Software Engineering

Part 7: *Software Project Planning and Management*

Prof. Dr. Stefan Leue

University of Konstanz
Chair for Software Engineering

Stefan.Leue@uni-konstanz.de
http://www.inf.uni-konstanz.de/~soft

Winter Term 2005/2006

Course Outline

1. Introduction and History

- History and Motivation
- Software Crisis
- Software Process

2. Object-Oriented Modeling

- Software Modelling
- Modelling Object Oriented Systems
- The Unified Modeling Language

3. Requirements and Early Life-Cycle Engineering

- Requirements Elicitation
- Software Specification
- State Machines and Petri Nets
- Object Oriented Analysis

Course Outline

4. Software Design

- Classical and Object-Oriented Design
- Design of Concurrent and Distributed Systems
- Interfaces and Contracts
- Software Architecture and Design Patterns

5. Software Quality Assurance

- Reviews and Inspections
- Testing
- Correctness Proofs
- (Software Metrics)

Course Outline

6. Software Production Process

Evolutionary Models

Spiral Model

Unified Process

Maturity Assessment

7. Software Project Planning and Management

Project Group

Staffing and Scheduling

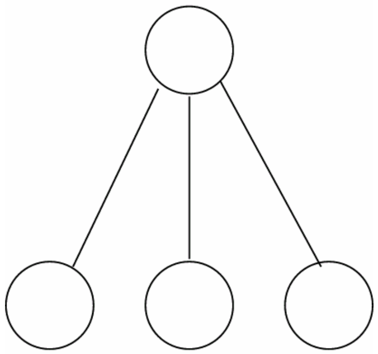
Software Size Metrics

Cost and Effort Estimation

Software Project Groups

- **How to choose your project partners…**
  - Goals
    - maximize equitable distribution of workload and
    - minimize tension and resentment.
  - Guidelines
    - Choose partners who have similar work habits to your own.
    - Choose partners who have similar goals and expectations.
    - Choose partners who have similar abilities.
    - Choose partners who have similar workloads and resources.
    - At least one member should have good communication skills.
    - Pay attention to personal dynamics.

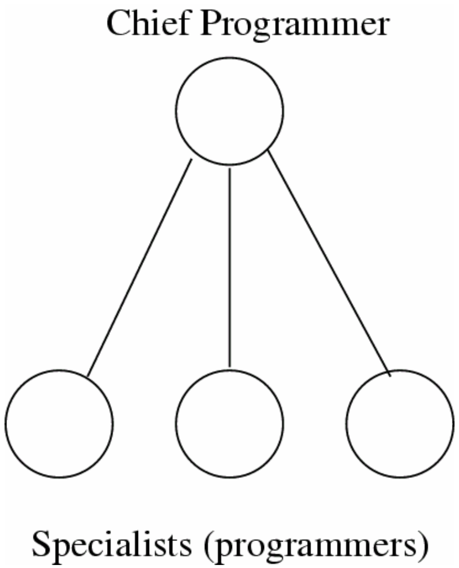
Software Project Groups



- **Centralized Control**
  - Chief programmer makes all of the important decisions, is responsible for coordinating efforts of other members of the team.
    - Advantages:
      - Speeds up project development - centralized decision making.
      - Less need for communication.
    - Disadvantages:
      - Single point of failure.
      - Little personnel development.
      - Lower team morale
        - Specialists may feel like slave of chief programmer!
  - Centralized control acceptable if chief is really much more competent than other team members.

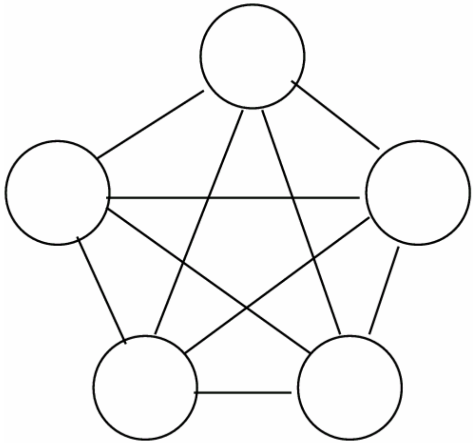
software engineering

Software Project Groups



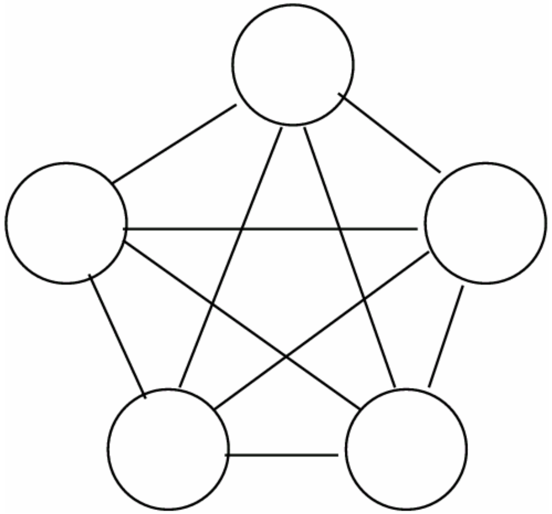
- **Rotating Centralized Control**
  - Can alleviate some of the negatives by rotating chief programmer.
  - Everyone must respect previous decisions.
  - Learn to manage as well as learn to be managed!

Software Project Groups



- **Decentralized Control**
  - everyone participates and has an equal say
  - Requires strategy for democratic desision making:
    - Decision by majority.
      - Difficult with small or even-membered teams.
    - Decision by unanimous consent.
      - Discussion proceeds and decision revised until all members decide decision is optimal.
      - Time consuming, but highest morale.
    - Decision by consensus.
      - Team discussion is open and responsive to all views.
      - Decision may not be considered optimal, but is understood, influenced by, and accepted by all participants.

