Software Engineering

David Vernon

Software Engineering 2

- The goal of this course is to provide a working knowledge of the techniques for the
  - ❑ Estimation
  - ❑ Design
  - ❑ Building
  - ❑ Quality assurance

  of software projects

Software Engineering 2

- Course Texts:

  Software Engineering, Ian Sommerville, Addison Wesley Longmans

  Software Engineering – A Practitioner's Approach, Roger Pressman, McGraw-Hill

Motivation for Studying Software Engineering

Software Engineering

- More than half of the world’s current software is “embedded” in other products, where it determines their functionality

- Software implements market-differentiating capabilities

  - automobiles, air travel, consumer electronics, financial services, and mobile phones, domestic appliances, house construction, medical devices, clothing and social care.

Software Engineering

- Competitive advantage
  - □ Is based on the *characteristics* of products sold or services provided
    - *functionality, timeliness, cost, availability, reliability, interoperability, flexibility, simplicity of use*
  - □ 90% of the innovation in a modern car is software-based
  - □ Innovation will be delivered through quality software
  - □ Software determines the success of products and services

Software Engineering

A Review of Software Engineering

Software Engineering

- A simple definition of Software Engineering:

  Designing, building and maintaining large software systems

Software Engineering

- A More Detailed Definition:

Software engineering is the branch of systems engineering concerned with the development of large and complex software-intensive systems

Software Engineering

- A More Detailed Definition:

It focuses on:

- the real-world goals for, services provided by, and constraints on such systems;
- the precise specification of system structure and behaviour, and the implementation of these specifications;
- the activities required in order to develop an assurance that the specifications and real-world goals have been met;
- the evolution of such systems over time and across system families.

Software Engineering

- A More Detailed Definition:

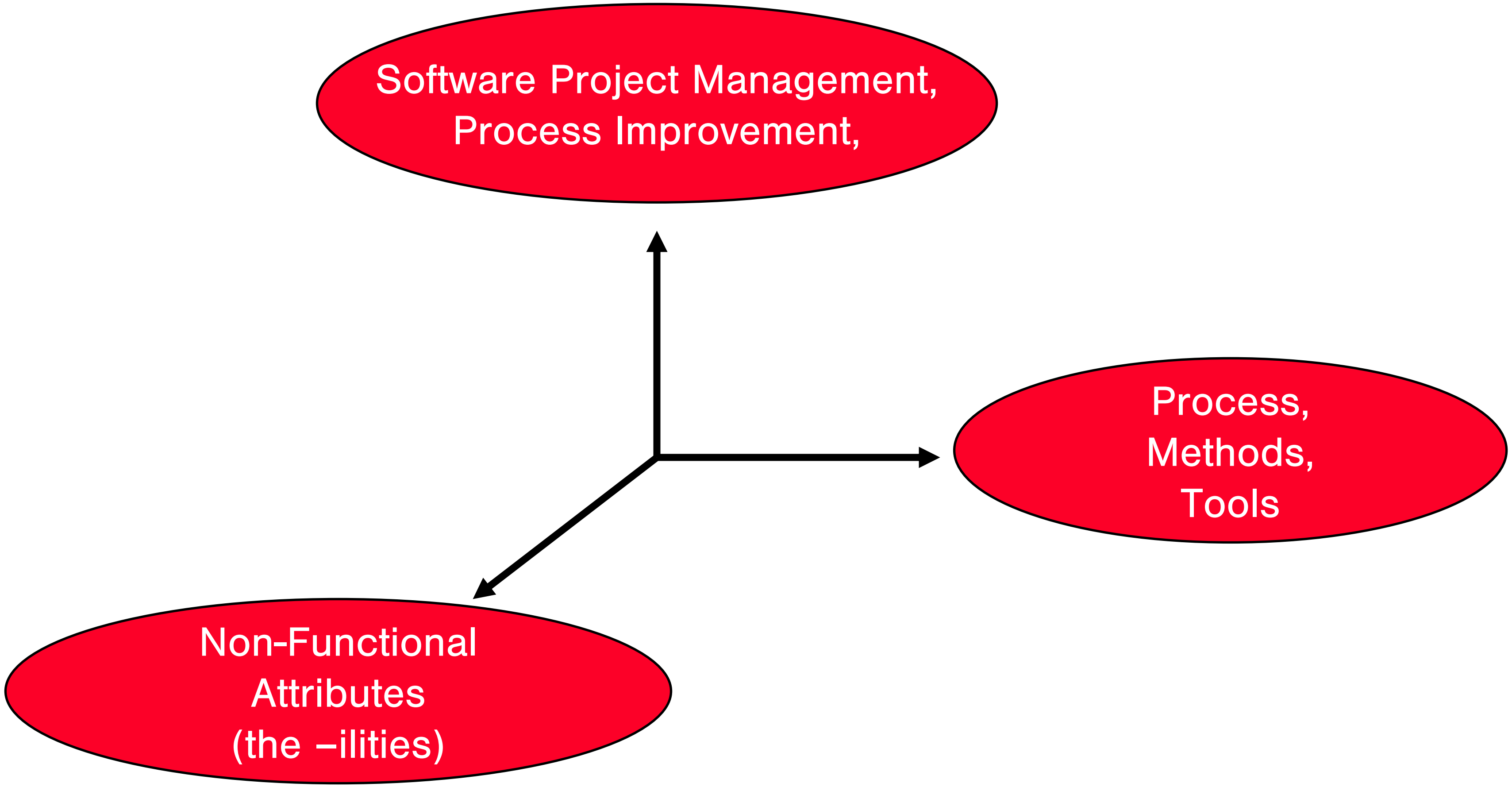
It is also concerned with the:

