Mythical Man Month

Dr. Paul West

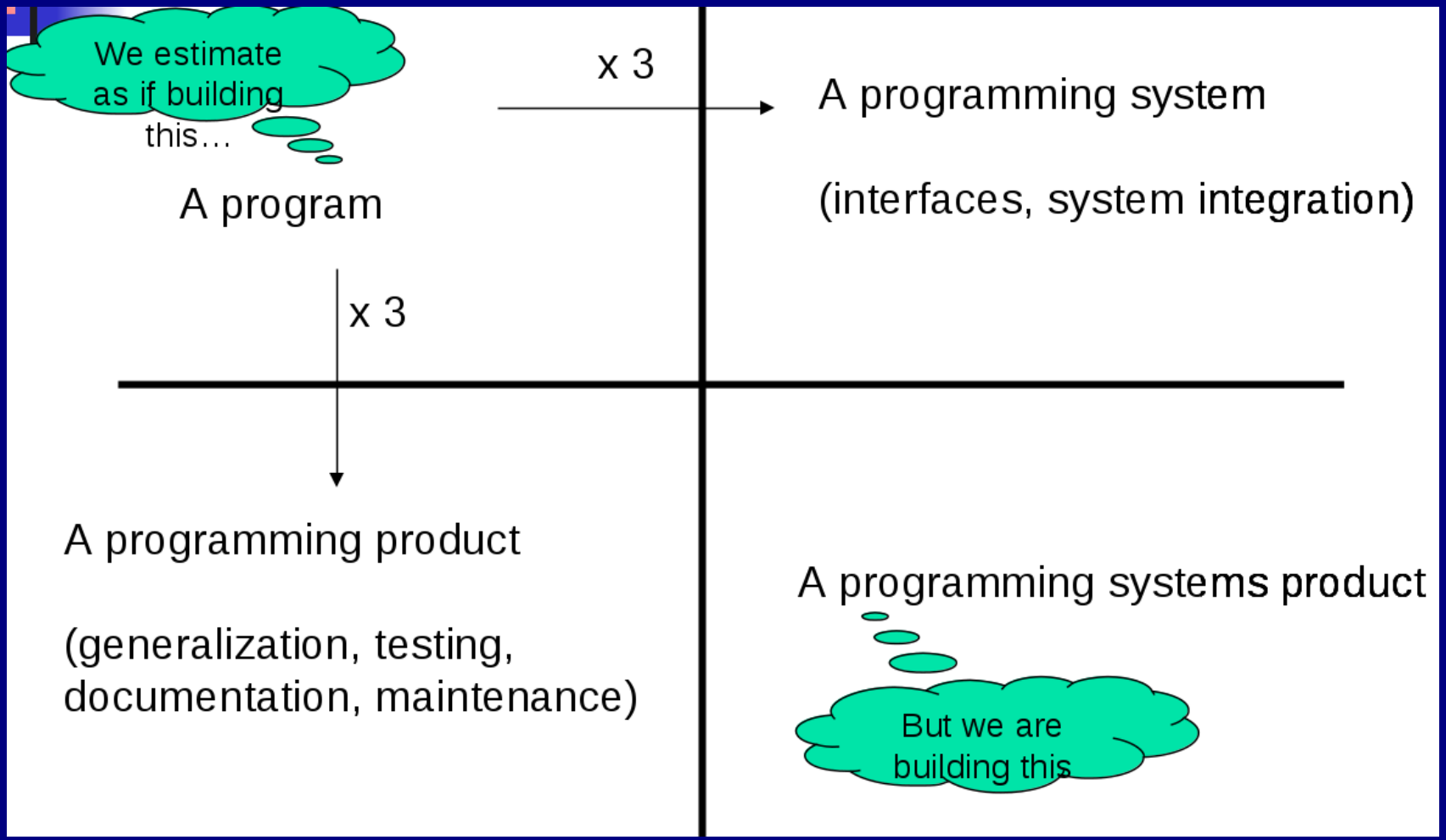
Department of Computer Science
College of Charleston

March 27, 2014

About the Book

- Author: Fred Brooks
- a book on software project management
- central theme : “Adding manpower to a late software project makes it later.”

- The Bible of Software Engineering: everybody reads it but nobody does anything about it!
- Working in IBM, managing the development of OS/360
- OS/360 was a great success, becoming the most important IBM mainframe operating system.
- mistakenly added more workers to a project falling behind schedule.
- mistakenly assert that one project, writing an Algol compiler, would require six months-regardless of the number of workers involved (It required longer).


We estimate as if building this…
A program
x 3
A programming system
(interfaces, system integration)
x 3
A programming product
(generalization, testing, documentation, maintenance)
A programming systems product
But we are building this