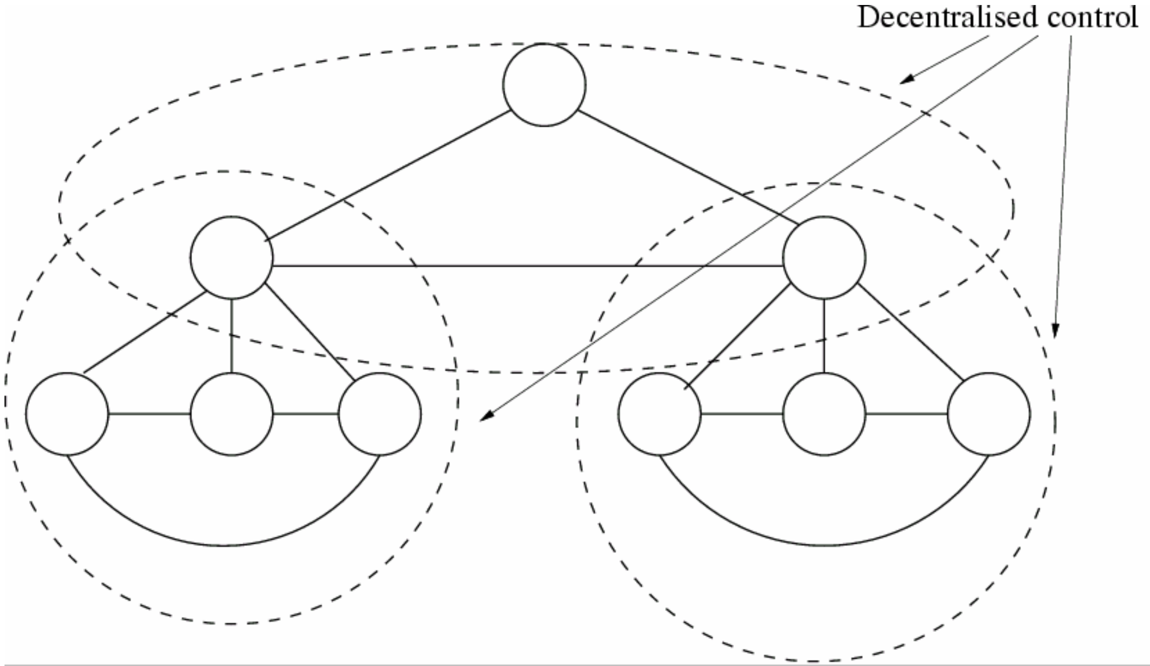
Software Project Groups



- **Decentralized Control**
  - Advantages:
    - Good personnel development.
    - Group can invent better solutions than individual.
    - High morale.
  - Disadvantages:
    - Many communication paths.
    - Less initiative because of lack of authority and responsibility.

software engineering

Software Project Groups



♦ Mixed Control

- Team may contain too many members to have an effective democratic organization.
- Instead have hierarchical organization.
- Highest member of group represents group at next higher level.

Software Project Groups

Team Responsibilities: how to organize project and members.

- Decompose required behaviour into components.
- Clean decomposition of responsibilities.
- Define interfaces between components ($\leadsto$ parallel development).
- Integrate components into cohesive specification.

  Flip side of divide/conquer is gather/integrate.
- Define deadlines and members' responsibilities.

software engineering

Software Project Groups

Member Responsibilities: minimize resentment.

- Responsible for work delegated to them.
- Provide interfaces agreed upon.
- Adhere to any quality/style standards imposed by team.
- Meet the deadlines that group has established.

Members have the right to be free of "micro-management" - able to make his/her own decisions on the fine-details of his/her task without unnecessary interference by others in the team.

Software Project Groups

Adequate division of labour.

Problems:

- The workload on every team member should be approximately equal.

- Develop a schedule that makes sure that tasks assigned to individual project members can be finished in time before system integration.

$\leadsto$ need to estimate effort necessary to implement the components into which each group has divided the problem.

Software Project Groups

Successful Group Meetings.

A useful forum for meeting is one in which:

Chair is responsible for keeping things on schedule,
recorder keeps summary notes,
and these responsibilities can **rotate**.

Things that may go on the agenda:

- members give progress reports
- record progress towards goals
- review/revise progress chart
- review/revise workloads based on schedules
- set deadlines, goals for next meeting
- review/reconsider past decisions that may not be working
- evaluate team dynamics
- air and resolve difficulties

Software Project Scheduling

What to do if a SW project falls behind schedule?

- **Hopeful optimism.**

  "Early requirements took a little longer than expected - but now we really understand what is going on. So the rest of the project is bound to take less time."

- **Team expansion.**

  From Fred Brooks' book "Mythical Man Month":

  - new people need time to catch up to speed
  - new people need training
  - project people need to train new people
  - increases amount and complexity of communication

Software Project Scheduling

- **Scope reduction.**

- **Feature elemination.**

  Better to have working system that supports some features (and promise late delivery of other features) than to have non-working system.

  (No applicability to the course project :-))

- **Deliver 'cheap' version of features.**

  For example, deliver functionality, but at lower than required performance. Promise later tuning.

Effort Estimation

Software Project Planning.

2 tasks:

- Analysis.
- Effort Estimation.

Most common cause of software development failures is bad (too optimistic) planning $\rightsquigarrow$ good estimation methods.

Effort Estimation

Potential risks in project estimation:

- Variation in product **structure** wrt. past products.
- Relative **complexity** wrt. past work.
- Relative **size** wrt. past work.

Strategy: keep these risks as low as possible.

$\leadsto$ keep historical database to help in future estimations.

Software Size Metrics

Metrics for **Software Productivity**.

1. Size-oriented metrics.

- Basis: lines of code (LOC, KLOC), other metrics based on

  delivered source instructions (DSI), or
  non-commented source statements (NCSS).

  - Productivity: KLOC/person-month.
  - Quality: faults/KLOC.
  - cost: $/KLOC.
  - documentation: pages/KLOC.

- Problem: imperfect measure, penalizes well-written code.

- However, empirical research has shown that most other metrics correlate well with LOC.

Software Size Metrics

2. Function-oriented metrics.

- Mainly **business-oriented** application development.
- Focus on quantifying program functionality.
- **Subjective** assessment of complexity.

Complexity is weighted sum of function points (FPs):

# Inputs	4
# Outputs	5
# Inquiries	4
# Files	10
# Interfaces	7

Software Size Metrics

Example:

Compute the function point FP for a payroll program that reads a file of employees and a file of information for the current month and prints check for all the employees. The program is capable of handling an interactive command to print an individually requested check immediately.

2 inputs	x 4	= 8
1 outputs	x 5	= 5
1 inquiry (2 inquiries?)	x 4	= 4
2 files	x 10	= 20
2 interfaces	x 7	= 14
Sum FPs		51

Software Size Metrics

Use of FPs:

- **Quantitative** characteristics of SW development:

  Productivity: FP/person-month.
  Quality: faults/FP.
  Cost: $/FP.
  Documentation: pages/FP.

- **Critique**:

  - **Adequacy:** e.g., does every *input* involve the same complexity?

  - **Solution:** attempt to assign individual FP weights to different functions ($\leadsto$ calibrarion of the method).

- Overall, appears to be more suitable measure than KLOC.