- Processes
- Methods
- Tools

for the development of software intensive systems in an *economic* and *timely* manner.

A. Finelstein and J. Kramer, “Software Engineering: A Road Map” in "The Future of Software Engineering" , Anthony Finkelstein (Ed.), ACM Press 2000

Software Engineering



The Three Dimensions of Software Engineering

Software Engineering

- People
  - People matter
  - Software engineering is as much about the organization and management of people as it is about technolgy
  - People use the system
  - People design the system
  - People build the system
  - People maintain the system
  - People pay for the system!
- Product
- Process

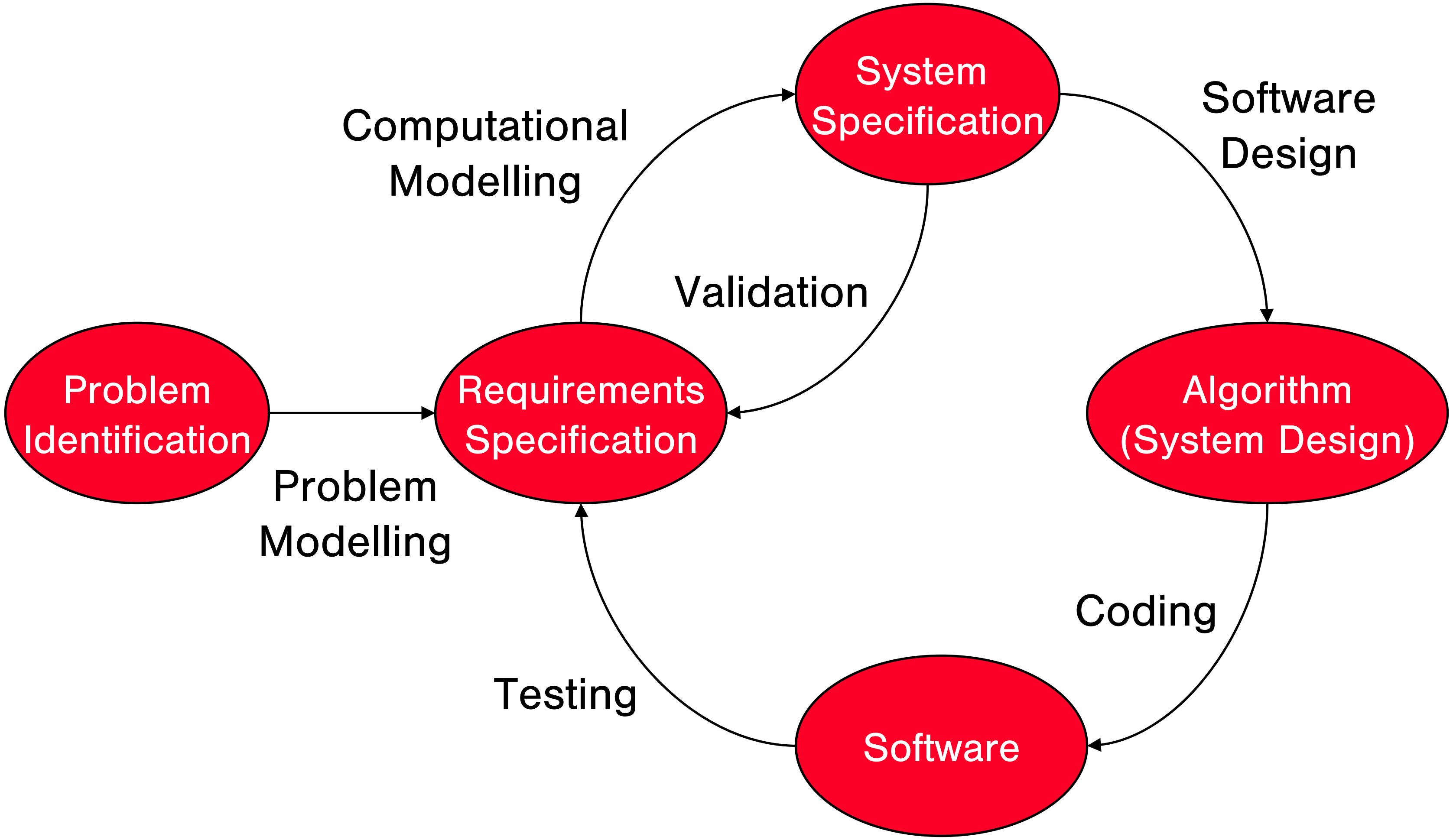
Software products

- Generic products
  - ❑ Stand-alone systems which are produced by a development organisation and sold on the open market to any customer
- Bespoke (customised) products
  - ❑ Systems which are commissioned by a specific customer and developed specially by some contractor
- Most software expenditure is on generic products but most development effort is on bespoke systems
- The Trend is towards the development of bespoke systems by integrating generic components (which must themselves be interoperable)

The software process

- Structured set of activities required to develop a software system
  - ❑ Specification
  - ❑ Design
  - ❑ Validation
  - ❑ Evolution
- Activities vary depending on the organisation and the type of system being developed
- Must be explicitly modelled if it is to be managed

The software process



Engineering process model

- Specification - set out the requirements and constraints on the system
- Design - Produce a paper model of the system
- Manufacture - build the system
- Test - check the system meets the required specifications
- Install - deliver the system to the customer and ensure it is operational
- Maintain - repair faults in the system as they are discovered