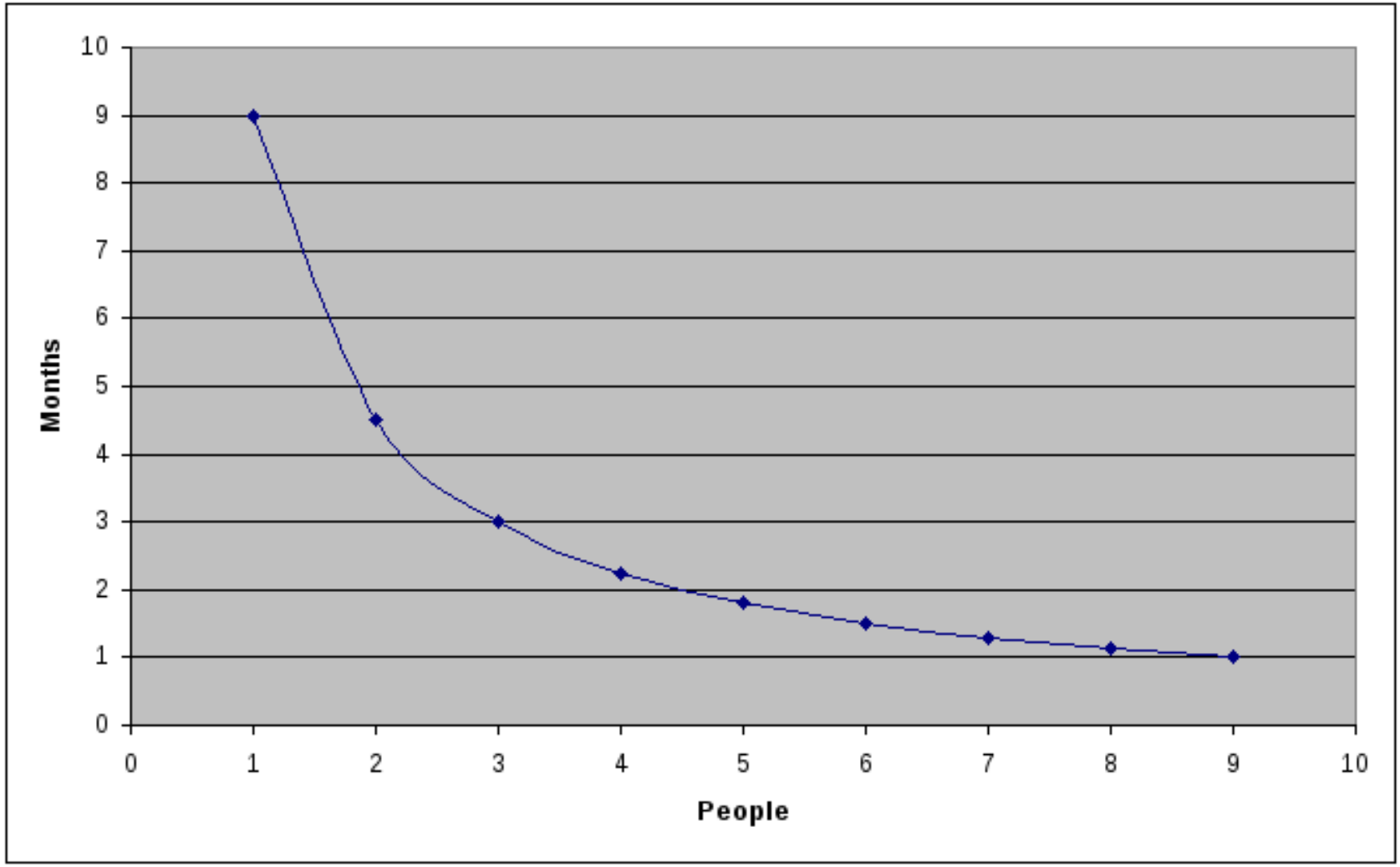
What makes programming fun

- Making things that others find useful.
- Making complex objects out of parts.
- Continuous learning because the task is always different.
- Using tools and “materials” that do not degrade.

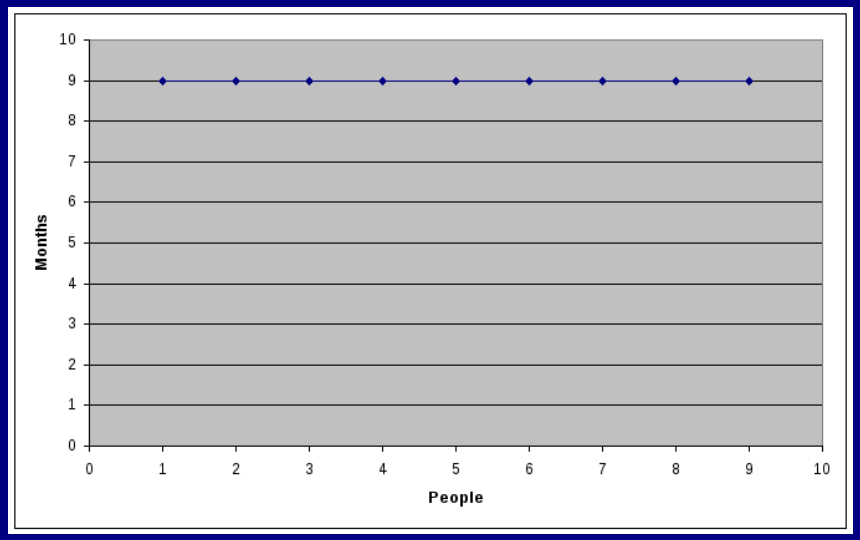
What causes problems?

- Computers demand perfection.
- A person does not control the “circumstances” of their work (goals, resources, information).
- Working out the bugs is just that - work.
- Working out the bugs takes an order of magnitude longer than one expects.
- The resulting software seems to be obsolete before it is released.
  - However, this is more of a perception

- According to Brooks, failure to meet schedule is the reason for most software project failures. Why?
  - We don't know how to estimate (overly optimistic).
  - We confuse effort with progress.
  - Software managers give in to pressure to reduce time estimates because they are uncertain of their estimate.
  - We don't monitor schedule progress properly.
  - We tend to handle schedule slips by adding people. However, this tends to magnify the problem.

