Measurement

Software Architecture, Process, and Management

In the previous lecture we saw that estimating properties that the project will have in the future is a difficult endeavour. In this lecture we will consider the conceptually easier task of simply measuring what you already have, whether that is at the end of the project or mid-way through.

Estimation and Measurement

- Both strategies require some data:
  - Human estimation cannot improve without feedback on previous estimations
  - Algorithmic methods require data to at least calibrate (parameterise) models, if not infer the model in the first place
- We need to measure both the answer, that is the effort and/or cost, as well as something that may help us predict the answer ahead of time

Estimation and Measurement

- Measuring the effort that was expended to produce a given amount of software is in theory rather simple assuming you have a reasonable record
- We could use the simple measure of how many person months
- This may not be the effort that was **necessary** but it is an upper bound on the effort that was **sufficient**
- Assuming of course the software is indeed finished:
  - Meaning it meets the requirements of the users in both quality and scope
  - Which is another measurement problem

Measurement

- There is a further reason to measure
- That is, to see where we are within a project
  - Hopefully in order to update our estimates
- This is complicated even further than the above case because the project is not yet finished
  - For example, the code may yet get **smaller**
- If we want to make reasonable decisions about projects, we have to measure some sort of data on which to base those decisions

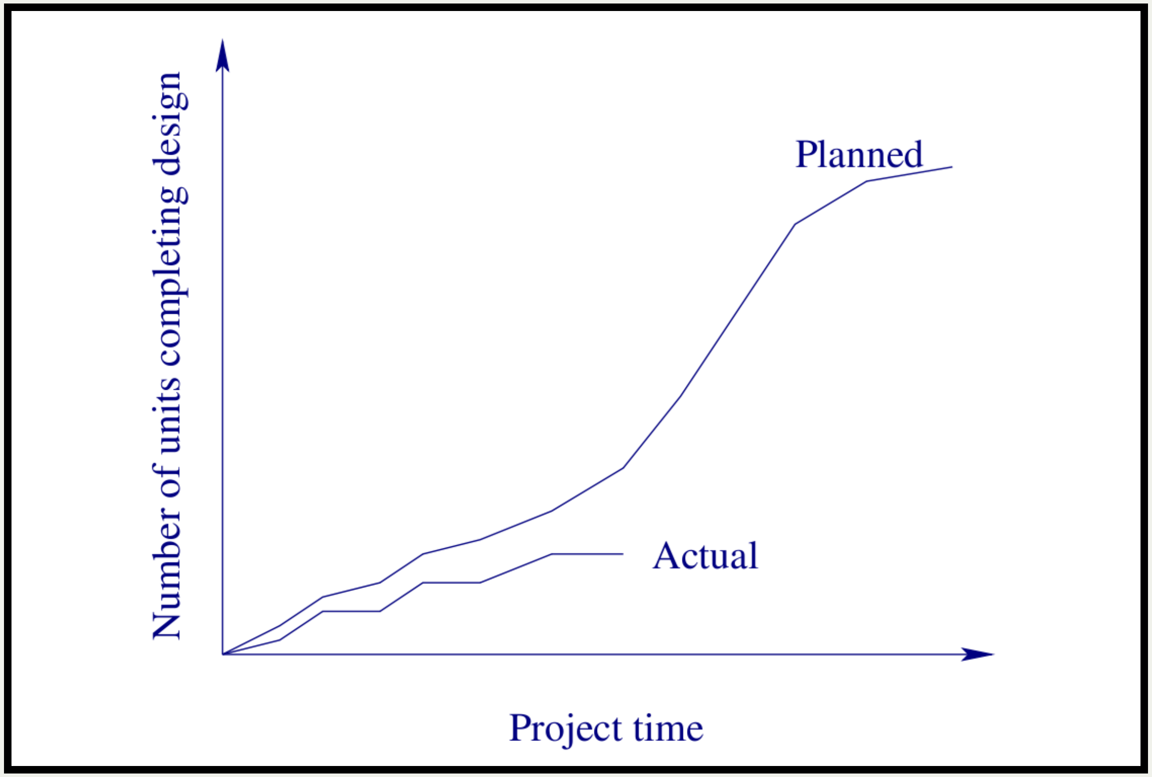
Identifying Issues

- How do you figure out what to measure?
- Some are obviously things that you must measure to have any idea what you are doing:
  - Project constraints e.g. you need to know if you are going over budget
  - External requirements/Product acceptance criteria (you need to demonstrate that requirements are met)
- Others are based on analysis of what risks you face in this project, what has gone wrong in previous projects, etc.