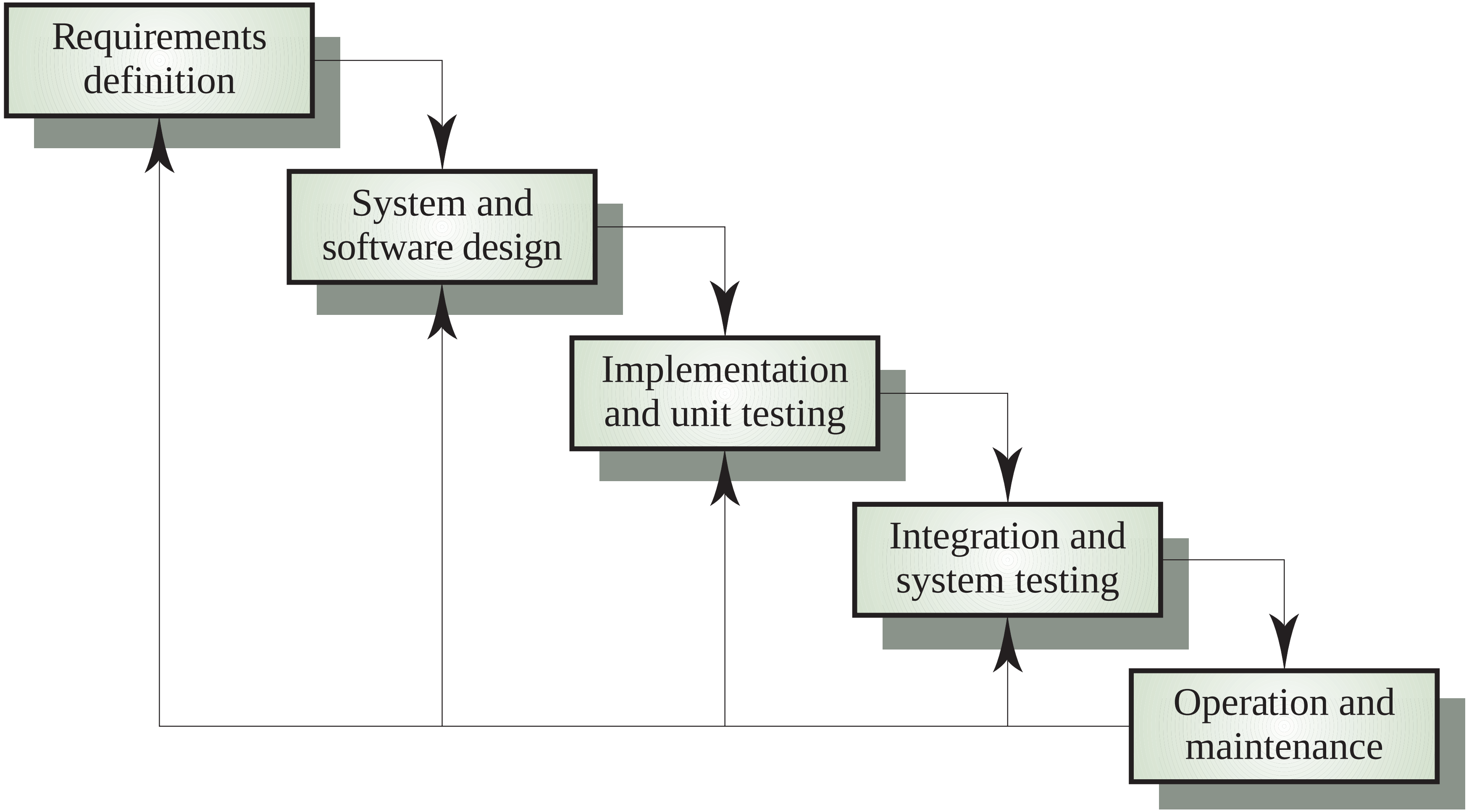
Software process models

- Normally, specifications are incomplete (or inconsistent)
- Very blurred distinction between specification, design and manufacture
- No physical realisation of the system for testing
- Software does not wear out - maintenance does not mean component replacement (it means fixing!)

Generic software process models

- The waterfall model
    - Separate and distinct phases of specification and development
- Evolutionary development