Software Size Metrics

- **Object Points**
  - third generation languages (3GL): high-level programming languages
  - fourth generation languages (4GL): prototyping languages, such as SQL, Oracle SQL*Plus, Oracle Reports, LINC, Metafont, Mathematica,...
  - object point estimates: weighted sum of
    - number of separate screens that are displayed
      - simple screens: 1
      - complex screens: 2 – 3
    - number of reports produced
      - simple: 2
      - moderately complex: 5
      - likely to be difficult: 8
    - number of 3GL objects that must be developed to supplement 4GL code
      - each 3GL module: 10
  - advantage
    - less reliant on concrete design, can be determined earlier from software specification

software engineering

Software Size Metrics

- **Obtaining LOC from FP/OP**
  - use FP/OP to estimated final code size
  - use history based approach
    - size = AVC x #FP
      - AVC: average #loc per FP (or OP)
        - typical values: 200-300 LOC/FP for assembler code, 2-40 LOC/FP for 4GL languages
- **Programmer Productivity**
  - can greatly vary (up to a factor 10)
  - average in large teams, small teams depend more on individual capability
  - range
    - 30 loc / programmer-month for complex embedded systems
    - 900 loc / programmer-month for well-understood application domains
    - 4-50 op/ programmer-month
  - however, loc/time is not a very reliable productivity measure when quality is crucial
    - process that focuses on constant code simplification and improvement
    - management may make inadequate judgements about individuals
  - software reuse not considered

Effort Estimation

Metrics for **SW project estimation.**

Project parameters:

- Development costs (programmer months).
- Development interval.
- Staffing levels.
- Maintenance costs.

General Approaches:

- Expert judgement.
- Analytical/empirical assessment models.

Cost Estimation Techniques

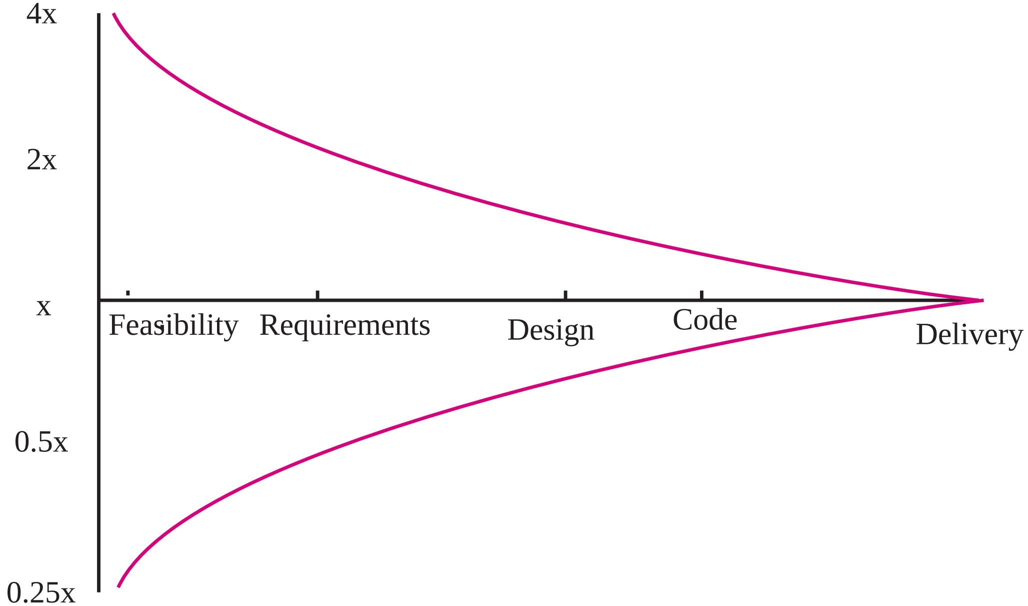
- **Algorithmic Cost Modeling**
- **Expert Judgement**
  - first stage: several experts estimate costs independently
  - second stage: discuss and reconcile estimates until agreement reached
- **Estimation by Analogy**
  - compare and establish analogy to other projects completed in same application domain
- **Parkinson's Law**
  - "work expands to fill the time available"
  - time and resources available rather than objective assessment determine cost
    - if 12 months time and 5 developers available, it will cost 60 person months
- **Pricing to Win**
  - cost determined by whatever customer is willing/able to spend
  - functionality adjusted according to budget
  - not that uncommon in practical project agreements
    - first agree on an overall goal and budget
    - then negotiate detailed specification of functionality, taking into consideration what can be implemented within the agreed budget limits

Algorithmic Cost Modeling

- **Algorithmic Cost Models**
  - built by analyzing completed projects
  - extracting mathematical models that allow one to predict parameters for future projects
  - basic model
    - effort = A x size^{B} x M
      - effort: predicted efforet
      - A: constant factor, represents local organizational practices and type of software to be developed
      - size: assessment of code size, e.g., loc, function/object points
      - B: constant representing effort for very large projects
        - assumption: effort increases exponentially with size of project
        - typical values: 1-1.5
      - M: constant determined according to process, product and development attributes
    - precondition
      - repeated application of this model to various projects
      - repeated calibration: adjustment of constants (in particular B and M) based on the accuracy of the derived estimates

Size Estimation Accuracy

- **The size of a software system can only be known accurately when it is finished**
- **Several factors influence the final size**
  - Use of COTS and components
  - Programming language
  - Distribution of system
- **As the development process progresses then the size estimate becomes more accurate**



 Software Engineering, 6th edition

Effort Estimation

CoCoMo (Constructive Cost Model)

Decompose SW into small enough units to be able to estimate LOC, uses KDSI and PM (= 152 hours).

Basic Model

$$PM = 2.4 * KDSI^{1.05}$$

$$T_{DEV} = 2.5 * PM^{0.38}$$

software engineering

Effort Estimation

Intermediate Model

Classify SW project:

Organic: Small SW development team, familiar in-house environment, extensive experiences, specifications negotiable.

Embedded: Firm, tight constraints, generally less known territory.

Semi-detached: in between organic and embedded.

Effort Estimation

Step I: Nominal effort estimation.

$$PM_{nom} = a_i * KDSI^{b_i}$$

	a_i	b_i
organic	3.2	1.05
semi-detached	3.0	1.12
embedded	2.8	1.20

Effort Estimation

Step II: Determine effort multipliers.

- Attribute groups:

  1. **Product**: required reliability, product complexity.
  2. **Computer**: execution time, primary storage requirements.
  3. **Personnel**: analyst and programmer's capabilities; application, machine and programming language experience required.
  4. **Project**: modern programming practices, use of CASE tools, project scheduling realistic.

- Total 15 attributes, [Ghezzi et al.], Table 8.5.