The What and the Why?

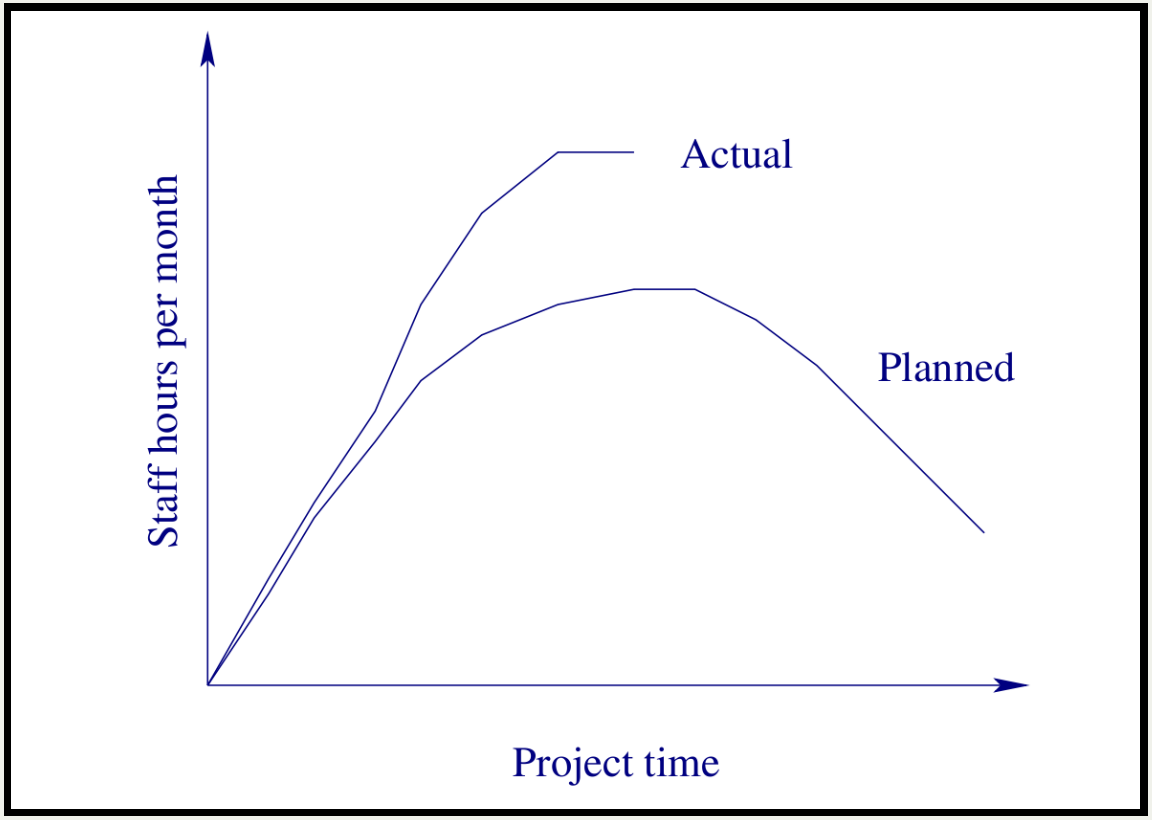
- It is important to distinguish **what** you are measuring from **why** you are measuring it
- In other words, it is important to distinguish what you are **measuring** from that which you wish to **find out**
  - You are measuring the number of source code lines because you wish to know how large your code base is
- Furthermore you need to understand what you will do about it when you have that answer

Indicator 1. Design Progress



- With an indicator and a plan, you can see if you are on track

Indicator 2. Effort



- With an indicator and a plan, you can see if you are on track

Issues That Can Be Measured

1. Schedule: Can we expect it to be done on time?
2. Cost: Can we afford to finish this project, or will it end up costing more than it is worth?
3. Size: How big is the product so far?
4. Quality: Is the product being made well, with few bugs?
5. Ability: How much design/coding/debugging/etc. can **this** team do per month?
6. Performance: Is the program fast enough, using reasonable resources?

