Software Measurement

We can't accurately measure software, yet we must have measures if we are to understand large-scale design. This lecture discusses: the practical aims of measurement; the measures appropriate to them; ways of identifying and prioritising measurement issues; how to put together a measurement plan; the limitations common to many measurement methods; and the use of measurement indicators and estimators.

0.1 Why Measure?

In traditional, structured lifecycles we want to:

- Assess and manage risk.
- Trade design decisions against others.
- Track progress.
- Justify objectives.

It is difficult to do any of these things in an objective way unless we have some picture of where we are in a project and how much progress we have made in design.

0.2 What is it Useful to Measure

Although software itself resists absolute measurement there are many aspects of software projects for which measurement (even rough or indirect measurement) may be useful:

Schedule : Is it on time?

Cost : Can we afford to finish?

Growth : Will it scale?

Quality : Is it well made?

Ability : How good are we at design?

Technology : Is the technology viable?

These interact. For example, design ability influences the cost of running to completion which interacts with the way the project schedule is put together which has an impact on software quality which contributes to the growth of the system. In the sections below we look at each of these aspects in more detail, listing important categories of measure and suggesting some appropriate units of measure for these categories.

0.2.1 Issue Categories (1): Schedule

Category	Measure
Milestone	Date of delivery
Work unit	Component status Requirement status Paths tested Problem report status Reviews completed Change request status

0.2.2 Issue Categories (2): Cost

Category	Measure
Personnel	Effort Staff experience Staff turnover
Financial performance	Earned value Cost
Environment availability	Availability dates Resource utilisation

0.2.3 Issue Categories (3): Growth

Category	Measure
Product size and stability	Lines of code Components Words of memory Database size
Functional size and stability	Requirements Function points Change request workload

0.2.4 Issue Categories (4): Quality

Category	Measure
Defects	Problem reports Defect density Failure interval
Rework	Rework size Rework effort

0.2.5 Issue Categories (5): Ability

Category	Measure
Process maturity	Capability maturity model level
Productivity	Product size/effort Functional size/effort

0.2.6 Issue Categories (6): Technology

Category	Measure
Performance	Cycle time
Resource utilisation	CPU utilisation I/O utilisation Memory utilisation Response time

0.3 Identifying and Prioritising Issues

The issues we described above are not equally important for all projects. It is necessary, therefore, to find the ones which matter and prioritise them. Identification is possible from various sources, including:

- Risk assessments.
- Project constraints (*e.g.* budgets).
- Leveraging technologies (*e.g.* COTS).
- Product acceptance criteria.
- External requirements.
- Past projects.

Prioritisation usually requires some succinct form of contrast between issues, based on previous projects. Sometimes it is possible to obtain (rough) probabilities of occurrence of identified issues; then modify these according to the impact each would have if it became a real problem and the exposure to which your particular project has to that sort of problem. The table below is an example presentation of this sort of prioritisation.

Issue	Probability of occurrence	Relative impact	Project exposure
Aggressive schedule	1.0	10	10
Unstable reqs	1.0	8	8
Staff experience	1.0	5	8
Reliability reqs	0.9	3	4
COTS performance	0.2	9	1

0.4 Making a Measurement Plan

It is likely that you will want to take measurements several times during the course of a large software project and in those circumstances a measurement plan will be needed. The following are some of the things measurement plans typically contain:

- Issues and measures.
- Data sources.
- Levels of measurement.
- Aggregation structure.
- Frequency of collection.
- Method of access.
- Communication and interfaces.
- Frequency of reporting.

0.5 Limitations of Measurement

Measurement is necessary but fallible and subject to many practical limitations. Some of these, concerning the measurement categories introduced in Section 0.2, are listed below.

- Milestones don't measure effort, only give critical paths.
- Difficult to compare relative importance of measures.
- Incremental design requires measuring of incomplete functions.
- Important measures may be spread across components.
- Cost of design is not an indicator of performance.
- Current resource utilisation may not be best.
- Reliable historical data is hard to find.
- Some software statistics are time consuming to collect.
- Some measures only apply after coding has been done.
- Size doesn't map directly to functionality, complexity or quality.
- Time lag between problems and their appearance in reports.

- Changes suggested by one performance indicator may effect others.
- Often no distinction between work and re-work.
- Overall capability maturity level may not predict performance on a specific project.
- Technical performance measures often are not as precise as they may seem.
- Technical resource utilisation may only be known after integration and testing.

Be sure also to consider how you will maintain the quality of your measurement data, for example:

- Are units of measure comparable (*e.g.* lines of code in Ada versus Java)? Normalisation?
- What are acceptable ranges for data values?
- Can we tolerate gaps in data supplied?
- When does change to values amount to re-planning.

0.6 Exercise

Choose any small-scale software project that interests you and write a measurement plan for it. Compare your measurement plan with those of your classmates (this is especially interesting if you independently wrote measurement plans for the same project). Which is most convincing and why?