- Effort adjustment factor:

$$EAF = \prod_{i=1}^{15} attribute_i$$

Effort Estimation

CoCoMo attributes

Cost Drivers	Very low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	.75	.88	1.00	1.15	1.40	
Data base size		.94	1.00	1.08	1.16	
Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer attributes						
Execution time constraints			1.00	1.11	1.30	1.66
Main storage constraints			1.00	1.06	1.21	1.56
Virtual machine volatility*		.87	1.00	1.15	1.30	
Computer turnaround time		.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	.86	.71	
Applications experience	1.29	1.13	1.00	.91	.82	
Programmer capability	1.42	1.17	1.00	.86	.70	
Virtual machine experience*	1.21	1.10	1.00	.90		
Programming language experience	1.14	1.07	1.00	.95		
Project attributes						
Use of modern programming practices	1.24	1.10	1.00	.91	.82	
Use of software tools	1.24	1.10	1.00	.91	.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

Effort Estimation

Step III: Estimate actual **development effort:**

$$PM_{dev} = PM_{nom} * EAF$$

Step IV: Estimate **ideal project duration**:

$$T_{dev} = c_i * PM_{dev}^{d_i}$$

	c_i	d_i
organic	2.5	0.38
semi-detached	2.5	0.35
embedded	2.5	0.32

Effort Estimation

Example: CAD graphical editor

Function	est. KLOC
GUI control	2.3
2-D geometry	5.3
3-D geometry	6.8
DB management	3.35
Graphic display	4.95
Peripheral control	2.1
Design analysis	8.4
Sum est. KLOC	33.2

Non-COCOMO estimation:

- Approximate productivity: 620 LOC/PM.
- Estimated effort: 54 PM.
- Cost per PM = \$8,000 $\leadsto$ cost per LOC = 13 \$, project cost = \$431,000.

Effort Estimation

COCOMO-estimation:

- Assume: semi-detached $\rightsquigarrow$ nominal effort:

$$PM_{nom} = 3.0 * 33.2^{1.12} = 173$$

- Actual effort taking EAF into account:

$$PM_{dev} = PM_{nom} * EAF = 173 * 0.68 = 117.64$$

- Ideal project duration:

$$T_{dev} = 2.5 * 117.64^{0.35} = 7.31$$

$\rightsquigarrow$ You need roughly 16 programmers to work on it in parallel.

Sometimes suggested to multiply PM_{dev} and T_{dev} with a 1.2 adjustment factor.

software engineering

COCOMO 2

- **Rationale**
  - COCOMO: software development from scratch
  - COCOMO 2: reuse, component assembly, re-engineering, use of 4GLs
- **Levels**
  - COCOMO 2 is a 3 level model that allows increasingly detailed estimates to be prepared as development progresses
    - early prototyping level
      - effort estimates based on object points
    - early design level
      - after completion of requirements and possibly early, architectural design
      - estimates based on function points that are then translated to LOC
    - post-architecture level
      - when system design is available
      - estimates based on lines of source code

COCOMO 2

- **Early Prototyping Level**
  - designed for projects with
    - extensive use of prototyping/4GL languages
    - extensive reuse
  - based on standard estimates of developer productivity in object points/month
  - takes CASE tool use into account
  - formula is
    - $PM = (NOP \times (1 - \%reuse/100)) / PROD$
      - PM: the effort in person-months
      - NOP: the number of object points
      - PROD: the productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NOP/month)	4	7	13	25	50

COCOMO 2

- **Early Design Level**
  - estimates can be made after the requirements have been agreed
  - based on standard formula for algorithmic cost estimation models
    - $PM = A \times Size^B \times M + PM_m$ (Size in KLOC), where
      - A = 2.5 in initial calibration
      - B: increased effort for large projects, varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity
      - $M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$
        * PERS: personnel capability
        * RCPX: product reliability and complexity
        * RUSE: required reuses
        * PDIF: platform difficulty
        * PREX: personnel experience
        * FCIL: support facilities
        * SCED: scheduling constraints

        all to be estimated on a 1 (very low) to 6 (very high) scale
      - $PM_m = (ASLOC \times (AT/100)) / ATPROD$: manual effort for projects with large scale code generation

COCOMO 2

- **Early Design Level**
  - estimates can be made after the requirements have been agreed
  - based on standard formula for algorithmic cost estimation models
    - $PM = A \times Size^B \times M + PM_m$ (Size in KLOC), where
      - $PM_m = (ASLOC \times (AT/100)) / ATPROD$: manual effort for projects with large scale code generation
        - * ASLOC: # of automatically generated lines of code
        - * AT: percentage of automatically generated code
        - * ATPROD: productivity for automatically generated coding

        i.e., productivity depends number of automatically generated modules

COCOMO 2

- **Post-Architecture Level**
  - uses same formula as early design estimates
  - estimate of size is adjusted to take into account
    - requirements volatility
      - $\rightarrow$ rework required to support change
    - extent of possible reuse
      - reuse is non-linear and has associated costs so this is not a simple reduction in LOC
  - formula taking reuse into account
    - ESLOC = ASLOC $\times$ (AA + SU +0.4DM + 0.3CM +0.3IM)/100
      - ESLOC is equivalent number of lines of new code.
      - ASLOC is the number of lines of reusable code which must be modified,
      - DM is the percentage of design modified,
      - CM is the percentage of the code that is modified
      - IM is the percentage of the original integration effort required for integrating the reused software.
      - SU is a factor based on the cost of software understanding
        - complex, unstructured: 50, well-written, oo: 10
      - AA is a factor which reflects the initial assessment costs of deciding if software may be reused
        - range: 0 - 8

COCOMO 2

- **Exponent Term B**
  - depends on 5 scale factors

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience, Extra high means that the organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client only sets general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis, Extra high means a complete a thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions, Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organisation. The computation of this value depends on the CMM Maturity Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

©Ian Sommerville 2000 Software Engineering, 6th edition

  - sum/100 is added to 1.01
  - example
    - precedentedness - new project - 4
    - development flexibility - no client involvement - very high = 1
    - architecture/risk resolution - no risk analysis - very low = 5
    - team cohesion - new team - nominal = 3
    - process maturity - some control - nominal = 3

    $\Rightarrow$ scale factor is therefore 1.17

COCOMO 2

- **Multipliers**
  - product attributes
    - concerned with required characteristics of the software product being developed
  - computer attributes
    - constraints imposed on the software by the hardware platform
  - personnel attributes
    - multipliers that take the experience and capabilities of the people working on the project into account.
  - project attributes
    - concerned with the particular characteristics of the software development project