Most of these interact strongly with the others.

Issues 1. Schedule

Wish To Know	Can Measure
Is progress being made?	Dates of milestone delivery
Is work being done?	• Components completed • Requirements met • Paths tested • Problem reports resolved • Reviews completed • Change requests completed

Issues 2. Cost

Wish To Know	Can Measure
How much is it demanding of our staff?	• Total effort • Number of staff involved • Staff experience levels • Staff Turnover
Are we getting our money's worth	• Earned value • Cost
Is the project making good use of external resources	• Available dates (too early/late?) • Resource utilisation

Issues 3. Size

Wish To Know	Can Measure
How large is this program so far?	• Lines of code • Number of components • Database size
How much does this program accomplish so far?	• Requirements met • Function points • Change requests completed

Issues 4. Quality

Wish To Know	Can Measure
How reliable is the software?	• Problem reports • Defect density • Failure interval
How hard is/was it to find and fix bugs?	• Rework size • Rework effort

Issues 5. Ability

Wish To Know	Can Measure
Is the development process well managed?	• Capability Maturity Model level 1. Initial(Chaotic) 2. Repeatable 3. Defined 4. Managed 5. Optimising
How productive is this team?	• Code size/effort • Functional size/effort

Issues 6. Performance

Wish To Know	Can Measure
Is the program fast enough?	Cycle time
Are the resources required by the program reasonable?	• CPU utilisation • I/O utilisation • Memory utilisation • Response time

Possible Measurements

Size/Complexity

- Number of lines of code
- Number of classes and interfaces
- Program size (binary)
- Weighted Micro Function Points
- Number of lines of customer requirements

Possible Measurements

Quality

- Defect Density(bugs per lines of code)
- Failure Interval
- Code coverage
- Cohesion
- Coupling
- Cyclomatic complexity
- Comment density
- Connascent software components

Possible Measurements

Other

- Program execution time
- Program load time
- Function point analysis
- Halstead Complexity
- Instruction path length
- Function Points and Automated Function Points, an Object Management Group standard

Measurement Example: Cohesion

- Cohesion is a measure of how strongly related are components of a particular module
- Ultimately, a module has high-cohesion if all its parts are working towards the same goal
- It is not a measure that lends itself well to automated calculation
- Many projects have a random “Utils” module which would typically have very low cohesion

Measurement Example: Coupling

- Coupling is a measurement of the dependency graph between modules within a software development project
- You can measure the coupling of a particular module which is simply the number of modules that it depends upon
- Or you can measure the graph as a whole, giving you some measure of how coupled the entire program is:
  - How you do this depends on the answer you wish for
  - Simply taking an average is risky

Coupling and Cohesion

- Coupling tends to be possible to calculate automatically:
  - Depends upon the kind of coupling
  - Depends on the language, a static type system can help
- Cohesion tends to be evaluated by humans
- But coupling and cohesion work well together:
  - One can artificially reduce coupling by increasing the size of modules
    - But this would reduce cohesion as well
  - One can artificially increase cohesion by reducing the size of modules
    - But this would increase coupling as well

