SSW-540: Fundamentals of Software Engineering

Software Project Planning and Estimation

Roberta (Robbie) Cohen, Ph.D.
Industry Professor
School of Systems and Enterprises

A key software metric is size

- Size is one of the core metrics behind software estimation:
  - Schedule (duration)
  - Effort (cost)
  - Quality (defects)
  - Productivity
  - Size (scope)
- Without software size, it's hard to estimate:
  - How long a software project will take
  - How much it will cost
  - How many people we will need
  - How many defects we can expect to find during testing
  - How productive we are likely to be

There's a strong non-linear, positive relationship between size and schedule, effort (cost), defects, and reduced productivity.

Estimating code size is easy, right?

- No!
- During each phase of the life cycle, we can *progressively elaborate* our estimate of size.
- There are two distinct sizes to estimate:
  - Functional size – how much software functionality will be delivered to the end user – the size end users care about.
  - Technical size – how much software logic is needed to create the needed functionality – the size developers focus on.

The most common sizing methods

- Order of magnitude (T-shirt) sizing
- Functional size (normalized to *Function Points*, then to *Implementation Units*)
  - Functional and Business Requirements
  - User Stories
  - Use Cases
  - Function Points (ISO Standards: IFPUG MARK-II, COSMIC, FISMA, NESMA)
- Technical size (normalized to *Implementation Units*)
  - Objects and Business Process Configurations
  - Technical Components
  - Source Code Files
  - SLOC Counts

Note progression across development phases.

Estimating *effort* and cost

- The cost of developing software is primarily the cost of personnel.
- Other cost factors (tools, resources, travel, etc.) typically are minor compared to the cost of staff.
- The major determinant of development staff cost is software size.
- Fred Brooks, in his essays, The Mythical Man Month, notes that software developers are not like farm laborers.
  - Different skills and experience are needed in various amounts and at different times in a project.
  - A project requiring 100 staff-months cannot be completed with 1 staff member working 100 months or 100 staff members working 1 month!
- But the cost of 100 staff-months is the same regardless of the # of staff and the # of calendar months! When we estimate *effort*, we express it in staff-months.

Agile estimation

- Within Scrum
  - Tasks divided into time-boxed sprints
  - What is built when? How much can be built in each sprint? How much remains to be done?
- The planning game
  - Developed originally as part of Extreme Programming (XP)
  - Dependent on user stories as a measure of progress in the project
- Watch the 12 minute film: Agile Estimation

Estimation techniques

- Software organizations need to make software *effort* and cost estimates so they can plan around a budget and estimate time to completion.
- There are two types of technique that can be used to do this:
  - *Experience-based techniques:* The estimate of future *effort* requirements is based on experience of past projects and the application domain. Essentially, the manager or developer makes an informed judgment of what the *effort* requirement is likely to be.
  - *Algorithmic cost modeling:* In this approach, a formulaic approach is used to compute the project *effort* based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.
- Which of these is used in Agile estimation?

Experience-based approaches

- Experience-based techniques use judgments based on the *effort* expended on software development activities in past projects.
- Typically, you first identify the deliverables of software components or systems that are to be produced in a project.
- You estimate them individually and compute the total *effort* required.
- It usually helps to get a group of people involved in the *effort* estimation and to ask each member of the group to explain their estimate.
- But…
  - A new software project may have little in common with previous projects.
  - Software development changes very quickly and a project will often require the use of new and/or unfamiliar techniques.
  - If you have not worked with these techniques, your previous experience may not help you to estimate the *effort* required, making it difficult to produce accurate costs and schedule estimates.