COCOMO 2

♦ Product Attributes - Project Cost Drivers

Product attributes			
RELY	Required system reliability	DATA	Size of database used
CPLX	Complexity of system modules	RUSE	Required percentage of reusable components
DOCU	Extent of documentation required		
Computer attributes			
TIME	Execution time constraints	STOR	Memory constraints
PVOL	Volatility of development platform		
Personnel attributes			
ACAP	Capability of project analysts	PCAP	Programmer capability
PCON	Personnel continuity	AEXP	Analyst experience in project domain
PEXP	Programmer experience in project domain	LTEX	Language and tool experience
Project attributes			
TOOL	Use of software tools	SITE	Extent of multi-site working and quality of site communications
SCED	Development schedule compression		

COCOMO 2

- **Example**
  - effects of cost drivers

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128, 000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

COCOMO 2 - Project Planning

Option	RELY	STOR	TIME	TOOLS	LTEX	Total effort	Software cost	Hardware cost	Total cost
A	1.39	1.06	1.11	0.86	1	63	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	*1.39*	*1.06*	*1.11*	*0.86*	*0.84*	*51*	*769008*	*100000*	*897490*
E	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

- **Algorithmic Cost Models for Risk and Options Assessment**
  - example: embedded spacecraft system
    - must be reliable, minimize weight (number of chips)
      - $\Rightarrow$ multipliers on reliability and computer constraints > 1
    - cost components
      - Target hardware
      - Development platform
      - Effort required
    - option D (use more experienced staff) appears to be the best alternative
      - high associated risk as experienced staff may be difficult to find
    - option C (upgrade memory) has a lower cost saving but very low risk
    - overall, the model reveals the importance of staff experience in software development

COCOMO 2 - Project Duration and Staffing

- **Estimation of Development Time**
  - calendar time can be estimated using a COCOMO 2 formula
    - $TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$
      - PM is the effort computation and
      - B is the exponent computed as discussed above (B is 1 for the early prototyping model)
      - time required is independent of the number of people working on the project
- **Conclusion**
  - algorithmic cost estimation is difficult because of the need to estimate attributes of the finished product
  - the COCOMO model takes project, product, personnel and hardware attributes into account when predicting effort required
  - algorithmic cost models support quantitative option analysis
  - the time to complete a project is not proportional to the number of people working on the project

Effort Estimation

Boehm, 1981:

"Today, a software cost estimation is doing well if it can estimate cost to within 20% of actual cost, 70% of actual time, and on its own turf... This is not as precise as we might like, but accurate enough to provide a good deal of help in software engineering economic analysis and decision making."

Staffing

- **Effort Distribution over time: Raleigh-Nordon Curve**
  - Indicates total manpower needed at any given point in time
  - Applicable only to more conventional, waterfall-type process models

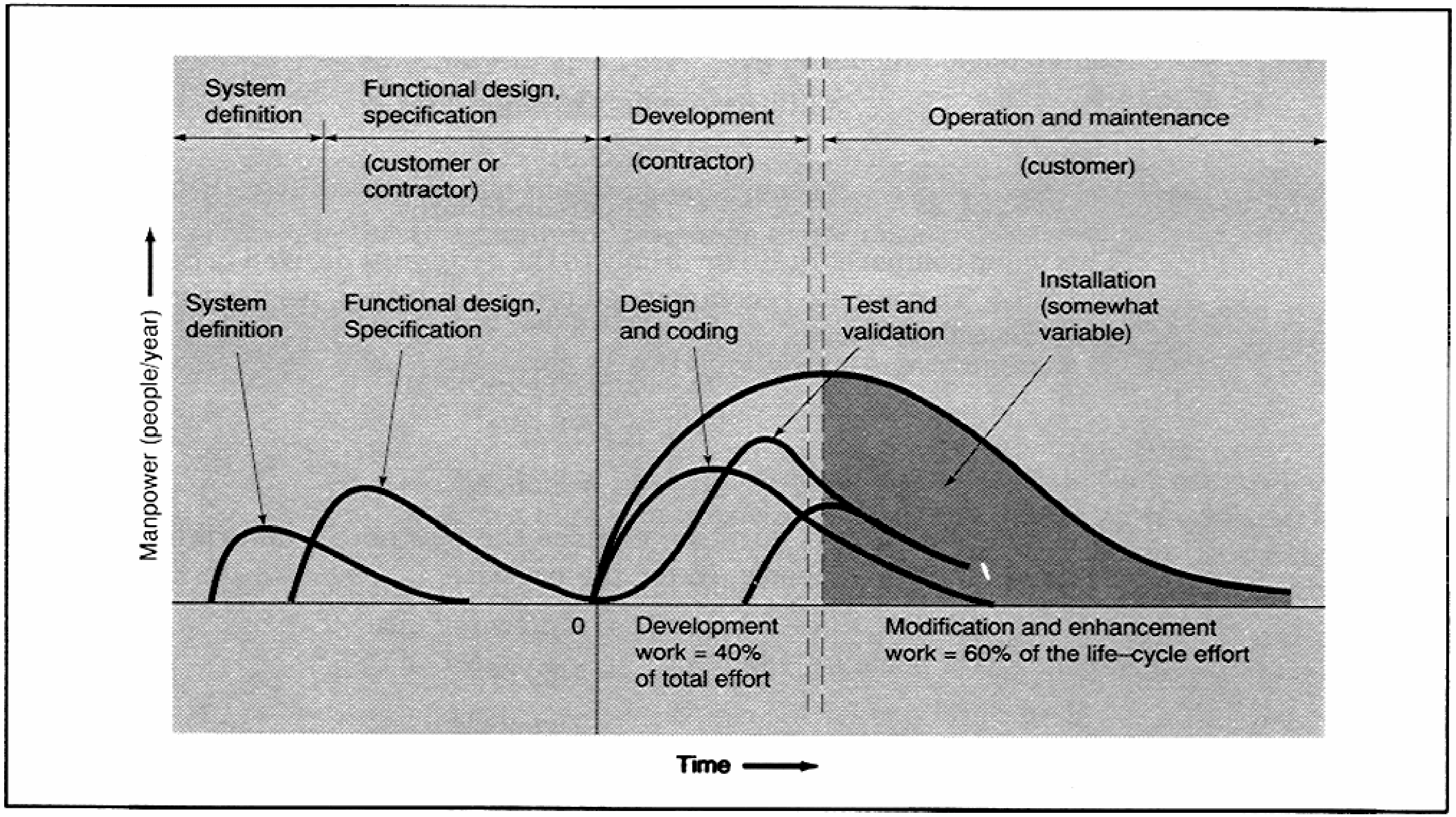


FIGURE 3.10. Effort distribution—large points. (*Source: Software Cost Estimating and Life Cycle Control, IEEE Computer Society Press, 1980, p. 15. Reproduced with permission.*)