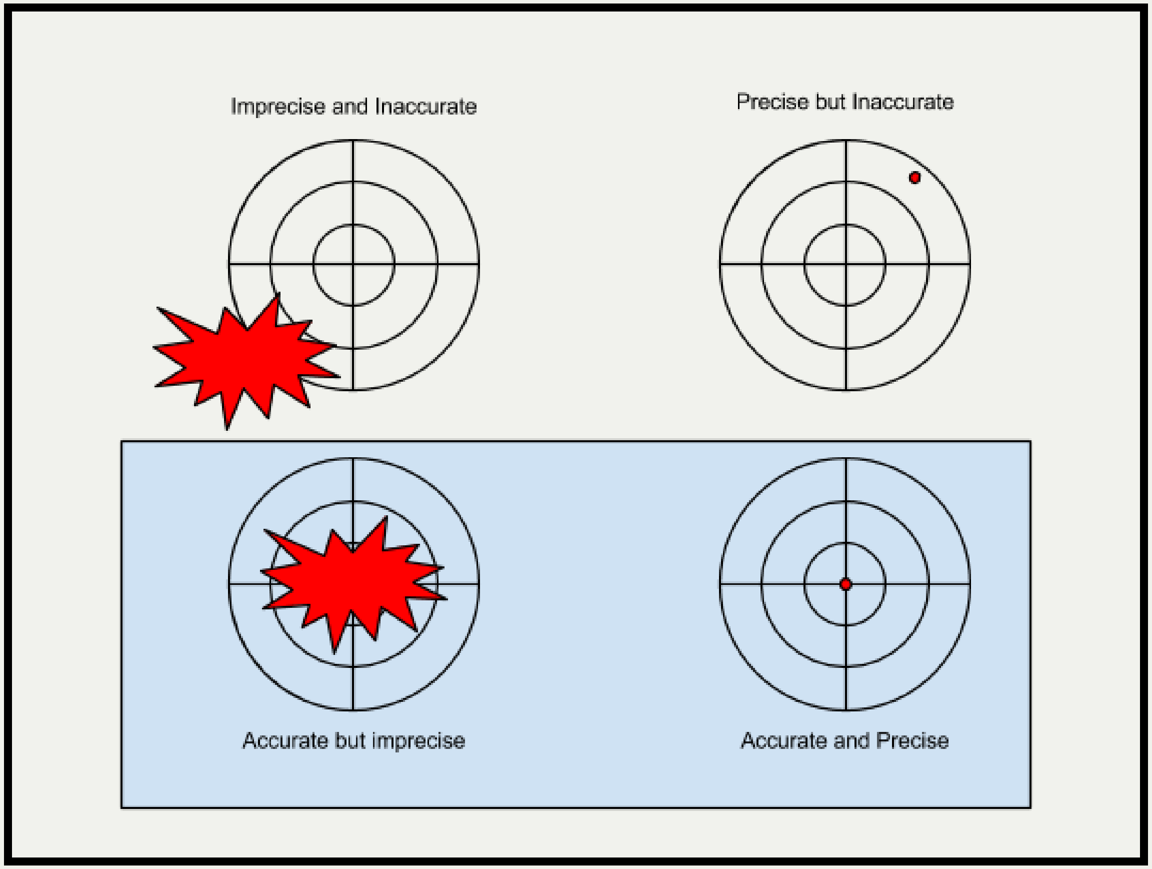
Problems with Measurements

Collating Measurements

- Difficult to compare relative importance of measures
  - Is high-cohesion more important than low-coupling?
- Important measures may be spread across components
- Hard to find reliable historical data to compare with
- Changes suggested by one performance indicator may affect others

Problems with Measurements

Precision and Accuracy: Better



Problems with Measurements

Precision and Accuracy

- Most measures are misleadingly precise, yet not very accurate
- Size does not map directly to functionality, complexity, or quality
- Milestones do not measure effort, only give critical paths
- Often no distinction between work and re-work

Derivatives

- Because of the problems with interpreting values it is likely less productive to look at the absolute values gained at snapshots
- It is generally more revealing to work out the trend in each measure
- E.g. Is your code-coupling increasing/decreasing or remaining the same over time?
- Is the rate of new code being added steady or increasing/decreasing?

Setting Targets/Rules

Measurement: Number of classes

- Note: The programmer is not necessarily trying to game the system
- There are times when a programmer must decide on the correct structure, for example trading-off simplicity/readability with adaptability/re-usability
- By giving incentives to produce more code/classes, you have given an additional reason to choose the latter over the former
  - If the programmer wants the reward it is very easy to convince themselves that the class-heavy approach is correct anyway
  - Among other psychological traits, the *confirmation bias* plays a role here

Estimation and Measurement

- Estimation and measurement share some of the same problems
- Psychological issues make human attempts at stating a value problematic
- As a result we would like an algorithmic push-button approach
- So far most attempts at this have been qualified or debatable successes at best
- Even those which might work can warp incentives and cause their own downfall
- However, we are compelled to try **something**
- A hybrid approach, of expert and algorithm is currently the best approach