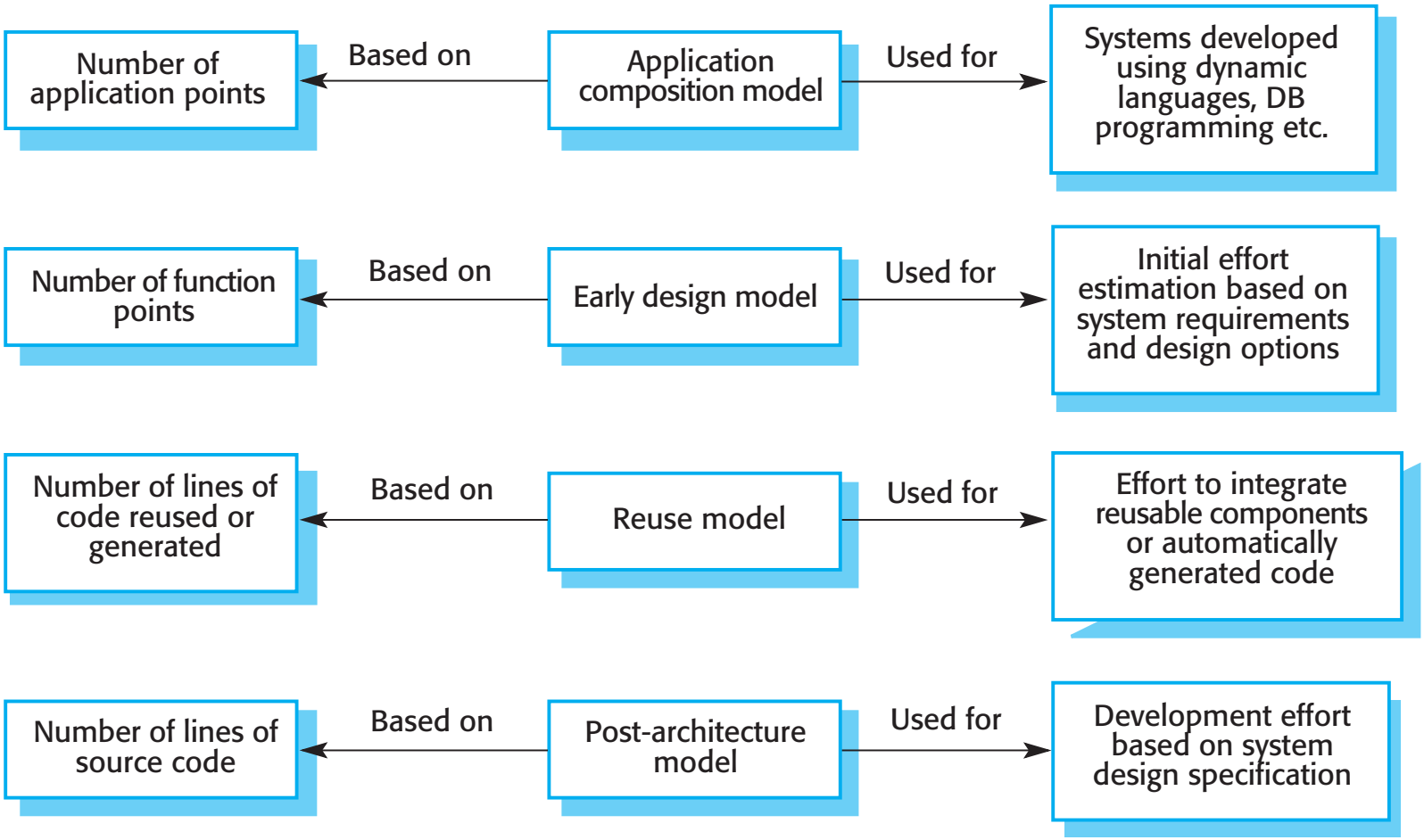
Algorithmic cost modeling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:
  - ***Effort* = A × Size^{B} × M**
  - A is an organization-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes.
- The most commonly used product attribute for cost estimation is code size.
- Most models are similar but they use different values for A, B and M.
- Algorithmic cost models are a systematic way to estimate the *effort* required to develop a system. However, these models are complex and difficult to use, and so mostly are used only by large companies.
- And even then, there are many attributes and considerable scope for uncertainty in estimating their values.

COCOMO cost modeling
(Constructive Cost Model)

- An algorithmic, empirical model based on project experience.
  - Well-documented, 'independent' and not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc. and has 4 sub-models:
  - Application composition model. Used when software is composed from existing parts.
  - Early design model. Used when requirements are available but design has not yet started.
  - Reuse model. Used to compute the *effort* of integrating reusable components.
  - Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

COCOMO estimating models



Detailed in the Sommerville textbook.