Scheduling

- **Work Breakdown Structure (WBS)**
  - Identification of all activities that a project must undertake
  - Decide which things need to be done, disregard their order
  - Refine until project manager can estimate develoment time for every activity

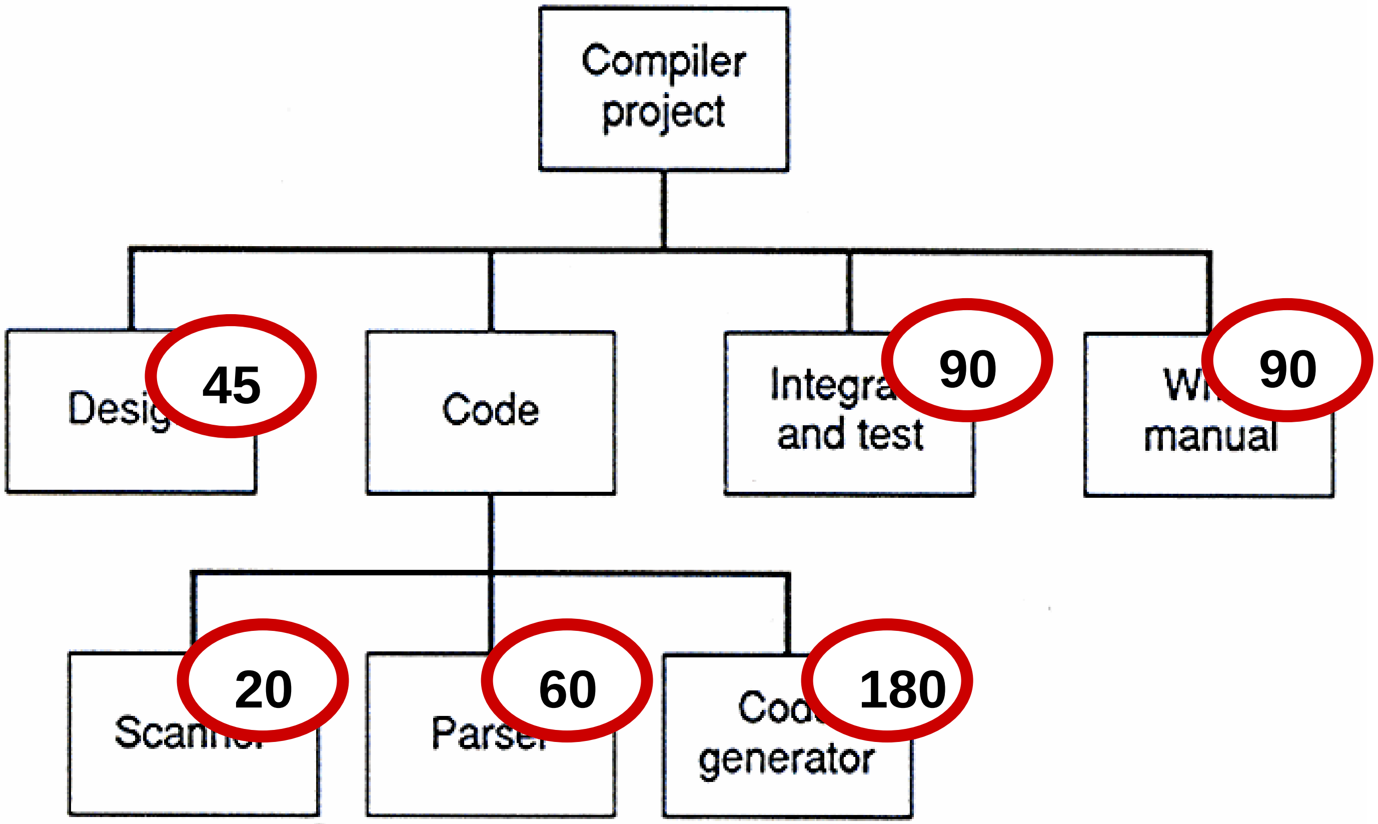


Figure 8.1 Work breakdown structure for a compiler project.

Scheduling

- **Program Evaluation and Review Technique (PERT) Chart**
  - Determine dependencies of tasks, i.e., second taks may not start before completion of first task
  - Critical path: delay in any activity along critical path will inevitably delay project completion

