chapter 5

Software Production Metrics

Choosing Metrics

When selecting metrics for control of a system, it is essential to focus on simplicity and relevance to the system goal. Metrics should ideally be self-generating and should provide leading or predictive indication of the system performance rather than lagging or reactive performance [Reinertsen 1997, pp. 197–217].

Agile Software Production Metrics

In the case of the software production system, it is important that the production metrics reflect and support the financial metrics from Chapter 2. Production metrics must be driven from economics. If the process control metrics do not directly relate to the economics and financial goals of the system, then they are not appropriate.

The quantity of the system input (ideas) relates to the Investment. The Inventory level within the system can demonstrate the current location of the Investment and its progress towards becoming Throughput. The production quantity coming out of the system will directly relate to Throughput. The lead time from input to output will show how long Investment was committed within the system. All of these metrics meet the criteria of being both simple and relevant. They allow the flow of value through the system to be tracked and the value of working code delivered to be recorded.

Inventory-based metrics can also be self-generating based on the stage of transformation of the idea into working code. Records of the transformation as electronic documents can be sourced from the version or document control system.

Inventory-based metrics are predictive. Inventory at the input to the system can be used, based on historical production rate data, to predict the flow through the system.

Tracking Inventory and its flow through the software production system provides metrics that meet all the ideal characteristics for a control system.

Traditional Software Production Metrics

Traditional methods of controlling software production systems have focused on the use of effort-based metrics. The old bell wether has been the line of code (LOC). Almost everyone in the software business will tell you that lines of code is a useless metric. The problem with LOC is that it is an effort-based metric. It is meaningless for measuring the delivery of software value, the output from the system. However, for want of a better metric, lines of code remained popular for a long time.

Another favorite metric is the level-of-effort estimate, generally an estimate in man-hours for the development of a certain piece of requested functionality. The effort expended in hours is then compared with the estimate and adjusted periodically.

Traditional software metrics relate to the Operating Expense (OE) metric from Chapter 2. Traditional software metrics are compatible and perhaps heavily influenced by traditional cost accounting methods. They are focused on cost. Focusing on managing OE is suboptimal in achieving the system goal of more profit now and in the future, with a healthy ROI. It is more important to focus on Throughput and Inventory. Hence, traditional metrics do not meet the criteria for relevance.

Nor do they meet the criteria for simplicity. Brooks [1995] and others have pointed out that the business of software production is nonlinear. As software production is a complex system with feedback loops, it exhibits nonlinear behavior, that is, the effort expended in the system to produce lines of working code is not proportional to the quantity or quality of the output from the system. Hence, tracking effort-based metrics requires the translation through an unknown nonlinear equation in order to communicate client-valued functionality.

Effort-based metrics are not always self-generating. Every developer who has filled in a time sheet can tell you that the time sheet did not self-generate.

Effort-based metrics are not very useful as predictive indicators because of the nonlinear nature of software development and inaccuracy in estimation. The estimate is unlikely to represent the final outcome. Hence, the actual results must be gathered historically. When a developer fills a time sheet, it is an historical record rather than a predictive estimate. Hence, time sheets and effort recoding are lagging indicators.

Traditional software development metrics of lines of code written or time expended do not meet any of the ideal characteristics for system control metrics.

Measuring Inventory in the Software Production System

The Inventory in the system of software production must be measured through measures of client-valued functionality. The client of the client-valued functionality is the business owner, or customer, who is paying for the software. The ideas captured as functional requirements or a marketing feature list represent the raw material for the software production system. Requirements represent the ideas being transformed into executable code. These are often referred to as the functional requirements.

Nonfunctional Requirements and Inventory

Are nonfunctional or architectural requirements, for example, performance characteristics, of interest, and should they be tracked?

Nonfunctional requirements should be defined with a minimum requirement. This minimum requirement represents the level below which the functional requirements are not viable in the market. In other words, the functional requirements have no Throughput value unless a minimum level of nonfunctional specification is met. Hence, the base level of nonfunctional requirements does not require individual tracking. If the base level of performance cannot be met, the functional requirement would not be considered as delivered or complete. As functional requirements are being tracked, it is inferred that base level nonfunctional requirements are being tracked along with them.

It is therefore vital to skillfully manage the development of nonfunctional requirements. Excess work on nonfunctional requirements will increase OE, increase lead time, and decrease T.

However, there are preferred levels of nonfunctional performance that could be considered market differentiating, and these have a Throughput value. As they have a Throughput value and extra effort must be undertaken to achieve the additional performance, they should be tracked as Inventory through the system. The idea for the market differentiating performance is the input to the system, and working code that delivers the performance is the output. Architectural features for these ideas should be created and tracked as Inventory in the system.