    - Specification and development are interleaved
- Formal transformation
    - A mathematical system model is formally transformed to an implementation
- Reuse-based development
    - The system is assembled from existing components

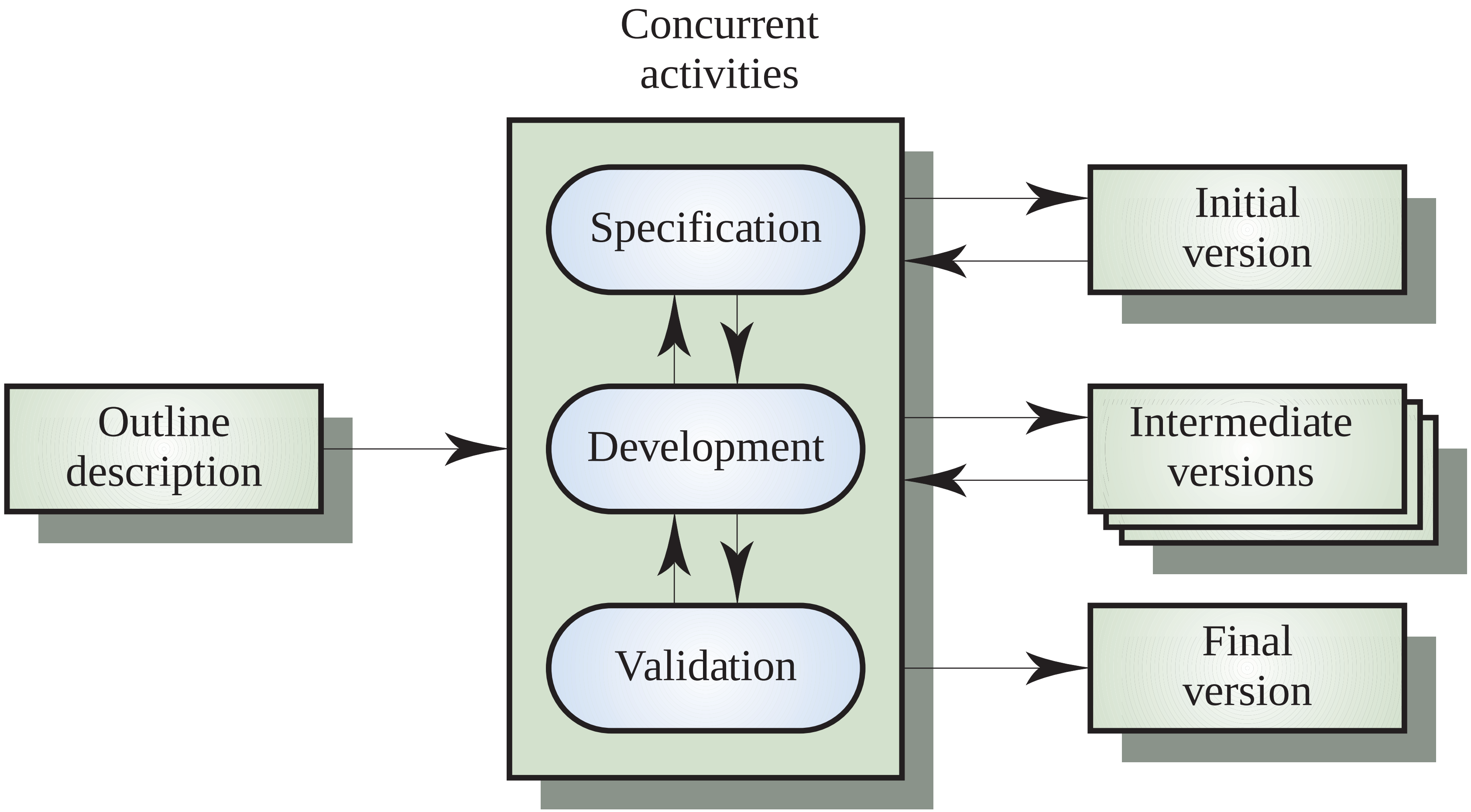
Waterfall model


Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Waterfall model phases

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The drawback of the waterfall model is the difficulty of accommodating change after the process is underway