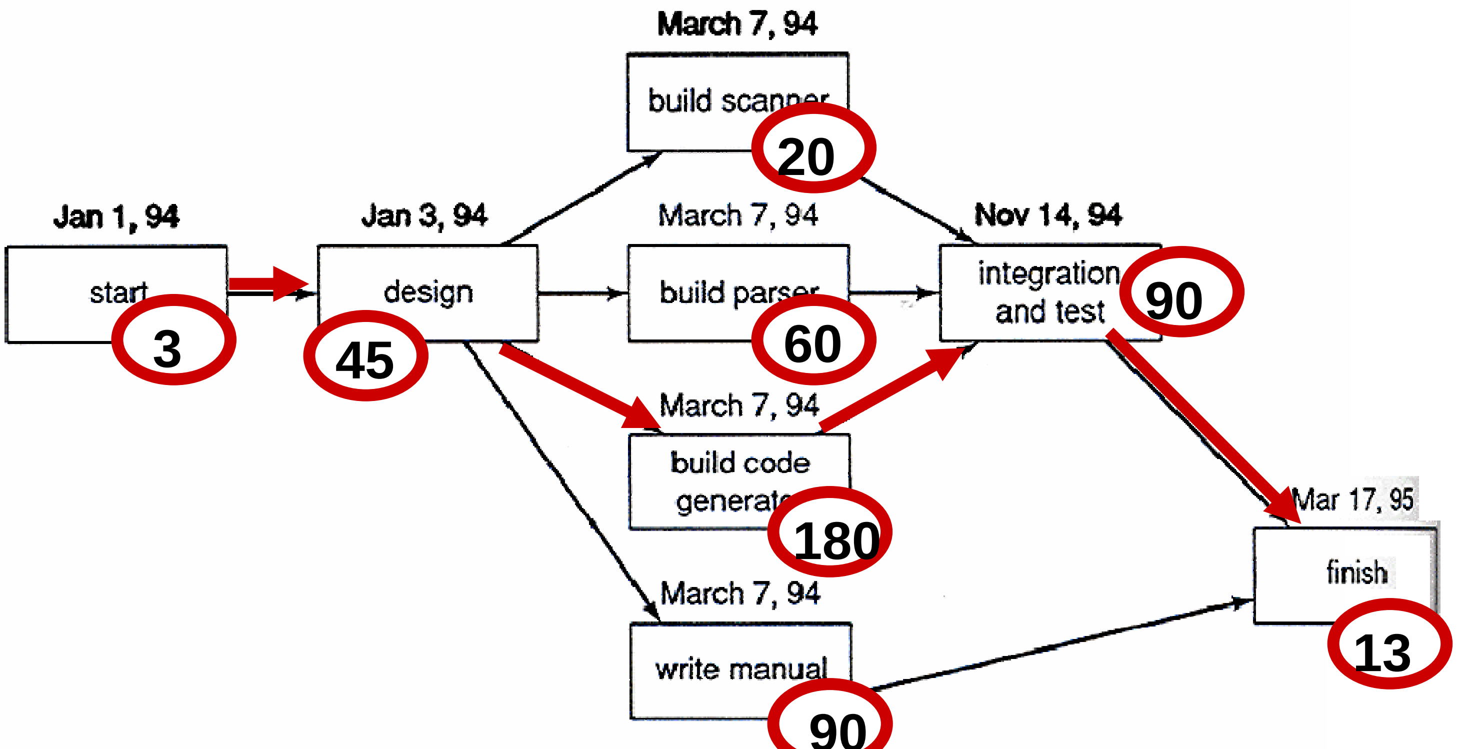


Figure 8.4 PERT chart for a simple compiler project. Activities on the critical path are shown in bold.

Scheduling

- **Program Evaluation and Review Technique (PERT) Chart**
  - Milestones
    - Tasks that will be key indicators of progress, e.g., completion of design, completion of code generator, completion of integration and testing
    - List their planned completion dates as milestones
    - Track actual project progress against milestones

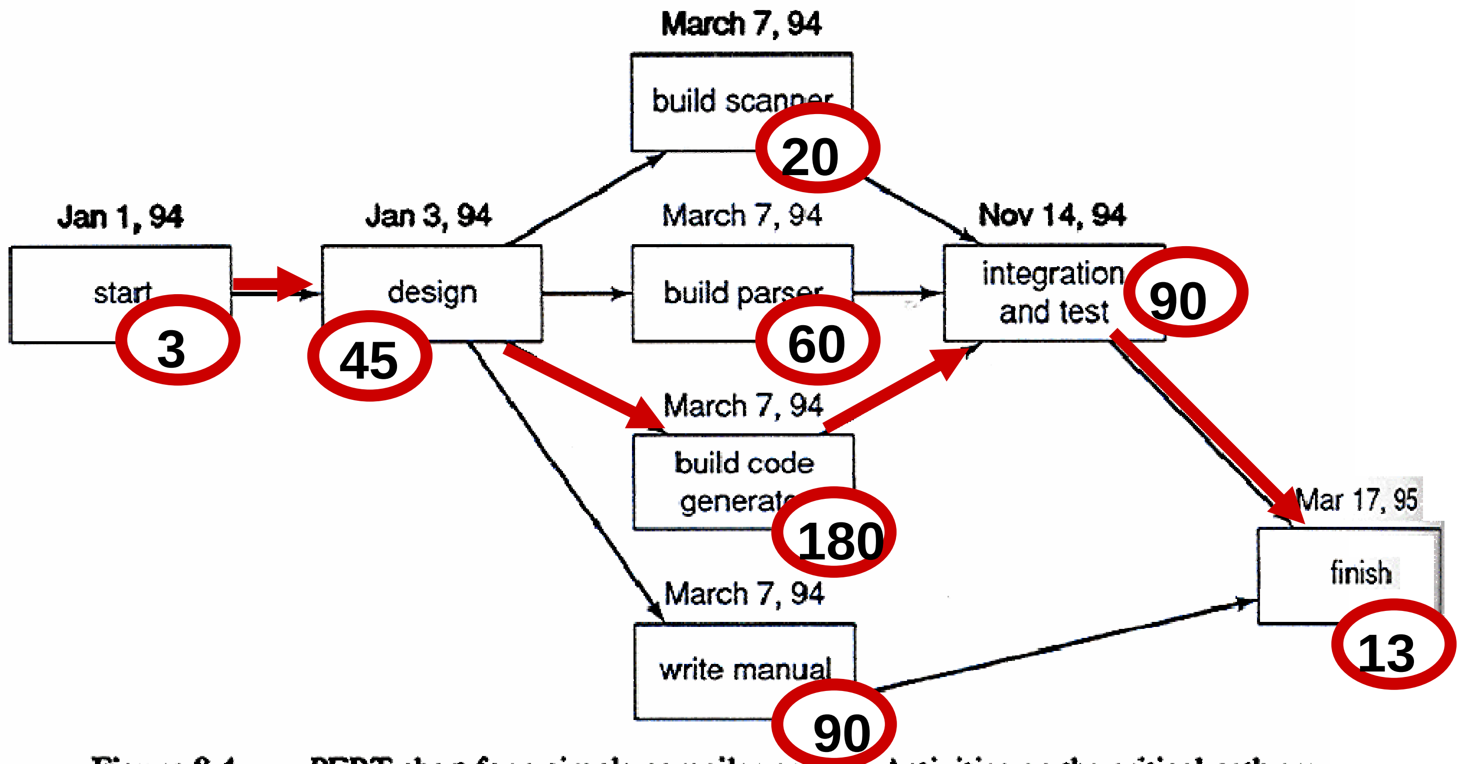


Figure 8.4 PERT chart for a simple compiler project. Activities on the critical path are shown in bold.

Scheduling

- **GANTT Chart**
  - Scheduling of activities in WBS chart
    - When should activities start/end
    - What is the slack
      - Note: no slack on critical path

