CHAPTER 6:

EXPERIMENTATION AND RESULTS III

6.1 REVIEW YIELD

The four explicit review measures are

- the size of the program being reviewed,
- the review time in minutes,
- the number of defects found, and
- the number of defects in the program that were later found.

As we build our planning skills, we need to think about plan quality. What constitutes a good plan? How do our plans measure up and what can we do to improve our planning skills? The six key questions to ask about our plan are the following:

1. **Is it complete?**

Here is where our defined process can be most helpful. Do we have a form that specifies what is needed? Is it filled out? Have we left any blanks? While there are many ways to ensure our plans are complete, using well-designed forms is among the most convenient and economical.

2. **Is it accessible?**

To be accessible, a plan must be where we can find it, it must be in the proper format, and it must not be cluttered with extraneous material. While having complete plans is important, voluminous plans are unwieldy. We need to know what is in the plan and where it is. We should be able to
find quickly the original schedule and all the subsequent revisions. The defect data should be clear and the program size data must be available for every program version. To be most convenient, these data should be in a prescribed order and in a known, consistent, and no redundant format.

3. Is it clear?

Some entries are scrawled, others are scratched out. Incomprehensible notes are jotted in the margins. Some spaces are blank, or two or three items are inscribed in a space in which one belongs. Such work is not data and should be rejected. If the entries on the forms are not unmistakably clear, they cannot be used with confidence. If they cannot be used with confidence, there is no point in entering them at all.

4. Is it specific?

A specific plan says what will be done, when, by whom, and at what costs. If these items are not absolutely clear, the plan is not specific. For the initial phase, the plans we produce with the Project Plan Summaries specifically answer these questions. We will write the numbered program by the following week and for the cost of our total estimated time.

5. Is it precise?

Precision is a matter of relating the unit of measure to the total magnitude of the measurement. If, for example, we analyzed a project that took 14 programmer years, we would not be interested in units of minutes, hours, or probably even days. In fact, programmer weeks would probably be the finest level of detail we could usefully consider. For a planning job that takes five hours, units of days or weeks would be useless. On the other hand, units of seconds would be far too detailed.
Here we are probably interested in time measured in minutes. To determine an appropriate level of precision, consider the error introduced by a difference of one in the smallest unit of measure. A project that is planned to take 14 programmer years would require 168 programmer months. An uncertainty of one month would contribute an error of at most 0.6 percent. In light of normal planning errors, this is small enough to be quite acceptable. For a five hour project, an uncertainty of one minute would contribute a maximum error of about 0.33 percent. A unit of measure of one minute would thus introduce tolerable errors, while units of an hour or even a tenth of an hour would probably be too gross.

6. Is it accurate?

While the other five points are all important, accuracy is crucial. As we learn to plan our planning work, we should not be too concerned about the errors in our small task plans as long as they appear to be random. That is, we want to have about as many overestimates as underestimates. As we work on larger projects or participate on development teams, these small-scale errors will balance each other and the combined total will be more accurate.

Limitations:

As per the discussion and inputs from Mr Sachin Punadikar, Senior Staff Software Engineer, NAS Team, India Software Lab, IBM India. Ozone2, Pune. Once we have selected a precise size measure, we will need automated means to ensure it is accurately and economically counted. As we shall see, this information is essential if we are to develop an effective estimating and planning system. Manually counting the instructions in even small programs is tedious, time consuming, and inaccurate. For large programs, it is practically impossible. In the late 1980s, IBM had an inventory of over 30,000,000 LOC and shipped several million new LOC every year. It used a
Software Development Strategy with High Quality Design for Large Scale Projects

single standard automated counter to ensure that every laboratory produced consistent and accurate size measurement data.

In judging the quality of a program, we will typically want to reduce the defect counts to a standard form for cross-project or cross-product comparisons. When we divide the total defects found in a program by that program's LOC, we get the program's defect density. This normalization, at least to some degree, compensates for the size differences among our programs. The principal reasons we would do this are as follows:

- To help determine the quality of all or some part of our development process
- To determine the relative defect content of some parts or versions of large programs
- To judge the future maintenance and support workload for a program

In making such analyses, software engineers most commonly count the LOC added and modified during the development process. When considering the relative quality of several finished programs, however, we would be wisest to consider their total finished LOC. Mr. Sachin from IBM suggest that total program size correlated most closely with product service costs. While this may seem strange, it is more logical when we realize that defect-related costs are only a small part of total service costs for any but very poor-quality products. Here, the total number of service calls is likely to be more closely related to the total functional content of the program than to the new LOC content in a given release. Every counting choice has advantages and disadvantages. The use of total LOC in quality measures would suggest that small modifications to large programs are not a significant quality concern [28,30]. This has historically not been the case. There is evidence that small code changes, on a defects per changed LOC basis, are nearly 40
Software Development Strategy with High Quality Design for Large Scale Projects

times as error prone as new development. Conversely, counting only new and changed lines tends to ignore the quality of the large inventory of reused code. If this code had any significant defect content, our maintenance cost estimates would then likely be too low. In determining the relative quality of several releases of a single program, one is likely interested in both of these measures. The defects per KLOC in the new and changed code is probably the best indicator of the quality of the development and test phases, while the defects per KLOC of the total program is likely the best indicator of customer problems, maintenance costs, and service calls. Again, when in doubt or if substantial amounts of money are involved, get the data and do the analyses to see what works best for us.