Evolutionary development



Evolutionary development

- Exploratory prototyping
  - ❑ Objective is to work with customers and to evolve a final system from an initial outline specification. Should start with well-understood requirements
- Throw-away prototyping
  - ❑ Objective is to understand the system requirements. Should start with poorly understood requirements

Evolutionary development

- Problems
  - ❑ Lack of process visibility
  - ❑ Systems are often poorly structured
  - ❑ Special skills (e.g. in languages for rapid prototyping) may be required
- Applicability
  - ❑ For small or medium-size interactive systems
  - ❑ For parts of large systems (e.g. the user interface)
  - ❑ For short-lifetime systems

Risk management

- Perhaps the principal task of a manager is to minimise risk
- The 'risk' inherent in an activity is a measure of the uncertainty of the outcome of that activity
- High-risk activities cause schedule and cost overruns
- Risk is related to the amount and quality of available information. The less information, the higher the risk

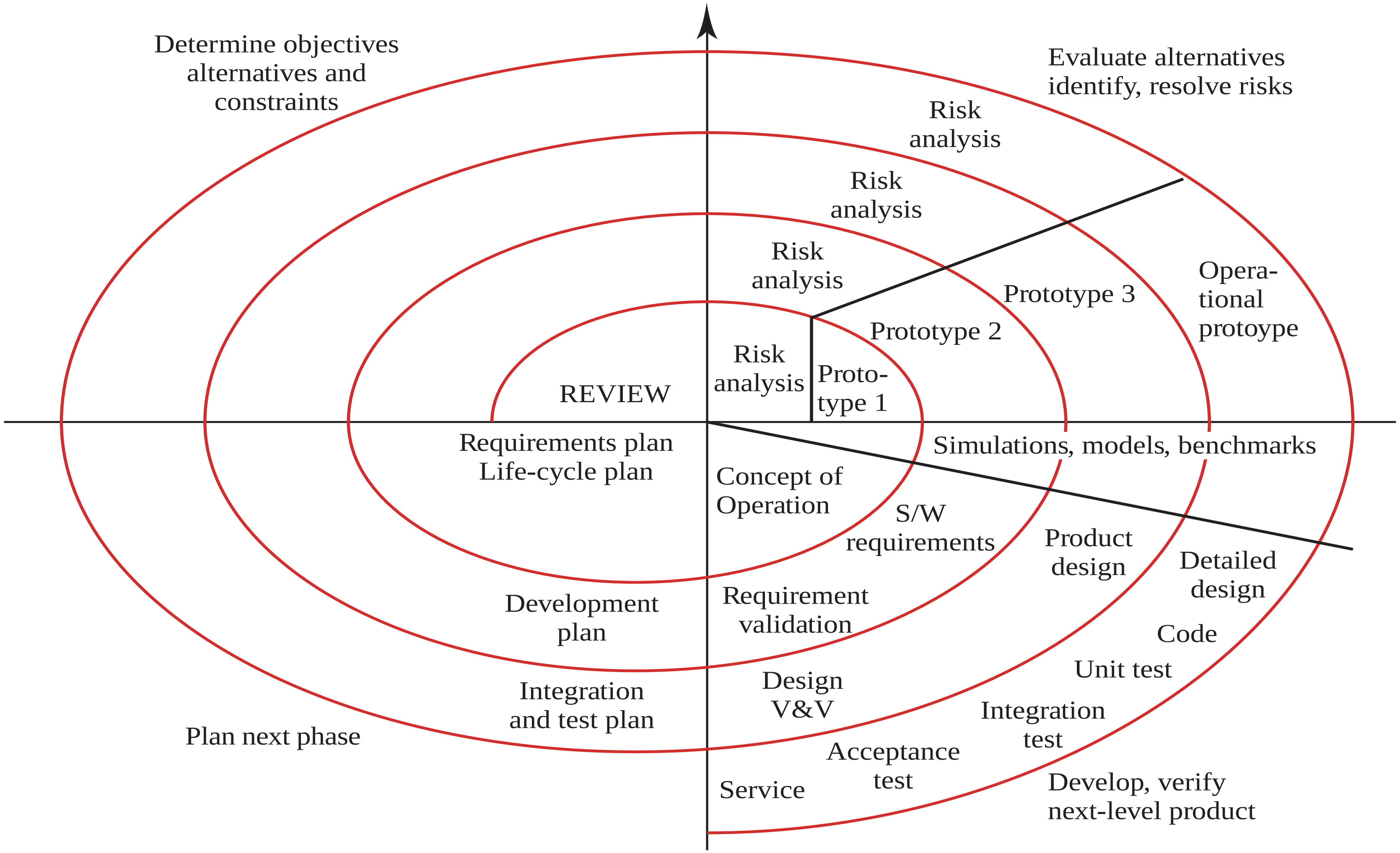
Process model risk problems

- Waterfall
  - High risk for new systems because of specification and design problems
  - Low risk for well-understood developments using familiar technology
- Prototyping (Evolutionary)
  - Low risk for new applications because specification and program stay in step
  - High risk because of lack of process visibility
- Transformational
  - High risk because of need for advanced technology and staff skills

Hybrid process models

- Large systems are usually made up of several sub-systems
- The same process model need not be used for all subsystems
- Prototyping for high-risk specifications
- Waterfall model for well-understood developments

Spiral model of the software process



Phases of the spiral model

- Objective setting
  - Specific objectives for the project phase are identified
- Risk assessment and reduction
  - Key risks are identified, analysed and information is sought to reduce these risks
- Development and validation
  - An appropriate model is chosen for the next phase of development.
- Planning
  - The project is reviewed and plans drawn up for the next round of the spiral

Template for a spiral round

- Objectives
- Constraints
- Alternatives
- Risks
- Risk resolution
- Results
- Plans
- Commitment

Quality improvement

- Objectives
  - ❑ Significantly improve software quality
- Constraints
  - ❑ Within a three-year timescale
Without large-scale capital investment
Without radical change to company standards
- Alternatives
  - ❑ Reuse existing certified software
Introduce formal specification and verification
Invest in testing and validation tools

- **Risks**
  - No cost effective quality improvement possible
    Quality improvements may increase costs excessively
    New methods might cause existing staff to leave