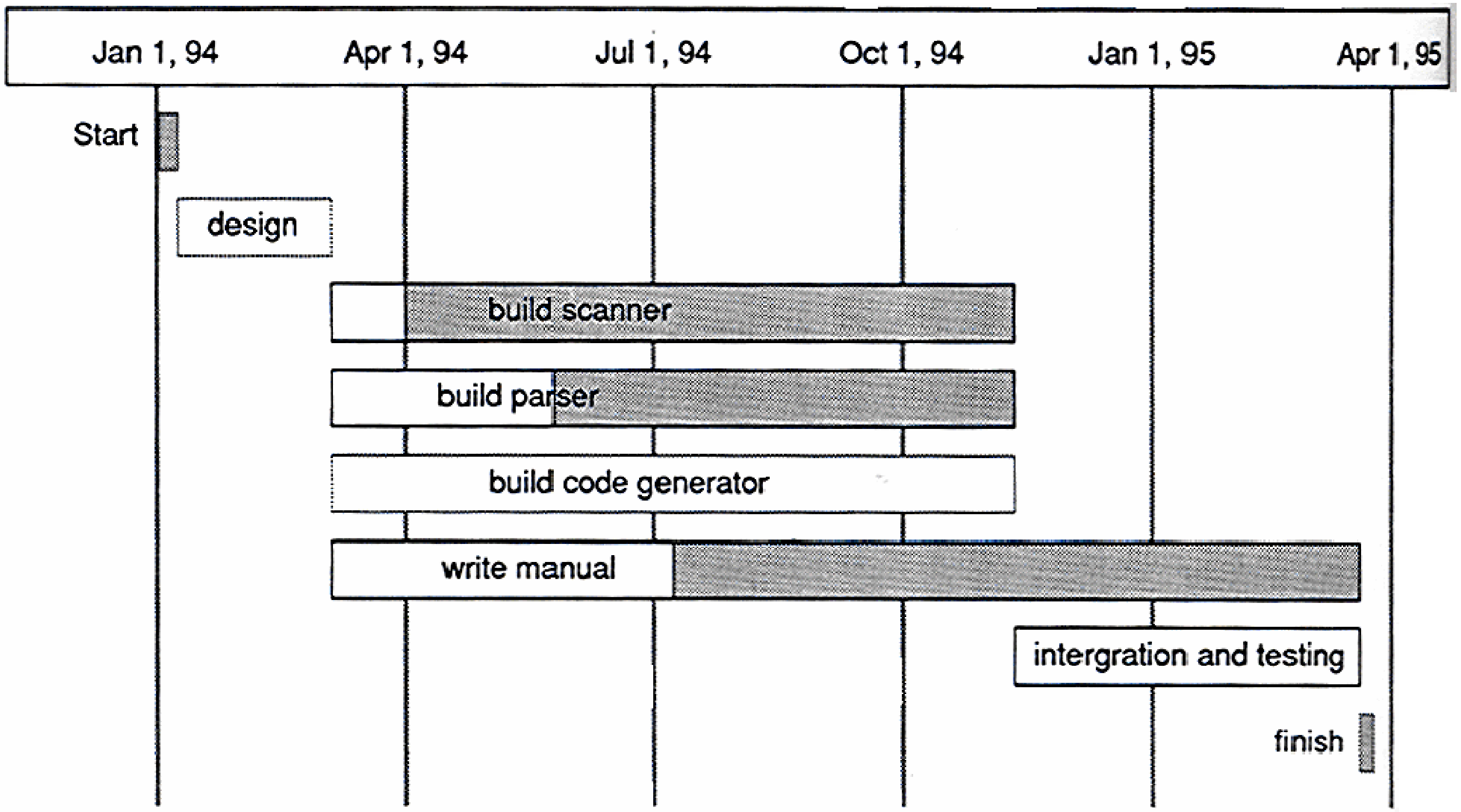


Figure 8.2 Gantt chart for a simple compiler project.

Scheduling

- **GANTT Chart**
  - Scheduling of activities in WBS chart
    - Scheduling of project members
      - E.g., under availability constraints

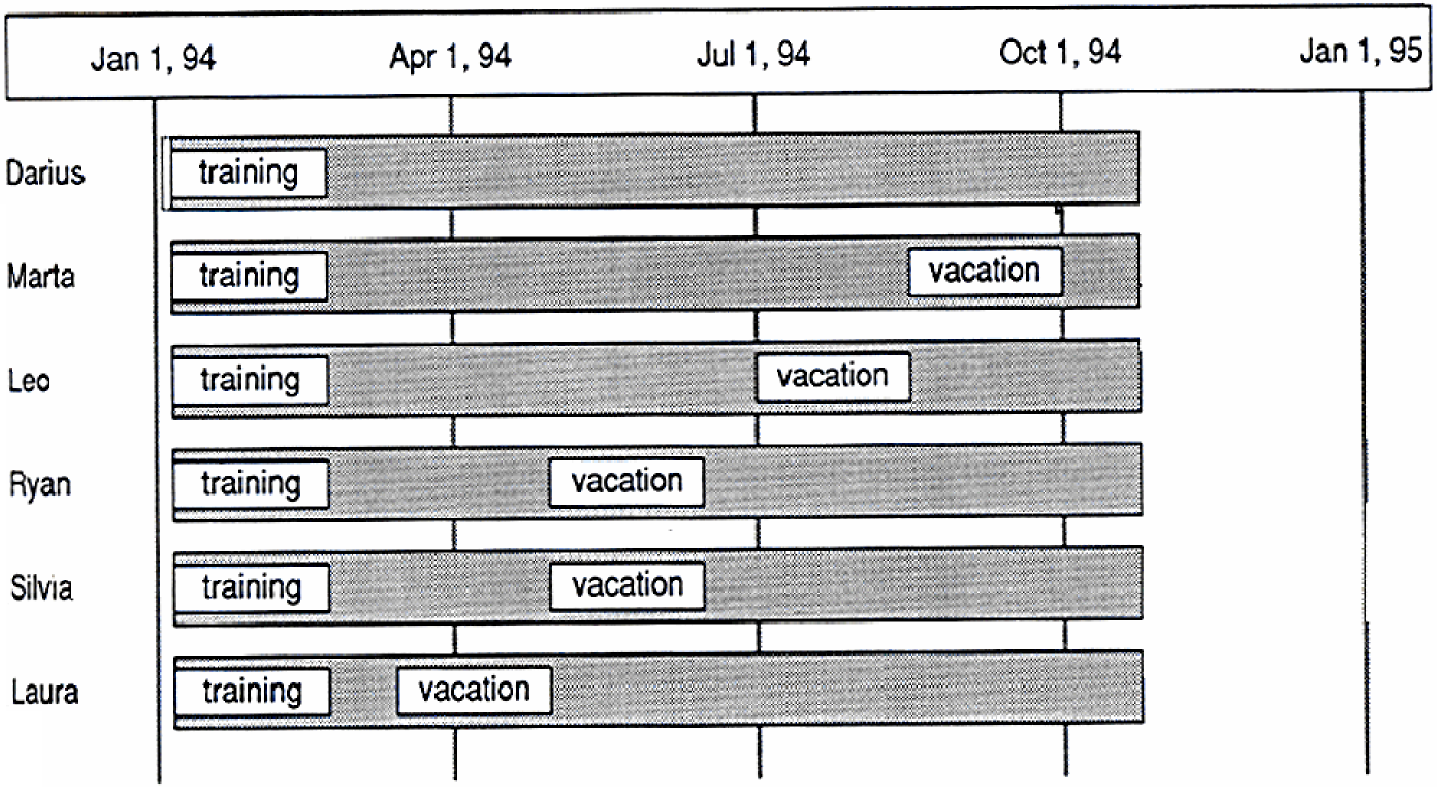


Figure 8.3 **Gantt chart for scheduling six engineers.**

software engineering