In this thesis work, our principal focus is on the quality of our personal development process. Here, we will use the lines of new and changed code for all the analyses called. We will also, however, keep track of the lines of reused, added, deleted, and modified code.

In addition to the issues of packaging, development evaluation, and quality, there is a serious question about how to motivate engineers to reuse previously developed programs rather than developing them again. In programming, we can either write new code or reuse code we obtain from elsewhere. Actually, we can also reuse someone else's design and do the coding our self. The possibilities in fact are almost endless. We may be called on to convert existing code into a new language or operating environment, to copy sections of code from another program, or to modify existing code. When we include many previously produced source lines in a new program, we need to decide how to count them. If we treat them in the same way as newly developed code, the productivity numbers we use in planning our development must reflect this practice. On the one hand, it is important to motivate software engineers to use existing reusable
Software Development Strategy with High Quality Design for Large Scale Projects

code. On the other hand, once we are assured of the reusable programs' function and quality, the effort required to incorporate it is substantially less, per LOC, than writing a new program. If we include two or more types of code that require significantly different amounts of development time, we risk destroying the correlation between size and development resources. Regardless of how logical such choices might seem, they destroy the effectiveness of size measures as a planning tool. The debates about how to treat reused code can become heated. Because different counting approaches are appropriate for different purposes, there is no single best answer. Some argue that counting only new and changed LOC could motivate engineers to write excessively long programs just to get better productivity numbers. This is a possible danger, particularly where the programmers' personal productivity numbers are closely monitored by management.

A more realistic concern is that counting only new and changed LOC does not properly motivate engineers to reuse existing code. While this is certainly true, the problems most organizations have with increasing the volume of reuse are caused by more than just lack of engineer motivation. Reusing code that someone else has produced does take a good deal of work. We must find code that fits our needs, learn how to use it, assure our self of its quality, and verify that it complies with all our standards and conventions. Until organizations recognize that reusable programs are products that must be managed, marketed, and supported and until there are a sufficient number of suitably supported reusable software components, reuse will not be widely practiced. Software engineers have learned that attempting to achieve significant volumes of reuse of anything but the simplest standard routines is not a productive way to spend their time.

As per the inputs based on the experience of Mr. Pramod Ghorpade, Dev Manager at NetApp, Bengaluru; Mr. Vinod Ghorpade, Senior Practice Manager at Wipro Technologies, Reading,
United Kingdom (UK); Ms. Vasudha C, Module Leader at Wipro Technologies, Bengaluru reuse is the only currently available technology that shows promise of order of magnitude improvements in software development quality and productivity. One of the principal reasons this work spends so much time addressing high quality issues is to help us to produce code that is of suitable quality for inclusion in a reuse library. It may initially be our own private reuse library, but at least that would be a useful start.

**Limitation:** When a large scale job is estimated as a single unit, there are two sources of error. First, estimating accuracy will fluctuate around some mean or average. Second, the estimate will typically have some bias. If we were to gather data on many estimates and calculate their biases and if the estimating process were reasonably stable, we should be able to make an accurate bias adjustment.
Table 6 A Yield Calculation Experimentation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Defects Found</th>
<th>Defects Injected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Design Review</td>
</tr>
<tr>
<td>Planning</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Detailed Design</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Design Review</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Code Review</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Compile</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Post Development</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

Yield

<table>
<thead>
<tr>
<th>Phase</th>
<th>Yield Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Review</td>
<td>6/6 = 100%</td>
</tr>
<tr>
<td>Code Review</td>
<td>16/16 = 100%</td>
</tr>
</tbody>
</table>

