builds, tests, and implements the software.
- **Perfective Maintenance** finalizes the newly installed software and removes any latent defects.

This process has been jokingly described as “**Deciding** to do something, **Determining** what to do, **Doing it**, and **Cleaning up** afterwards”.

The precise words used to describe these activities are not important. What matters is that this approach is both practical and flexible. QSM didn’t set out to describe waterfall, iterative, agile, Rational, or any other method; those are all micro views of a more general process in which certain elements are usually present. Our approach addresses fundamental issues every project faces, but though these activities are common to all software projects, in practice not all of these activities will be recorded for every software project.

In fact, our database shows many activities are routinely handled by outside groups. Frequently the initial go/no-go decision is made by a group which then has no further involvement with the project. Post-implementation support is often performed by a production support team. Overall, about:

- **83%** of projects in our database include **Requirements & Design** and **Build & Test** data.
- **40%** report **Concept Definition** data
- **36%** included effort and time for **Post-implementation** support (**Perfective Maintenance**).

When deciding which phases to capture historical data for post-implementation, **practicality is often the best guide.** Ask: "What activities do we most need to estimate and/or benchmark?" and **focus your data collection activities accordingly**. If the go/no-go decision is handled externally or there is no consistent process for this phase, it may more sensible to concentrate data collection efforts on phases for which you have consistent and readily available data.

Though many different lifecycle models exist, all of these models have certain elements in common. Moreover, these common elements occur in the same general order. For each developed software element, **Requirements** definition begins before **Design**, which precedes **Coding**, which is followed by **Integration and testing** (RDCI). The SLIM model has been used successfully for many years to model Waterfall, Spiral, Incremental, and many other development methodologies.

It is not essential that the program being estimated or reconstructed be developed according to the default QSM life cycle. We can analyze any development style and map its activities to one or more high-level elements that may be overlapped, staffed, and iterated in various ways. Once your high-level activities have been mapped to the four QSM phases, the Milestones, WBS, and Skill Categories tabs in SLIM-Estimate can be used to identify incremental builds or other more detailed activities within each high-level phase.

This is where collecting and analyzing your historic data can dramatically improve the accuracy of your estimates. By analyzing the way your organization develops and deploys software, you can construct lifecycle templates tailored to your development environment and methods. Listed below are some general guidelines for dealing with software life cycle approaches that are unique, but which still fit into QSM's software measurement approach when some basic rules are applied for categorizing the life cycle phases.

Rapid-application Prototypes

In prototype or rapid application environments where the prototype will be used in the delivered architecture, include all of the time and effort for each of the "prototype cycles" performed (design, code, integration, and user approval) as well as the time and effort to integrate and stabilize the final product in Phase 3. If possible, determine the time and effort for Phase 2 by taking a percentage of the total time and effort spent defining software requirements.

Enhancements, Modifications, and Large-scale Perfective Maintenance

For enhancements, modifications, and large-scale maintenance efforts that are beyond the scope of the parent project's Phase 4, include the time and effort for analysis of potential modifications/changes in Phase 2. Design modifications, coding and integration should be included in Phase 3.

Throw-away Prototypes

In environments where prototypes are built to stabilize requirements and are subsequently discarded, include the time and effort spent building and verifying the prototypes in Phase 2. Do not count the discarded software in the delivered code count.

The QSM Default Phase Definitions

Phase 1: Concept Definition.

The earliest phase in the software life cycle where complete and consistent requirements and top-level, feasible plans for meeting them are developed.

The objectives of this phase are to develop a complete and technically feasible set of requirements for the system and to formulate the top-level approach and plan for their implementation. Typical products of these activities include:

- a system specification, statement of need, or list of capabilities that the user or the marketing organization expects from the system, or a marketing specification;
- a feasibility study report or an assessment of whether the need can be met with the available technology in a timely and economic manner; and
- a set of plans documenting the project management approach to development, quality assurance, configuration management, verification, etc.

This phase is complete when it is determined that building this system is feasible. A system-level requirements review may be held at the completion of this phase to determine the scope of the effort and the approach for Phase 2.

Phase 2: Requirements & Design.

Phase 2 develops a technically feasible, modular design within the scope of the system requirements.

The objectives of this phase are to complete the system-level design by choosing the appropriate constituent technologies (hardware, software, etc.) and to allocate each system-level requirement to the appropriate technology. Software requirements are defined. The software top-level architecture is defined. Typical products include the following:

- specifications defining the interfaces between the system-level components;
- software requirements specifications describing the inputs, processing (logic and/or algorithms), and outputs; and
- updated project plans.

A design review (e.g., PDR) may be held during this phase to baseline the system design, software requirements, and software top-level architecture. The Concept Definition Phase normally overlaps the start of Phase 3 as the top-level software design is iterated and refined.

Phase 3: Construct & Test.

This phase produces a working system that implements the system specifications and meets system requirements in terms of performance and reliability. It typically begins with detailed logic design, continues through coding, unit testing, integration and system testing, and ends at full operational capability. The Construct & Test phase seeks to implement the software requirements as defined in Phase 2. Once the software requirements have been base lined, the activities in Phase 3 implement these requirements through software detail design, coding, and integration. Typical products include:

- design documentation;
- verification (inspection and/or test) procedures and results;
- user manuals and other operating documentation;
- maintenance manuals;
- acceptance test procedures and reports; and
- a full-functionality software product that has achieved at least 95% reliability and is acceptable for initial use in the customer's environment.