COCOMO II online tool

- http://softwarecost.org/tools/COCOMO/

Software Size Sizing Method: Source Lines of Code

	SLOC	% Design Modified	% Code Modified	% Integration Required	Assessment and Assimilation (0% - 8%)	Software Understanding (0% - 50%)	Unfamiliarity (0-1)
New	10000						
Reused	5000	0	0	20	3		
Modified							

Software Scale Drivers

Precedentedness	Nominal	Architecture / Risk Resolution	Nominal	Process Maturity	Nominal
Development Flexibility	Nominal	Team Cohesion	Nominal		

B factors

Software Cost Drivers

Product		Personnel		Platform	
Required Software Reliability	Nominal	Analyst Capability	Nominal	Time Constraint	Nominal
Data Base Size	Nominal	Programmer Capability	Nominal	Storage Constraint	Nominal
Product Complexity	Nominal	Personnel Continuity	Nominal	Platform Volatility	Nominal
Developed for Reusability	Nominal	Application Experience	Nominal	**Project**	
Documentation Match to Lifecycle Needs	Nominal	Platform Experience	Nominal	Use of Software Tools	Nominal
		Language and Toolset Experience	Nominal	Multisite Development	Nominal
				Required Development Schedule	Nominal

M attributes

Maintenance Off

Software Labor Rates

Cost per Person-Month (Dollars) 20000

Tool results

Results

Software Development (Elaboration and Construction)

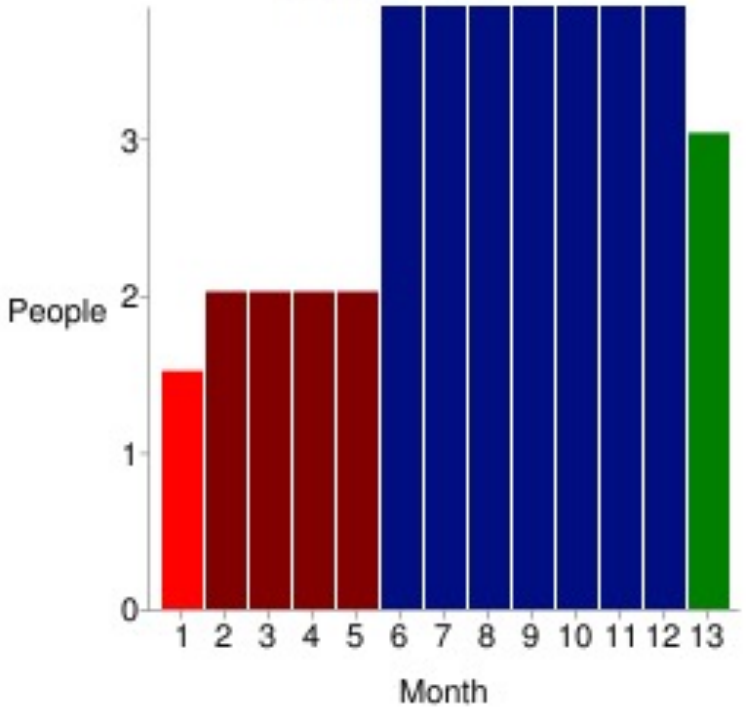
Effort = 38.8 Person-months
Schedule = 12.3 Months
Cost = $776424

Total Equivalent Size = 10450 SLOC

Acquisition Phase Distribution

Phase	Effort (Person-months)	Schedule (Months)	Average Staff	Cost (Dollars)
Inception	2.3	1.5	1.5	$46585
Elaboration	9.3	4.6	2.0	$186342
Construction	29.5	7.7	3.8	$590083
Transition	4.7	1.5	3.0	$93171