Yield = (100 * Defects removed before compile) / Defects injected before compile

Courtesy: W Hamphrey
Phase_yield: Design_review_yield

\[
\text{Phase\_yield} = 100 \times \frac{\text{Removed}\_\text{in}\_\text{phase}}{\text{Removed}\_\text{in}\_\text{phase} + \text{Net}\_\text{escapes}\_\text{for}\_\text{phase}}
\]

\[
\text{Design\_review\_yield} = 100 \times \frac{3}{3+3}
\]

\[
= 50\%
\]
Phase_yield: Code_review_yield

\[ \text{Phase\_yield} = 100 \times \frac{\text{Removed\_in\_phase}}{\text{\textit{Removed\_in\_phase}} + \text{\textit{Net\_escapes\_for\_phase}}} \]

\[ \text{Code\_review\_yield} \]
\[ = 100 \times \frac{8}{8 + 9} \]
\[ = 47.1\% \]
Phase_yield: Complie_Yield

\[
\text{Phase\_yield} = 100 \times \frac{\text{Removed\_in\_phase}}{\text{Removed\_in\_phase} + \text{Net\_escapes\_for\_phase}}
\]

\[
\text{Complie\_Yield} = 100 \times \frac{6}{6+3} = 66.7\%
\]
Process_Yield

\[
\text{Process\_Yield} = 100 \times \frac{\text{Removed before compile}}{\text{Removed before compile} + \text{escapes into compile and test}}
\]

\[
\text{Process\_Yield} = 100 \times \frac{(3+1+8)}{(3+1+8)+9)}
\]

\[
= 100 \times \frac{12}{21}
\]

\[
= 57.1\%
\]
### 6.2 RISK REDUCTION LEVERAGE

Table 7 Risk Exposure

<table>
<thead>
<tr>
<th></th>
<th>Requirement change</th>
<th>Lack of communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initial stage</td>
<td>Mid stage</td>
</tr>
<tr>
<td>Risk Likelihood</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Risk Impact</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>TOTAL</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>RE= RL X RI</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

NMIMS, MUMBAI

Page 79
REQUIREMENT CHANGE

** Cost Per Person/Hr=Rs.100

Risk Exposure (RE)= Risk Likelihood*Risk Impact

Risk Exposure: Before

Initial stage: RE=4*2=8

Mid stage : RE=3*5=15

Final stage : RE=2*8=16
** REQUIREMENT CHANGE **

** Cost Per Person/Hr=Rs.100

Risk Exposure (RE)= Risk Likelihood*Risk Impact

** Risk Exposure: After **

Initial stage: RE=2*2=4

Mid stage : RE=1*5=5

Final stage : RE=1*8=8
LACK OF COMMUNICATION

** Cost Per Person/Hr=Rs.100

Risk Exposure (RE)= Risk Likelihood*Risk Impact

Risk Exposure

Before=7*7=49

Risk Exposure: After

After=4*7=28
RISK REDUCTION LEVERAGE

\[ RRL = \frac{R.E \text{ (Before)} - R.E \text{ (After)}}{\text{COST OF RISK REDUCTION}} > 1 \]

REQUIREMENT CHANGE

\[ RRL = \frac{1600(0.75) - 800(0.25)}{800} = 1.25 \]

LACK OF COMMUNICATION

\[ RRL = \frac{4900(0.66) - 2800(0.33)}{2100} = 1.1 \]
6.3 AGILE APPROACH FOR LARGE SCALE DEVELOPMENT

Customer will be the part of the development team, the active involvement of the customer in the project development improves the change management.

There is a reason why the Agile methods are becoming mainstream. [5, 8, 15] They can work! Although every Agile practice is not necessarily appropriate for every organization, each practice has delivered real value to many organizations, and some Agile practices can be used by anyone! [8]. Agile methodology is a relatively new lightweight methodology and is looked upon by many companies as the way forward in software development. When compared to traditional methodologies, agile provides a practical approach to software development which takes the changing nature of the business environment into account. The main aspects of agile development are about increased speed of development, less time-to-market, quality, efficiency, and management of the impact of change [1, 15]. Project Manager does the resource allocation to assign the available resources in an economic and in most efficient way. It is part of resource management. In project management, resource allocation is the scheduling of activities and the resources required by those activities while taking into consideration both the resource availability and the project time. A risk is something that may happen and if it does, will have a positive or negative impact on the project. A few points here, "That may happen" implies a probability of less than 100%. If it has a probability of 100% - in other words it will happen - it is an issue. An issue is managed differently to a risk and we will handle issue management in a later white paper. A risk must also have a probability something above 0%. It must be a chance to happen or it is not a risk. The second thing to consider from the definition is "will have a positive or negative impact". Most people dive into the negative risks but what if something goes right?
Change Management and dealing with changes is a key project management function for any industry. There might be enormous variance in efforts, cost, schedule, quality of deliverables if changes are not handled effectively. [4] Many projects are delayed and many other are closed (terminated) because project managers were not able to manage changes. Change management and dealing of with scope or requirements change is an importance function of project managers. However clear the requirements are, whatever the tool used for documenting requirements (right from Rational tools to Word document) and whatever the category of execution model (Pure Waterfall to evolutionary prototype) requirements are bound to change. [3, 12] The only difference to change is to what extent, and what is the impact on project success or failure. Having said this successful execution of any project relies on how good and effective is our change management mechanism. This is the reason why Agile approach for Monitoring and Controlling is required along with the Process model for better Change Management. [5]