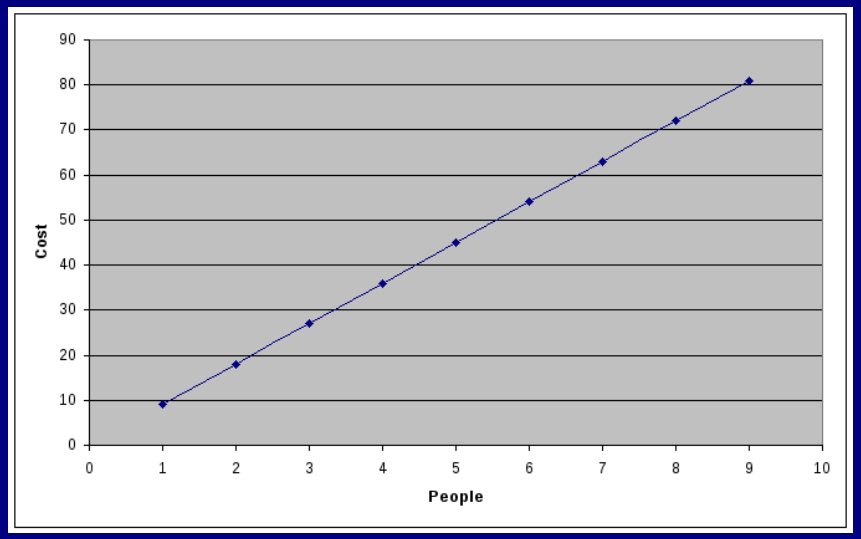


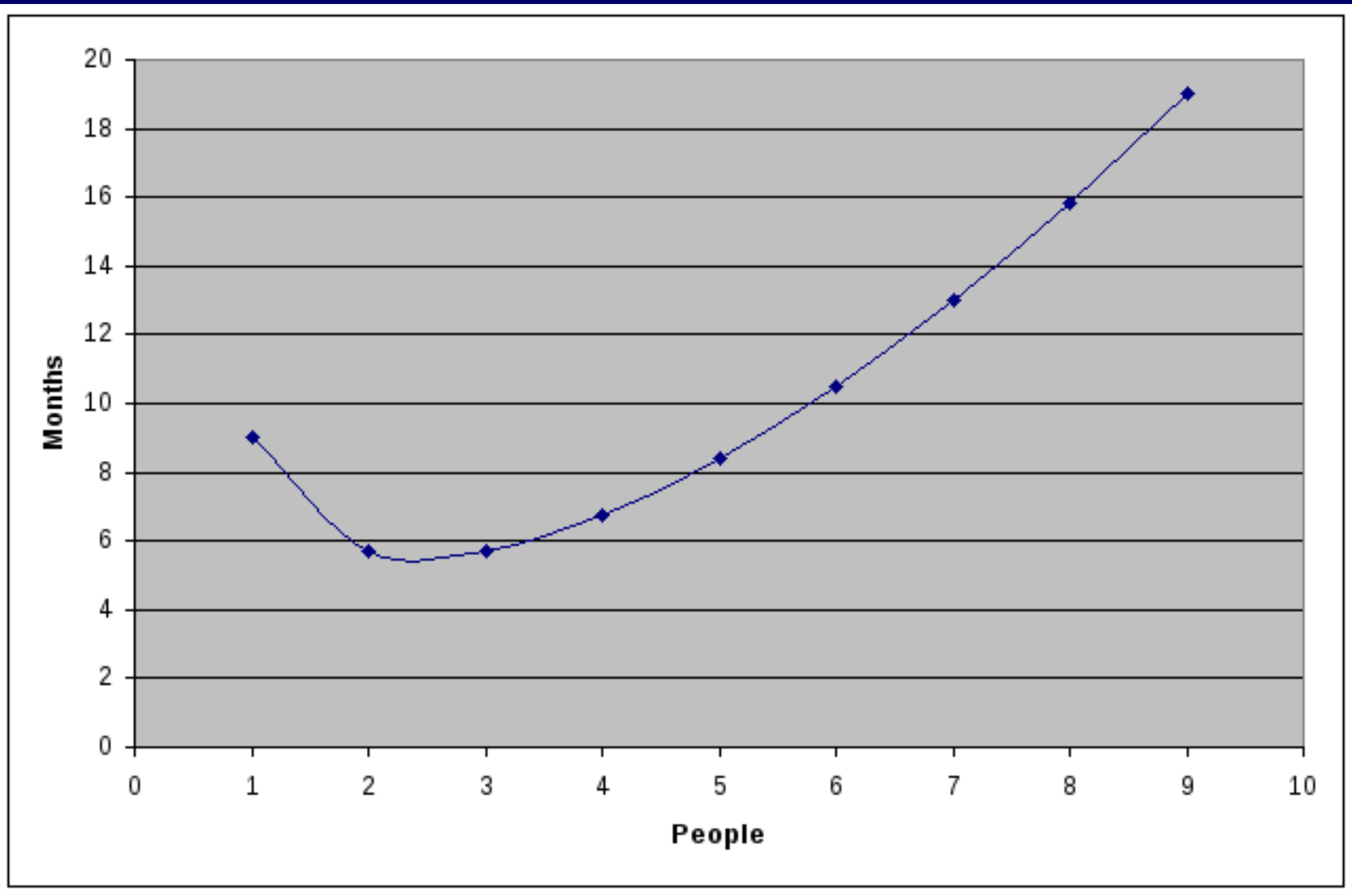
When there is no dependency among people, the amount of time to do a task diminishes with each new person. Note that the cost (people * months) is constant.