The phase starts on the date when the very first module design is started. The phase ends on the date when it is possible, for the first time, to deliver a fully functional system with 95% of the total defects identified. Subjectively, this is the point in the program where a full-functionality system can be delivered with sufficient reliability so as not to be returned by the customer.

Phase 4: Perfective Maintenance.

The phase that usually coincides with the operations phase. It may include correcting errors revealed during system operation or enhancing the system to adapt to new user requirements, changes in the environment, and new hardware.

The objective of this phase is to provide the customer base with product support once the system is operational. In some life cycles, this phase includes continued testing to increase reliability and provide quality assurance to customers and certification authorities. During this phase, residual bugs in the system are corrected. In addition, development of later releases and functional enhancements may be performed to meet the new requirements of a changing world. The principal activities of this phase include:

- increased reliability and assurance testing;
- correction of latent defects;
- new enhancements;
- modification and tuning of the system; and
- operational support.

In some life cycles, this phase is complete when management decides that the system is no longer of practical use and cuts off additional resources. Some time before this happens, management may seek more improvements and initiate a feasibility study for a new system to replace it. In other life cycles, this phase is complete when some sort of certification occurs. Evolution of the product beyond this point typically spawns new projects.

Phase Staffing Shapes

For each active phase, you should pick the staffing profile that best matches your historical effort buildup:

- **Level Load:** maintains a constant staffing level throughout the phase.
- **Front Load Rayleigh:** builds to a peak at about 40% of the phase schedule, then tapers down.
- **Medium Front Load Rayleigh:** peaks closer to the middle of the phase before tapering down.
- **Medium Rear Load Rayleigh:** peaks at about 75% of the phase schedule.
- **Rear Load Rayleigh:** (Phase 3 only) reaches a staffing peak at the end of Phase 3.
- **Default Rayleigh:** (Phase 3 only) determines the staffing shape based on the size of the application. QSM has found that small projects (less than 18,000 lines of code) tend to have a Front Load Rayleigh profile while larger systems (greater than 100,000 lines of code) typically reach peak staffing towards the end of phase 3. For very small systems (3 to 6 months in duration with a peak staff of 1-3 persons), a level load profile is more appropriate.
- **Exponential:** (Phase 4 only) gives the most rapid drop off of staffing from the end of Phase 3.
- **Stair Step:** (Phase 4 only) begins at about half of the staffing level from the end of Phase 3 and stair steps down.
- **Straight Line:** (Phase 4 only) a straight-line decrease in staffing from the end of Phase 3.
- **Rayleigh:** (Phase 4 only) a natural tailing off/continuation of any Phase 3 Rayleigh curve. Not valid if the selected Phase 3 staffing shape is Level Load.

QSM Default Application Domains

The application type tells us something about the complexity level of a project. Project data in the QSM database is stratified into nine application complexity domains, each with its own set of characteristics, size ranges, and productivity indices. Classifying completed projects by application domain will help you make apples to apples comparisons when analyzing project metrics or choosing appropriate projects to calibrate new estimates.

- **Microcode & Firmware.** Software that is the architecture of a new piece of hardware or software that is burned into silicon and delivered as part of a hardware product. This software is the most complex because it must be compact, efficient, and extremely reliable. Examples are: boot ROMs, fundamental instruction set for a computer architecture, controllers for microwave ovens, instrumentation, etc. Often written in machine language for compactness and speed.
- **Real Time.** Software that must operate close to the processing limits of the CPU. This is interrupt-driven software and is generally written in C, Ada or Assembly language. It generally operates with a very small executive for an operating system interface to the basic processor. Typical examples are military systems like radar, signal processors, missile guidance systems, etc.
- **Avionic.** Software that is on-board and controls the flight and operation of the aircraft.
- **System Software.** Layers of software that sit between the hardware and applications programs. Examples are operating systems (DOS, UNIX, VMS, etc.), GUI's (graphical user interfaces, Windows, Xwindows etc.), Executives or Data Base Management systems, Network products, and Image processing products.
- **Command & Control.** Software that allows humans to manage a dynamic situation and respond in human real time. Examples are battlefield command systems, SAC command and control systems, telephone network control systems, government disaster response systems, military intelligence systems, electric utility power control systems, and air traffic control systems.
- **Telecommunications.** Software that facilitates the transmission of information from one physical location to another. Examples are telephone switches, transmission systems, modem communication products, fax communication products, and satellite communications products.
- **Scientific.** Software that involves significant computations and scientific analysis. Examples are statistical analysis systems, graphics products, and data reduction systems. This type of software is often sensor driven with data capture schemes to accumulate data (from a spacecraft, say) then followed by extensive data analysis. Frequently written in FORTRAN.
- **Process Control.** Software that controls an automated system. Generally sensor driven. Examples are software that runs a nuclear power plant, or software that runs an oil refinery, or a petrochemical plant.
- **Business.** Software that automates a common business function. Examples are payroll, financial transactions, personnel, order entry, inventory management, materials handling, web, warranty, and maintenance products. May involve batch processing of rollup information summaries, or software that performs transaction processing at the supermarket, teller machine, etc. Often involves telecommunications (LANs) and database transactions.