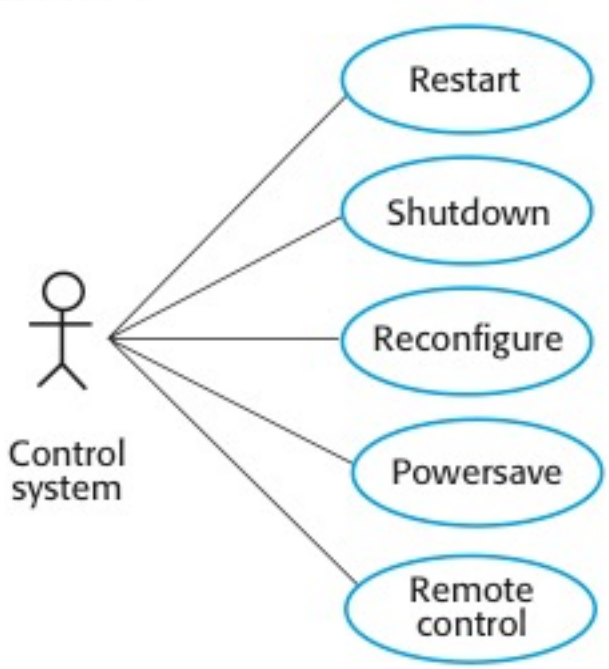
Staffing Profile



A more contemporary estimation technique is Use Case Points

- Use case points (UCP) are a method for estimating the *effort* required to build software
- Inputs are the detailed use cases in the software system where one takes:
  - A weighted count by type of unique actors across all use cases (UAW)
  - A weighted count of use cases, categorized by the number of steps required to carry out a successful case, and, if known, number of object classes needed (UUCW)
  - Factors that add technical complexity to the project (TCF)
  - Factors that add "environmental" complexity to the project (ECF)
  - A productivity factor (PF)
- **UCP = (UAW + UUCW) * TCF * ECF; *Effort* = UCP * PF**

Example project: Control System



	Actors			
Use Cases	Cntl Sys Admin	Cntl Sys Ops	Comm System	# of steps
Restart		√	√	3
Shutdown		√	√	3
Reconfigure	√		√	10
Powersave			√	2
Remote Control	√	√	√	10

Categorize and weight Use Cases

Category	Weight	Description	EX
Simple	5	At most 3 steps in the success scenario; *at most 5 classes in the implementation*	3 =15
Average	10	4 – 7 steps in the success scenario; *5-10 classes in the implementation*	
Complex	15	More than 7 steps in the success scenario; *more than 10 classes in the implementation*	2 =30
			Tot= 45

A count is made for each category of use case and that count is multiplied by the category weight to get an unadjusted use case weight (**UUCW)**.

Actor categories

Category	Weight	Description	EX
Simple	1	Actor represents another system with a defined API	1 = 1
Average	2	Actor represents another system interacting through a protocol	
Complex	3	Actor is a person interacting through an interface	2 = 6
			Tot 7

A count is made for each category of actor and that count is multiplied by the category weight to get a unadjusted actor weight (**UAW**). The unadjusted use case points are (**UUCP**) **= UUCW + UAW**.

Technical complexity factor (TCF)

- TCF captures some of the difficulties of implementing the non-functional requirements
- A number of factors are evaluated, for example:
  - Is concurrency required?
  - Will the software be part of a distributed system?
  - Are special security implementations needed?
  - Etc.
- The factors reflect characteristics of more modern, non-transactional systems as contrasted with function point analysis.
- TCF may reduce or enlarge the nominal effort measured by UUCP by about 40%

Technical Complexity Factors (TCF)

$TCF = 0.6 + (0.01 \times \sum W_i * F_i)$, where F is **0 to 5** and TCF ranges from 0.6 to 1.3.

Technical requirement	Weight
Distributed System	2
Performance	1
End User Efficiency	1
Complex Internal Processing	1
Reusability	1
Easy to Install	0.5
Easy to Use	0.5
Portability	2
Easy to Change	1
Concurrency	1
Special Security Features	1
Provides Direct Access for Third Parties	1
Special User Training Facilities Are Required	1

Example system's TCF

Technical requirement	Weight	Rating	Total
Distributed System	2	5	10
Performance	1	4	4
End User Efficiency	1	2	2
Complex Internal Processing	1	1	1
Reusability	1	1	1
Easy to Install	0.5	2	1
Easy to Use	0.5	2	1
Portability	2	1	2
Easy to Change	1	0	0
Concurrency	1	3	3
Special Security Features	1	5	5
Provides Direct Access for Third Parties	1	0	0
Special User Training Facilities Are Required	1	0	0