- **Risk resolution**
  - Literature survey
    Pilot project
    Survey of potential reusable components
    Assessment of available tool support
    Staff training and motivation seminars

- Results
  - ❑ Experience of formal methods is limited - very hard to quantify improvements
    Limited tool support available for company standard development system.
    Reusable components available but little reuse tool support
- Plans
  - ❑ Explore reuse option in more detail
    Develop prototype reuse support tools
    Explore component certification scheme
- Commitment
  - ❑ Fund further 18-month study phase

Catalogue Spiral

- Objectives
  - Procure software component catalogue
- Constraints
  - Within a year
    Must support existing component types
    Total cost less than $100, 000
- Alternatives
  - Buy existing information retrieval software
    Buy database and develop catalogue using database
    Develop special purpose catalogue

- Risks
  - ❑ May be impossible to procure within constraints
    Catalogue functionality may be inappropriate
- Risk resolution
  - ❑ Develop prototype catalogue (using existing 4GL and an existing DBMS) to clarify requirements
    Commission consultants report on existing information retrieval system capabilities.
    Relax time constraint

- Results
  - ❑ Information retrieval systems are inflexible. Identified requirements cannot be met.
    Prototype using DBMS may be enhanced to complete system
    Special purpose catalogue development is not cost-effective
- Plans
  - ❑ Develop catalogue using existing DBMS by enhancing prototype and improving user interface
- Commitment
  - ❑ Fund further 12 month development

Spiral model flexibility

- Well-understood systems (low technical risk) - Waterfall model. Risk analysis phase is relatively cheap
- Stable requirements and formal specification. Safety criticality - Formal transformation model
- High UI risk, incomplete specification - prototyping model
- Hybrid models accommodated for different parts of the project

Spiral model advantages

- Focuses attention on reuse options
- Focuses attention on early error elimination
- Puts quality objectives up front
- Integrates development and maintenance
- Provides a framework for hardware/software development

Spiral model problems

- Contractual development often specifies process model and deliverables in advance
- Requires risk assessment expertise
- Needs refinement for general use

Process visibility

- Software systems are intangible so managers need documents to assess progress
- However, this may cause problems
    - ❑ Timing of progress deliverables may not match the time needed to complete an activity
    - ❑ The need to produce documents constrains process iteration
    - ❑ The time taken to review and approve documents is significant
- Waterfall model is still the most widely used deliverable-based model