When there is a dependency and the task cannot be partitioned, adding people has no effect on time required

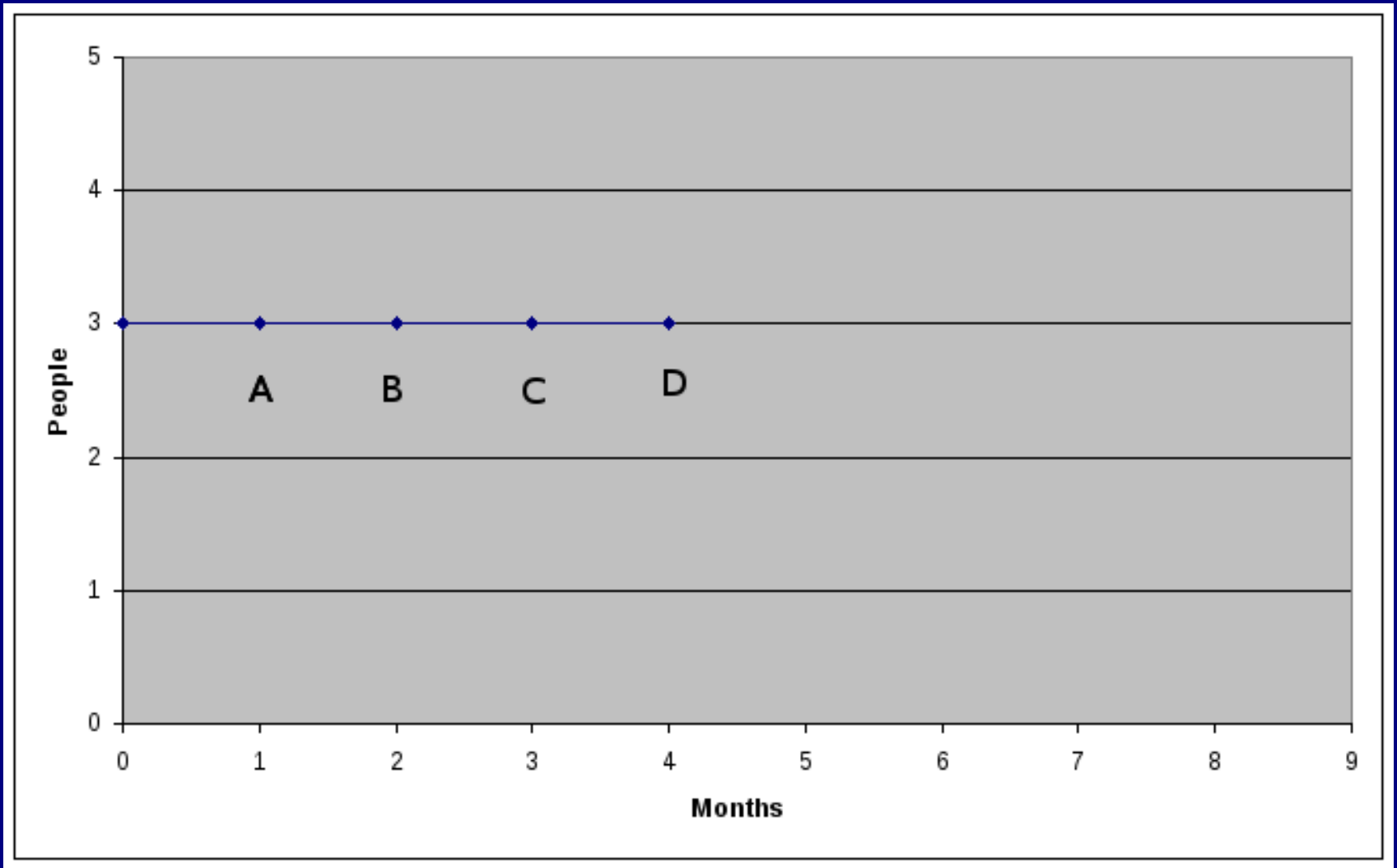
but it has a big effect on cost.