$TCF = 0.6 + (0.01 \times \sum W_i * F_i)$

$TCF = 0.6 + (0.01 \times 30) = 0.6 + 0.30 = 0.90$

Environmental Complexity Factors (ECF)

- ECF focuses on the influence that the development team will have on the effort
- The more experienced, more capable, more committed a team is, the faster the development is likely to go
- ECF also examines two other areas of development
  - How stable the requirements are
  - How difficult the programming language is
- ECF does not reflect the requirements (TCF does), but rather the implementers and the environment in which implementation will take place

Environmental complexity factors

Description	Weight
Familiarity with UML	1.5
Part-time workers	-1
Analyst capability	0.5
Application experience	0.5
Object-oriented experience	1
Motivation	1
Difficult programming language	-1
Stable requirements	2

$$ECF = 1.4 - (0.03 \times \sum W_i * F_i)$$

where F_i is **0 to 5** and ECF ranges from 0.725 to 1.4

Example system's ECF

Description	Weight	Rating	Total
Familiarity with UML	1.5	1	1.5
Part-time workers	-1	5	-5
Analyst capability	0.5	3	1.5
Application experience	0.5	0	0
Object-oriented experience	1	1	1
Motivation	1	5	5
Difficult programming language	-1	0	0
Stable requirements	2	5	10

$ECF = 1.4 - (0.03 \times \Sigma W_i * F_i)$

ECF $= 1.4 - (0.03 \times 14) = 1.4 - 0.42 = \mathbf{0.98}$

UCP metric

- Unadjusted use case points (unadjusted actor weight plus unadjusted use case weight) is
- adjusted for technical complexity factors (adjustment factor can range from 0.6 to 1.3) and
- adjusted for environmental complexity factors (adjustment factor can range from 0.42 to 1.55).
- This yields a UCP - Use Case Points.

Example: UCP = 52 * 0.9 * 0.98 = ~46

- UCP has little meaning unless we know how many use case points a development team can develop in a fixed amount of time. That's where PF, the productivity factor, comes in.

Productivity factor (PF) and UCP calculation

- A PF (Productivity Factor) is like a Velocity – it captures how productive a team is (or probably will be).
  - The Productivity Factor (PF) is a ratio of the number of staff hours per use case point based on past projects. It is expressed as "hours needed to complete a Use Case Point" so the higher the PF, the lower the productivity!!!
  - A team that needs 30 hours to complete a use case point is only half as fast as one needing 15!
- With no historical data,
  - Industry experts suggest a figure between 15 and 30 hours.
  - A typical PF value is 20 hours per use case point.
- Total Estimate of STAFF HOURS = UUCP x TCF x ECF x PF

Using a PF of 20 in our example, 46 x 20 = 920 staff hours = ~25 staff weeks = 6 to 7 staff months.

Summary:
Use Case Points (UCP)

- Use Case Points allow us to estimate development effort based on the use cases we have defined for our project.
- Different actors may require different interfaces so UNIQUE actors are an important contributor to total effort.
- Different goals will require different system solutions so unique Use Cases (each achieving a goal) form another important contributor.
- Technical complexity factors must be accommodated in our estimates as must "environmental" complexity factors.
- Productivity (defined as hours of effort needed per Use Case Point) will differ from team to team.
- For a detailed walk-through of an example, look at the Clemmons paper in this week's Canvas module.