Waterfall model documents

Activity	**Output documents**
Requirements analysis	Feasibility study, Outline requirements
Requirements definition	Requirements document
System specification	Functional specification, Acceptance test plan Draft user manual
Architectural design	Architectural specification, System test plan
Interface design	Interface specification, Integration test plan
Detailed design	Design specification, Unit test plan
Coding	Program code
Unit testing	Unit test report
Module testing	Module test report
Integration testing	Integration test report, Final user manual
System testing	System test report
Acceptance testing	Final system plus documentation

Process model visibility

Process model	**Process visibility**
Waterfall model	Good visibility, each activity produces some deliverable
Evolutionary development	Poor visibility, uneconomic to produce documents during rapid iteration
Formal transformations	Good visibility, documents must be produced from each phase for the process to continue
Reuse-oriented development	Moderate visibility, it may be artificial to produce documents describing reuse and reusable components.
Spiral model	Good visibility, each segment and each ring of the spiral should produce some document.

Key points

- Software engineering is concerned with the theories, methods and tools for developing, managing and evolving software products
- Software products consist of programs and documentation. Product attributes are maintainability, dependability, efficiency and usability
- The software process consists of those activities involved in software development

Key points

- The waterfall model considers each process activity as a discrete phase
- Evolutionary development considers process activities as concurrent
- The spiral process model is risk-driven
- Process visibility involves the creation of deliverables from activities
- Software engineers have ethical, social and professional responsibilities

Warning! Software Development vs. Software Engineering

- Software textbooks tend to emphasize the management aspects and process aspects of software development
- While software engineering is certainly important, it is not everything.
- The following points, taken from a recent article in IEEE Software[2], make the argument (see J. A. Whittaker and S. Atkin, “Software Engineering Is Not Enough”, IEEE Software, July/August 2002, pp. 108-115.)

Warning! Software Development vs. Software Engineering

- Software Development Is More Than Methodology (or Process, or Estimation, or Project Management)
  - ❑ ‘Software development is a fundamentally technical problem for which management solutions can be only partially effective.’
  - ❑ Coding is immensely difficult without a good design but still very difficult with one
  - ❑ Maintaining code is next to impossible without good documentation and formidable with it.’

Warning! Software Development vs. Software Engineering

- Programming is Hard
  - ‘Programming remains monstrously complicated for the vast majority of applications’
  - ‘The only programs that are simple and clear are the ones you write yourself’
  - (Why?)

Warning! Software Development vs. Software Engineering

- Documentation is Essential
  - ❑ ‘There is rarely such a thing as too much documentation ...
  - ❑ Document Control Blocks and Data Structures
  - ❑ ‘Documentation – often exceeding the source code in size – is a requirement, not an option.’

Warning! Software Development vs. Software Engineering

- You Must Validate Data
  - ❑ Validate input
  - ❑ Validate parameters
  - ❑ ‘Constraints on data and computation usually take the form of wrappers – access routines (or methods) that prevent bad data from being stored or used and ensure that all programs modify data through a single, common interface’

Warning! Software Development vs. Software Engineering

- Failure is Inevitable – You Need To Handle It
  - ❑ Constraints prevent failure.
  - ❑ Exceptions let the failure occur and then trap it (and handle it with a special routine called an exception handler).
  - ❑ ‘Failure recovery is often difficult.’

Warning! Software Development vs. Software Engineering

- Before you can even begin, you must be an expert in both the problem domain and the solution domain

  - ❑ *Problem-domain expertise*
    (understanding and modelling the problem)

  - ❑ *Solution-domain expertise*
    (editors, compilers, linkers, and debuggers, ... make-utilities, runtime libraries, development environments, version-control managers, and ... the operating system)

  - ❑ **‘Developers must be masters of their programming language and their OS; methodology alone is useless.’**

Software Engineering 2

Software Process and Project Metrics

Software Process and Project Metrics

Measurement is fundamental to any engineering discipline (Why?)

- *Software metrics* is a term used to describe a range of measurements for software.

- Measurement is applied to:

  1. The software ***process*** to improve it
  2. A software ***project*** to assist in
     - Estimation (of resources)
     - Quality control
     - Productivity assessment
     - Project control
  3. A software ***product*** to assess its quality

Software Process and Project Metrics

Terminology: Measures, Metrics, and Indicators

- ***Measure***: something that provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some attribute of a product or process. For example, the number of errors uncovered in a single review is a measure.

- ***Metric***: a quantitative measure of the degree to which a system, component, or process possesses a given attribute. For example, the number of errors uncovered per review is a metric. Metrics relate measures.

- ***Indicator***: a metric or combination of metrics that provides some insight into the software process, project, or product.