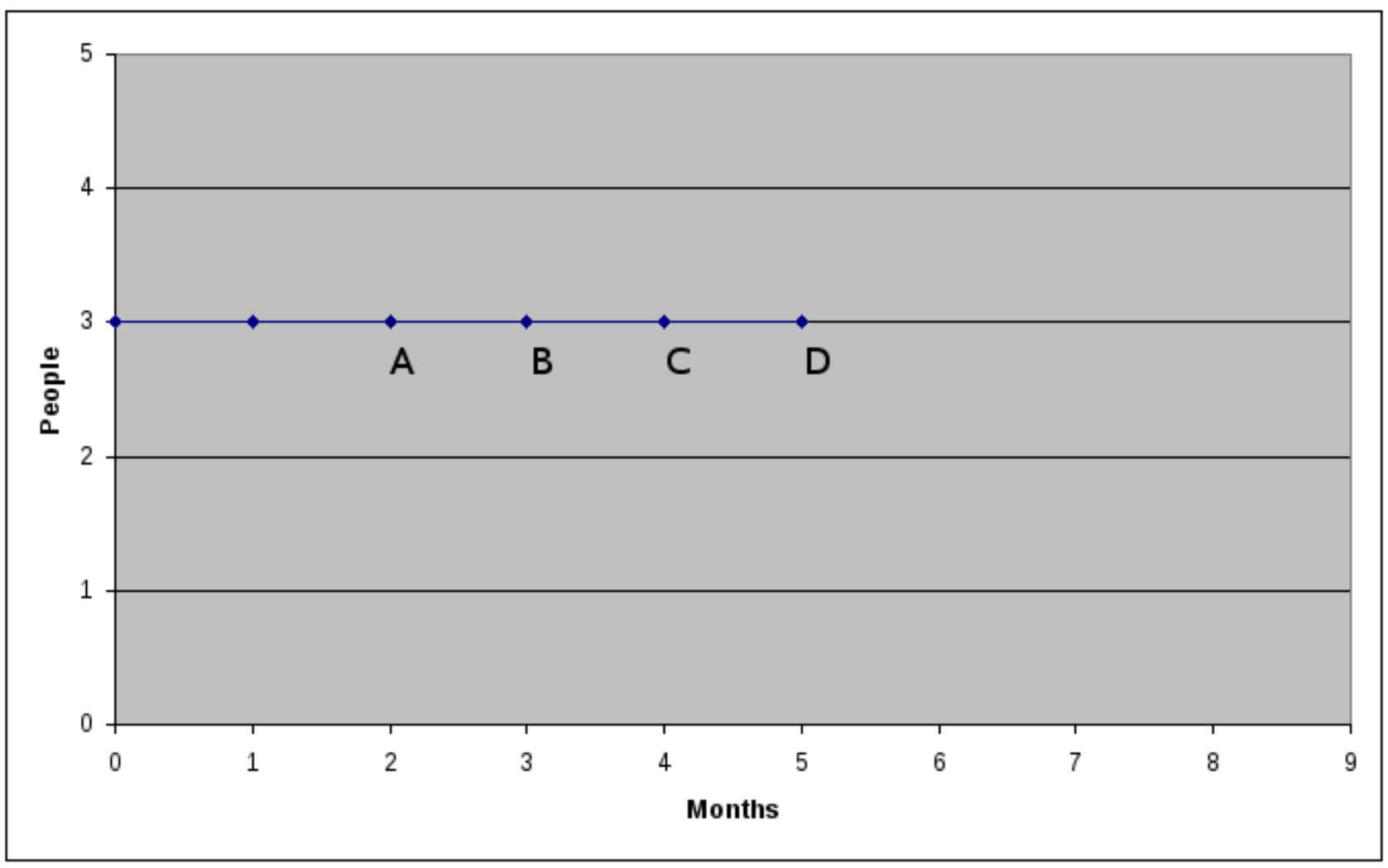


If task can be partitioned, but requires communication, must handle training and communication as each person is added. Can cause project to be later.

What about a late project?