Software Production Is Development Rather Than Research

There may be product or service concepts that are exceptionally new and are dominated by nonfunctional requirements, for example, on-demand, interactive video streaming broadcast over the Internet. Such projects are probably more rightly classified as research activity and should not be tracked with a management system such as the one described. This system of management is intended for development projects that are mostly infomatic in nature and can be thought of as software production rather than algorithmic research.

Tasks Are Not Inventory

Tasks represent effort that must be expended. They are essentially an internal organization of activity in the software production system. Tasks are not a system input and hence do not represent Inventory. Tasks are actions performed inside the system to move input through the system and generate output. A task list might be an essential project management tool, but it is of no interest to the client. It does not help the customer for the output of the system assess how much value has been delivered. The client does not value a task such as, “Assess the use of the Vitria™ message bus for loosely coupled asynchronous messaging of distributed components.” There is no direct correlation between development system tasks and delivery of value. Tasks are interesting only to the internal system. They do not represent the Inventory in the system or the output from the system.

Expressions of Inventory

Inventory can be expressed in different ways depending on the software engineering lifecycle method being used. This will be examined in depth in Section 2. In general, a unit of inventory is an idea for a client-valued function described in a format suitable for use by software developers. In UDP, a unit of inventory is a Use Case. In Extreme Programming, it is a Story Point. In FDD, it is a Feature from the Feature List. In traditional SDLC structured methods, the Function Point is the unit of inventory.

Measuring Production Quantity

Chapter 2 established that Throughput is the most important metric. Production Quantity is the output from the system of production that directly relates to the financial metrics of Throughput. Hence, the production system must track Production Quantity—the output of client-valued functions as working code. In Figure 5–1 production Quantity (Q) is generated by the system of software production from units of Inventory (V) at a rate (R).

Because the system of software production is treated as a continuous process producing a stream of software, it is more interesting to monitor the derivative of Q, the Production Rate (R). R is the quantity of functional requirements, expressed in the same units as the Inventory (V) delivered out of the system in a given period of time.

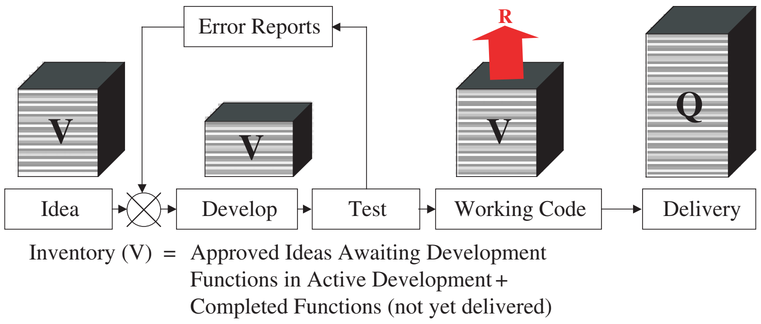
Error Reports
R
V
V
V
Q
Idea
Develop
Test
Working Code
Delivery
Inventory (V) = Approved Ideas Awaiting Development
Functions in Active Development +
Completed Functions (not yet delivered)

Figure 5–1
Inventory in the software production system.

Tracking Inventory with a Cumulative Flow Diagram

The inventory throughout the system can be conveniently tracked using a cumulative flow diagram[1]—a technique borrowed from Lean Production.

In Figure 5–2, the height between the deployment line and the requirements line displays the total inventory in the system. Different software development methods produce different patterns of cumulative flow. The pattern could be thought of as a method signature. Cumulative flow method signatures for each type of software development method will be discussed in Section 2. The pattern shown in Figure 5–2 represents a very Agile process with new material being fed into the system each week and little more than a week of inventory queuing at each step in the system.

Lead Time

There is a derivative measure of Inventory (V) which is important to calculations of OE—Lead Time (LT). Lead Time measures the length of time that Inventory stays in the system. Lead Time is how long it takes a unit of V to pass through the system from input to output. This is sometimes known as cycle time or queue time plus service time.