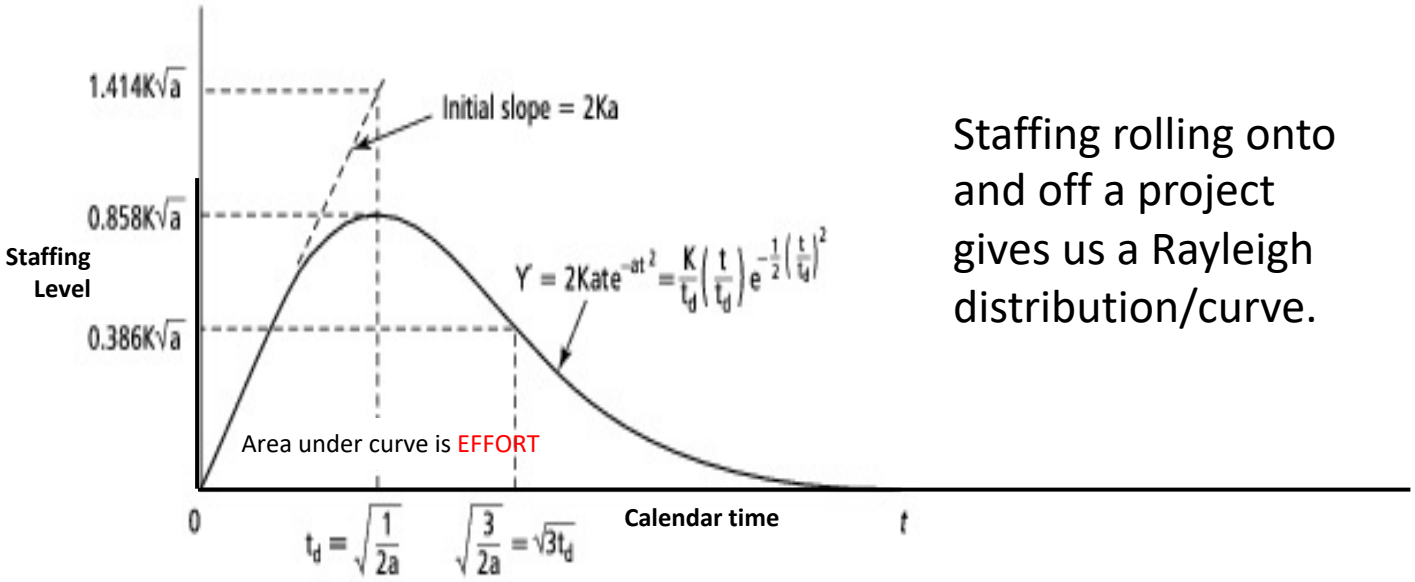
Brooks' Mythical Man Month – On Scheduling

- Usually projects fail for lack of calendar time.
  - Estimation is optimistic, last bug syndrome.
  - Costs vary by staff months, progress does not.
  - Sequential nature of some tasks.
  - Communication is key difference.
    - Farm workers vs. developers.
    - Each new worker must be trained by experienced staff.
    - Brooks' communication heuristic n(n-1)/2.
- Testing is the most poorly scheduled part.
  - Never enough time to adequately test.
  - Development runs late and test eats it.

Estimating staffing requirements

- Staff months (person-months) is an estimate of **budget**
  - Not calendar time
  - Not staffing levels
- The number of people working on a project varies depending on the phase of the project and the skills of the people.
- The more people who work on the project, the more total effort is usually required.
  - The more people on a team, the more people with whom each team member must communicate.
- A very rapid build-up of people often correlates with schedule slippage.
  - Brooks' Law: Adding people to a late project makes it later!

Staffing impacts calendar time



Staffing rolling onto and off a project gives us a Rayleigh distribution/curve.

- *Effort* (staff months) does not spread uniformly across time.
- At the project start, a small number of engineers is needed.
- As the project progresses, more work is required that needs more staff. At one point, the number of staff peaks (often system test).
- It then drops down (development finish, installation and delivery).

Estimating calendar time

- COCOMO 2 has a formula for estimating calendar months from *effort*.
  - First convert your *effort* estimate to staff months (use a 35-hour work-week or 150 staff-hours per month).
  - Then apply the approximate calendar time formula from COCOMO 2.

$$\mathbf{TDEV = 3 \times Effort^{1/3}}$$

- The actual COCOMO formula is $\mathbf{TDEV = 3 \times Effort^{(0.33+0.2*(B-1.01))}}$, where B is based on the COCOMO II scale factors and has the effect of representing the diseconomy of scale.
- Most critically, calendar time is **NOT** estimated as a function of staffing levels—only *effort*.

Some sample calculations

Staff-months	Calendar-months
5	5.1
10	6.5
15	7.4
20	8.1
30	9.3
40	10.3
50	11.1

- Note:
  - These estimates are computed independently of # of staff members, but clearly more staff will be needed/used when greater *effort* is required.