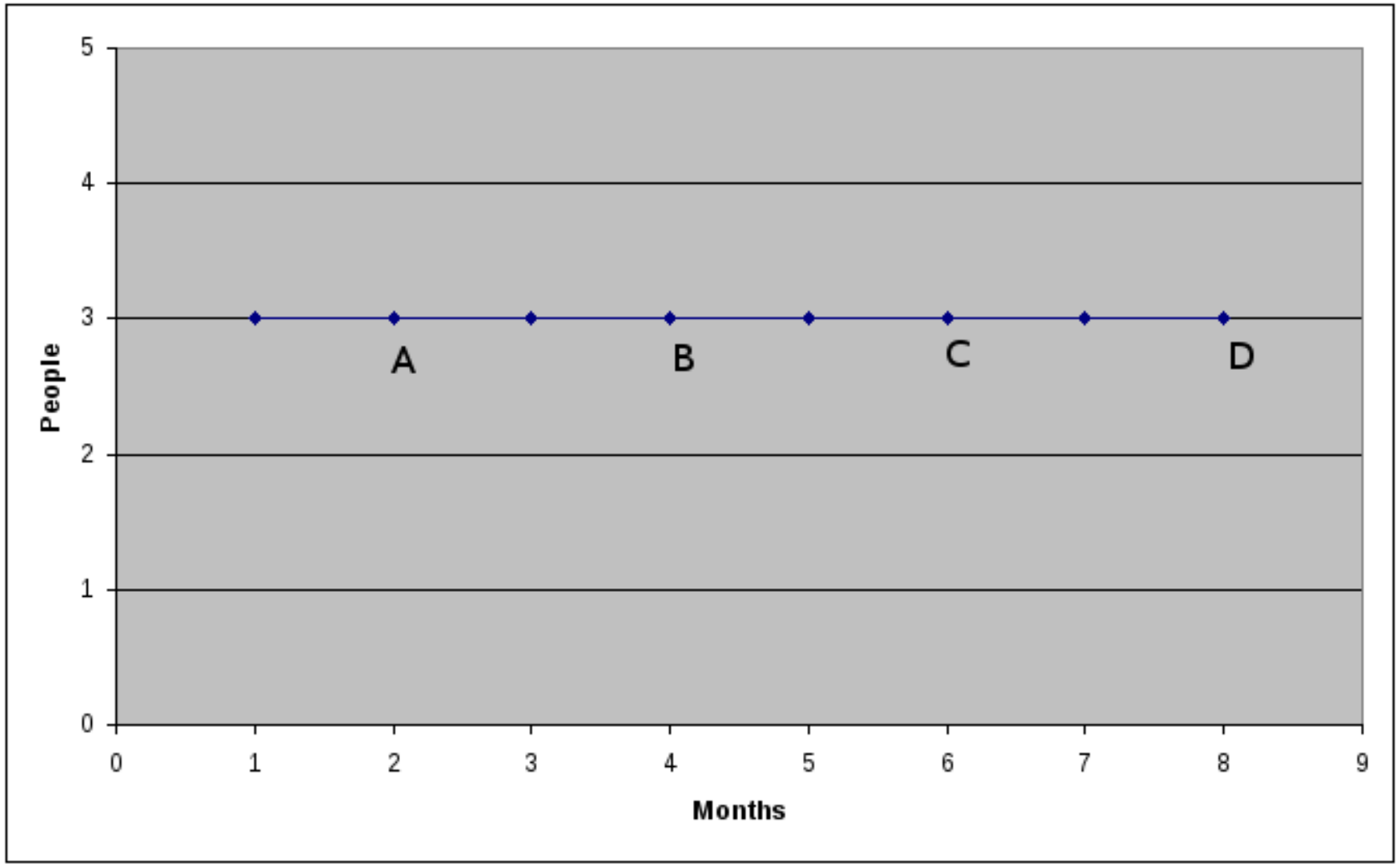


Suppose that at 2 months, we achieve milestone A. We must deliver on time. What can we do?



Assume the project will go according to plan from here on (optimism!)

So 9 person-months must be accomplished in 2 months - Add 2 people



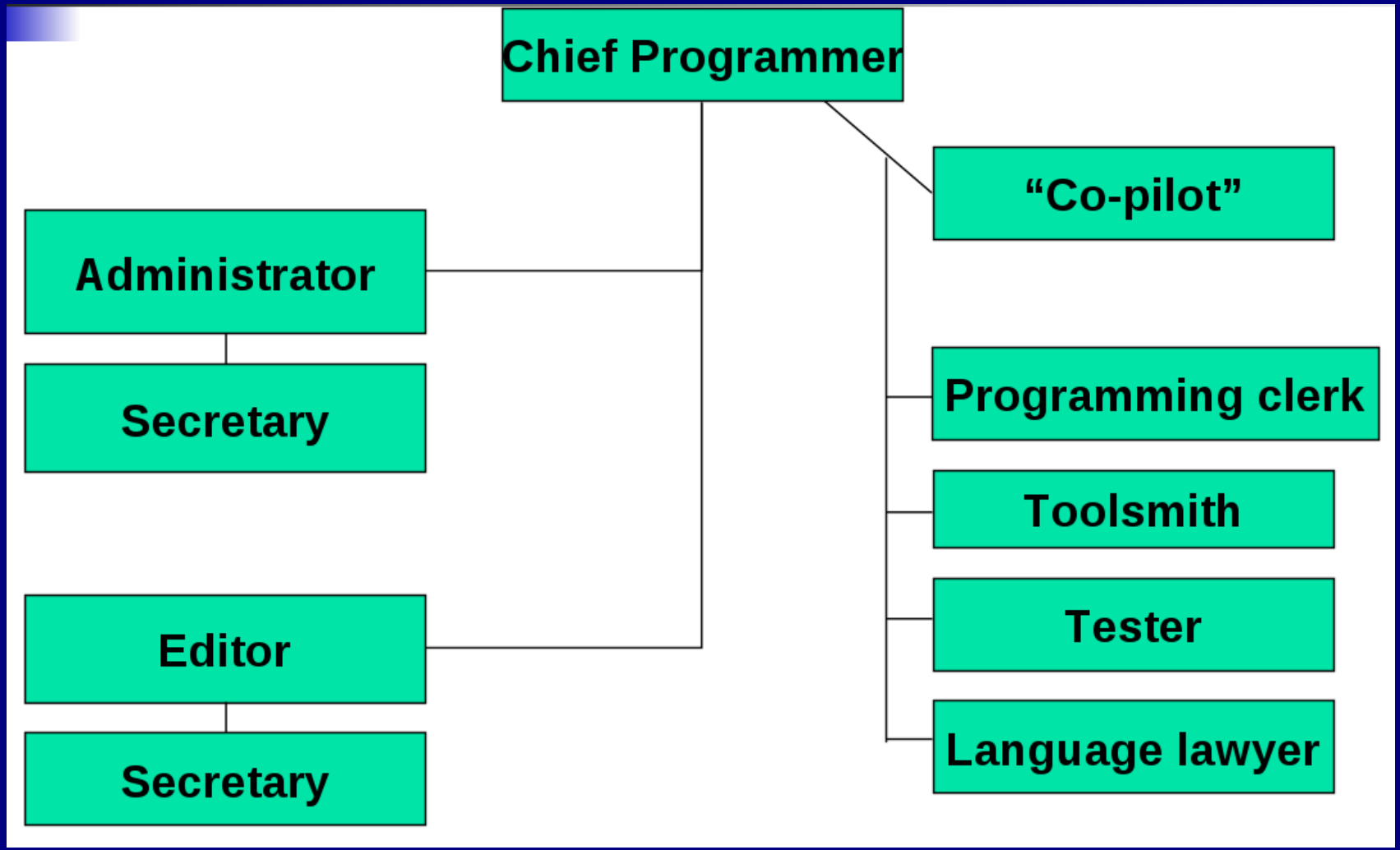
ssume the project estimate is off by a factor of 2.
So 18 person-months must be accomplished in 2 months - Add 6 people.