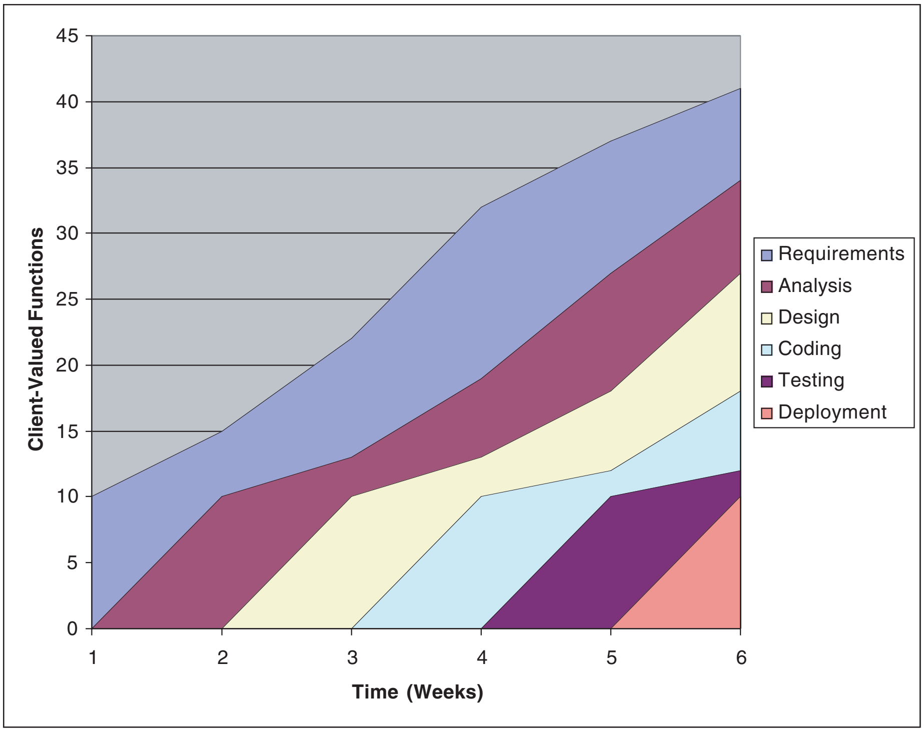

Figure 5–2
Cumulative flow of system inventory.

[1]Donald Reinertsen introduced the idea of using cumulative flow diagrams for design activities in *Managing the Design Factory*, p. 50.

Lead Time relates to the financial metrics from Chapter 2 because there is investment sunk in the inventory of ideas. The investment must be financed. The faster ideas can be turned into working code and delivered to a customer, the quicker inventory can be released. Releasing inventory faster has the effect of reducing the overall inventory in the system. Lead Time (LT) and Inventory (V) are related. This relationship is directly proportional as long as the production rate of the system remains constant. This is known as Little's Law.

LT has direct relevance to business. It determines how long it will be before the potential value added, stored in the inventory will be released as Throughput. The value of the Throughput may be affected by LT. The customer may be willing to pay more for shorter LT and hence, T is increased with shorter LT.

OE per Unit

The cost to transform a single unit of inventory through the system is the OE per unit or the Average Cost Per Function (ACPF). This must be determined by examining the global production quantity for a given period of the whole system and dividing it into the OE for operating the system for a period of time.

$$\text{ACPF} = \frac{\text{OE}_{\text{Quarter}}}{\text{Q}_{\text{Quarter}}}$$

For example, if OE per quarter is \$1,350,000 and production is 30 units per month, as shown in Figure 5–2, ACPF will be \$15,000.

$$\text{ACPF} = \$15{,}000 = \frac{\$1{,}350{,}000}{90}$$

Care must be taken to determine the cost to process inventory through individual steps in the system. If the system has been balanced using a Drum-Buffer-Rope subordination to the system constraint, for example, System Test in Figure 3–1, then there will be idle capacity in many steps in the system. Figure 3–1 showed that steps such as Coding had a capacity of 50 units per month. After subordination there will be 20 units of spare capacity. As all OE is a fixed cost, then production can be increased in nonbottleneck steps without incurring additional OE. Hence, the cost accounting concept of a cost per unit in a local process step is dangerous and misleading. The cost accounting approach shows reduced cost per unit when nonbottleneck capacity is used to increase local production. The local cost per unit is also called "efficiency," which was discussed in Chapter 2.

The only valid and useful cost metric for software production is ACPF; the purpose of which is purely estimation of future OE.

Summary

The software production system must measure the inventory level (V) at each step in the process, including the input. The total inventory held in the system is directly related to the Investment (I) metric from Chapter 1. The rate of delivery of production, the Production Rate, (R), for the whole system is directly related to financial metric Throughput (T). The cost (OE) of moving a single unit of inventory through the system is the Average Cost per Function (ACPF). This can be determined by the OE for a given period of time divided by the Production Quantity (Q), for the same period. Hence, Q, V, and ACPF are the equivalent production metrics to the financial metrics T, I, and OE.