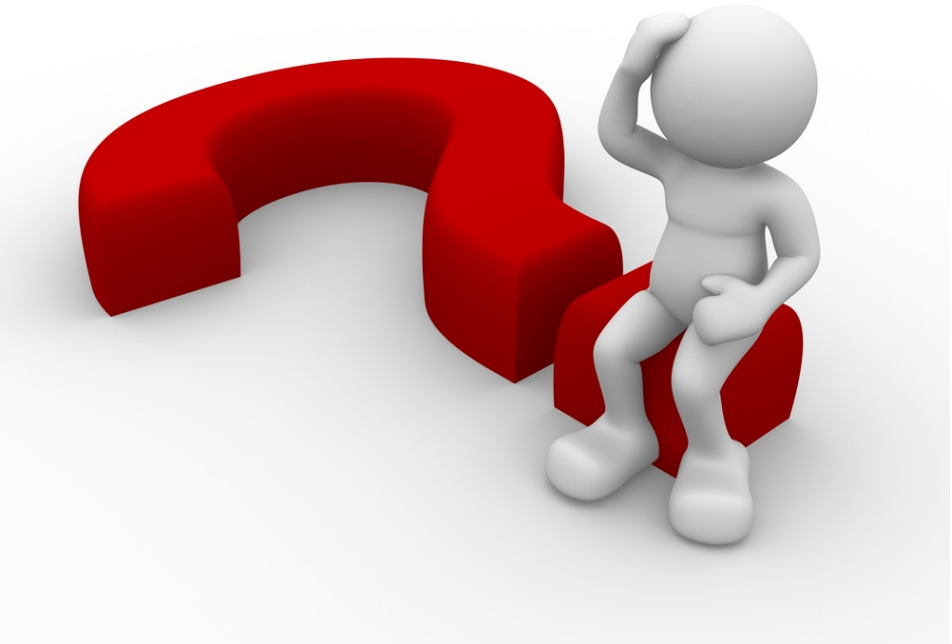
Caveats for *Effort* estimation

- The best estimates, even when algorithmic, are guided by experience.
- Observing and understanding organization-specific productivity is critically important for all estimation activity.
- UCP estimates are strongly influenced by the detail level of the use cases
  - the more steps, the higher the UCP
  - the more actors, the higher the UCP (but # of actors doesn't vary by detail-level)
  - (the variation of detail-level washes out with specific PF's!)
- Without experience-based tweaking, UCP estimates tend to be high compared to those of experts, unless organization specific PF's are applied.

So you've estimated your cost, what about pricing?

- Estimates of the total cost to the developer of producing a software system take into account, hardware, software, travel, training and, of course, *effort* costs.
- But there is no simple relationship between the development cost and the price charged to the customer.
- Broader organizational, economic, political and business considerations influence the price charged including, but not limited to
  - Contractual terms
  - Cost estimate uncertainty
  - Requirements volatility
  - Financial health of the developing company/organization
  - Market opportunity

Questions?



Your turn

This is an individual assignment.

Using Use Case Points, provide an estimate of the **effort** required to complete a project with 10 use cases and 15 actors. The actors and steps for each of the 10 use cases are shown in the assignment on Canvas. Also provided are the Technical Total Factor and Environmental Total Factor. These represent the sum of the weighted ratings given to each of the factors and must be transformed into complexity factors affecting the use case points.

Read the assignment carefully. The UCP Calculator found in this module can be of some help, but it assumes you are rating each of the complexity factors (which you are not) and it estimates UCP, not **Effort**. The Clemmons paper found in the module will walk you through the full process of computing **Effort**.

Python Pointers: Ins and Outs

Good practices you should be using in your scripts:

- Separate input from processing
  - Input is subject to user errors; processing should not be
  - Processing algorithms should be easily modiafied
  - Good practice: put processing in functions; put input in try/except clauses
- Handling likely exceptions within a script is not optional
  - Good programmers anticipate errors and provide handling for them
  - Good practice allows users to try again
  - Good practice gives users a way out as well