What about training?

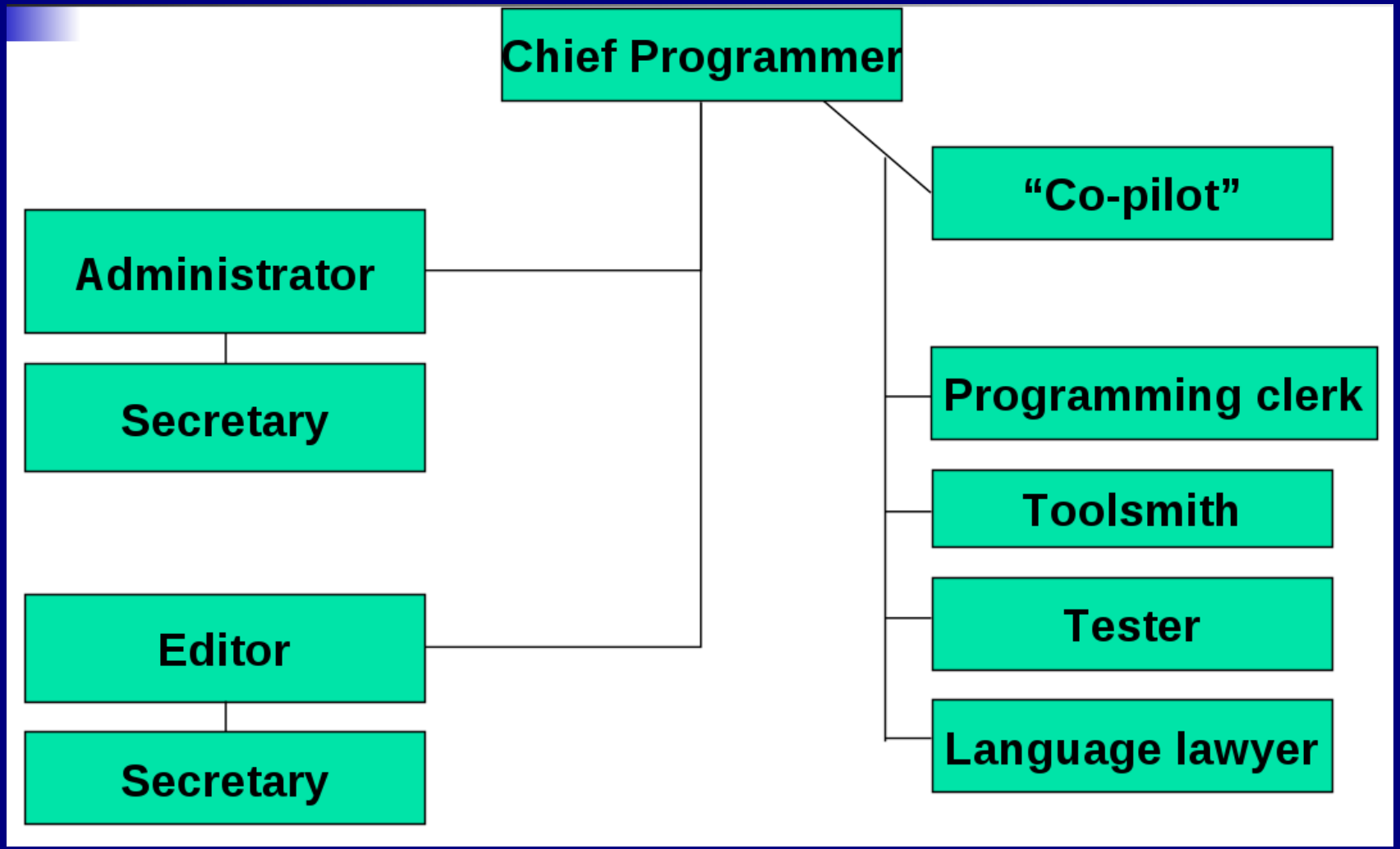
- Let's say we add 2 people at month 2.
  - Must train them - assume this takes 1 month and requires 1 of the other 3 people.
  - During month 3:
    - 3 person-months of training work
    - 2 person-months of actual work.
  - Still have $9 - 2 = 7$ person-months of work, but only 1 month left!
  - So what do we do? Add more people equals having a later delivery.
- Brooks's law: Adding people to a late software project makes it later.

- How should software teams be organized?
- Issues
  - Productivity varies widely among individuals.
  - Want as few, highly productive individuals as possible.
  - But, need to be able to scale to large software systems.

- surgery is led by one surgeon performing the most critical work himself while directing his team to assist with or overtake less critical parts
- it seems reasonable to have a “good” programmer develop critical system components while the rest of a team provides what is needed at the right time.
- Additionally, Brooks muses that “good” programmers are generally 5-10 times as productive as mediocre ones.



How might this concept be updated for 2014?



How might this concept be updated for 2014? This works for small projects. How might it be scaled for larger projects?

Aristocracy, Democracy and System Design

- Unity of design (“conceptual integrity”) is the most important property of a system.
  - Why? Because of ease of use.
- Achieving this requires an architecture, separate from implementation.
  - Architecture is a complete description of the software system from the user’s point of view.
  - Developed by architects, separate from implementers.
  - Architecture requires creative activity, but so does implementation.
  - There is implementation work to do even before the architecture is ready.

The Second-System Effect

- The second system an engineer designs is the most dangerous system he will ever design, since he will tend to incorporate all of the additions he originated but did not add (due to the inherent time constraints) other things.
- Thus, when embarking upon a second system an engineer should be mindful that he is susceptible to over-engineering it.

Self-Discipline

- First job by architect
  - Play it conservative
  - Make sure to do the job right
  - Scrupulously keep “added features” out
- Second job by architect
  - The most dangerous system
  - Avoid using an architect for the 2nd system
  - Why is this?

2nd System Difficulties

- No generalization from experience
- Tendency to over design
- Tendency to refine obsolete techniques

What can be Done?

- Architect
  - Capability x is worth not more than m bytes of memory and n microseconds per invocation
- Project Manager
  - Insist senior architect has >= 2 systems
  - Ask the right questions during design

How to make sure everyone hears architectural decisions?

- Written specifications
- Formal definitions
- Direct incorporation
- Conferences
- Multiple implementations
- Telephone log
- Product test

Why Did the Tow of Babel Fail?

- There were:
  - A clear mission
  - Enough resources (people and materials)
  - Enough time
  - Proper technology
- What was missing was communication!

Communication

Team 1:
I'm running behind on schedule. My component runs infrequently. I will change the design so the component takes more time.

Team 2:
My component relies on Team 1's. I'm glad they will meet their time allowance, because my component depends on that.

- This will be a disaster.
- Teams should communicate
  - Informally
  - Meetings
  - Workbook

Progress Tracking

- Question: How does a large software project get to be one year late?
- Answer:

Progress Tracking

- Question: How does a large software project get to be one year late?
- Answer: One day at a time! Incremental slippages on many fronts eventually accumulate to produce a large overall delay. Continued attention to meeting small individual milestones is required at each level of management.

Conceptual Integrity

- To make a user-friendly system, the system must have conceptual integrity, which can only be achieved by separating architecture from implementation.
- A single chief architect, acting on the user's behalf, decides what goes in the system and what stays out.
- A "super cool" idea by someone may not be included if it does not fit seamlessly with the overall system design.
- In fact, to ensure a user-friendly system, a system may deliberately provide fewer features than it is capable of.
- The point is that if a system is too complicated to use, then many of its features will go unused because no one has the time to learn how to use them.

The Pilot System

- When designing a new kind of system, a team *will* design a throw-away system (whether it intends to or not).
- This system acts as a pilot plant that reveals techniques that will subsequently cause a complete redesign of the system.
- This second *smarter* system should be the one delivered to the customer, since delivery of the pilot system would cause nothing but agony to the customer, and possibly ruin the system's reputation and maybe even the company's.

Code Freeze and Versioning

- Software is invisible. Therefore, many things only become apparent once a certain amount of work has been done on a new system, allowing a user to experience it.
- This experience will yield insights, which will change a user's needs or the perception of the user's needs.
- The system should, therefore, be changed to fulfill the changed requirements of the user.
- This can only occur up to a certain point, otherwise the system may never be completed.
- At a certain date, no more changes would be allowed to the system and the code would be frozen.
- All requests for changes should be delayed until the next version of the system.

Specialized Tools

- Instead of every programmer having his own special set of tools, each team should have a designated tool-maker who may create tools that are highly customized for the job that team is doing,
- e.g., a code generator tool that spits out code based on a specification.
- In addition, system-wide tools should be built by a common tools team, overseen by the project manager.

- The Mythical Man-Month
  - Assigning more programmers to a project running behind schedule will make it even later
  - time required for the new programmers to learn about the project
  - the increased communication overhead.
- Group Intercommunication Formula: $\frac{n(n-1)}{2}$
- Example: 50 developers -> $\frac{50(50-1)}{2} = 1225$ channels of communication

Second System

- The second system an engineer designs is the most dangerous system he will ever design
- tend to incorporate all of the additions he originated but did not add (due to the inherent time constraints) to the first system.
- an engineer should be mindful that he is susceptible to over-engineering it.

The Manual

- What the chief architect produces are written specifications for the system in the form of the manual.
- It should describe the external specifications of the system in detail, i.e., everything that the user sees.
- The manual should be altered as feedback comes in from the implementation teams and the users.

Formal Documents

- Every project manager should create a small core set of formal documents which acts as the roadmap as to what the project objectives are
  - how they are to be achieved,
  - who is going to achieve them,
  - when they are going to be achieved, and
  - how much they are going to cost.
- These documents may also reveal inconsistencies that are otherwise hard to see.

Communication

- To avoid disaster, all the teams working on a project should remain in contact with each other in as many ways as possible (email, phone, meetings, memos etc.)
- Instead of assuming something, the implementer should ask the architects to clarify their intent on a feature he is implementing, before proceeding with an assumption that might very well be completely incorrect.

Lowing costs

- There are two techniques for lowering software development costs that Brooks writes about:
- Implementers may be hired only after the architecture of the system has been completed (a step that may take several months, during which time prematurely-hired implementers may have nothing to do).
- Another technique Brooks mentions is not to develop software at all, but to simply buy it “off the shelf” when possible.

- Select a Chapter I have not gone over (not chapter 1, 2, 3, 4, 5, 7) and prpare a presentation.
- Key questions:
  1. What is the problem?
  2. Are there any solutions?
  3. How relevant is the chapter today?
  4. How does this help with software testing & maintenance?
- Presentation should be 10-20 minutes + questions.

Reading

Chapters 16 & 17 of the book.