Software Process and Project Metrics

Metrics in the Process and Project Domains

- The effectiveness of a software engineering process is measured indirectly:
  - ❑ We derive a set of metrics based on the outcomes that result from the process
  - ❑ We derive process metrics by measuring characteristics of specific software engineering tasks

Software Process and Project Metrics

- **Examples of outcomes include:**
  - ❑ **Measures of errors uncovered before the release of the software;**
  - ❑ **Defects delivered to and reported by end users;**
  - ❑ **Human effort expended**
  - ❑ **Calendar time expended**
  - ❑ **Conformance to the planned schedule**

- **Examples of characteristics include:**
  - ❑ **Time spent on umbrella activities (*e.g.* software quality assurance, configuration management, measurement ..)**

Software Process and Project Metrics

- **Software process metrics should be used carefully:**
  - ❑ **Use common sense and organizational sensitivity when interpreting them**
  - ❑ **Provide regular feedback to individuals and teams who have worked to collect the measures and metrics**
  - ❑ **Don’t use metrics to appraise (judge) individuals**
  - ❑ **Never use metrics to threaten individuals or teams**
  - ❑ **Metric data that flag problem areas should not be considered ‘negative’ – they are simply an indicator of potential process improvement**

Software Process and Project Metrics

Software Measurement

There are two types of measurements:

1. **Direct measures**

   - ❑ **direct process measures include cost and effort**
   - ❑ **direct product measures include lines of code (LOC), execution speed, memory size, defects per unit time**

2. **Indirect measures**

   - ❑ **Indirect product measures include functionality, quality, complexity, efficiency, reliability, maintainability**

Software Process and Project Metrics

Size-oriented Metrics

- ❑ Errors per KLOC
- ❑ Defects per KLOC
- ❑ $ per KLOC
- ❑ pages of documentation per KLOC
- ❑ errors per person-month
- ❑ LOC per person-month
- ❑ $ per page of documentation

Size-oriented metrics are not universally accepted as the best way to measure the process of software development.

For example LOC are language dependent and can penalize shorter well-designed programming

Software Process and Project Metrics

Function-Oriented Metrics

- ❑ Since functionality cannot be measured directly, it must be derived indirectly using other direct measures

- ❑ The *function point* metric most common function-oriented metric

- ❑ Function points are computed as follows

Software Process and Project Metrics

Step 1. Complete the following table.

				Weighting Factor			
Measurement Parameter	Count		Simple	Average	Complex		
Number of user inputs		×	3	4	6	=	
Number of user outputs		×	4	5	7	=	
Number of user inquiries		×	3	4	6	=	
Number of files		×	7	10	15	=	
Number of external interfaces		×	5	7	10	=	

User input — a distinct application-oriented data to the software

User output — a distinct application-oriented output such as error messages, menu, etc

User inquiry — an on-line input that results in the generation of some immediate software response in the form on an on-line output

File — a logically-distinct repository of information

External interface — a machine readable interface such as port, disk, tape, CD

Software Process and Project Metrics

Step 2. Compute the complexity adjustment values by answering the following questions and rating each factor (F_i) on a scale of 0 to 5:

F_1 Does the system require reliable backup and recovery?

F_2 Are data communications required?

F_3 Are there distributed processing functions?

F_4 Is performance critical?

F_5 Will the system run in an existing, heavily utilized operational environment?

F_6 Does the system require on-line data entry?

F_7 Does the on-line data entry require the input transaction to be built over multiple operations?

F_8 Are the master files updated online?

F_9 Are the inputs, outputs, files, or inquiries complex?

F_{10} Is the internal processing complex?

F_{11} Is the code designed to be reusable?

F_{12} Are conversion and installation included in the design?

F_{13} Is the system designed for multiple installations in different organizations?

F_{14} Is the application designed to facilitate change and ease of use by the user?

Note:

0	No influence
1	Incidental
2	Moderate
3	Average
4	Significant
5	Essential

Software Process and Project Metrics

Step 3. Compute the function point value from the following equation:

$$FP = count\_total \times \left(0.65 + 0.01 \times \sum F_i\right)$$

Once function points have been calculated, they are used to normalize measures of software productivity, quality, and other attributes:

- ❑ Errors per FP
- ❑ Defects per FP
- ❑ $ per FP
- ❑ page of documentation per FP
- ❑ FP per person-month

Software Process and Project Metrics

Extended Function Point Metrics

- The function point metric was originally designed to be applied to business information systems applications

- It is not quite as useful for engineering applications because they emphasize function and control rather than data transactions

- Extended function point metrics overcome this by including a new software characteristic called algorithms (a bounded computation problem that is included within a specific computer program)

- More sophisticated extended function point metrics, such as 3-D function points, have also been developed

Software Process and Project Metrics

3-D Function Point Metric

- Key idea: extend the standard FP to include
  - ❑ the complexity of the data processing *and*
  - ❑ the functional (algorithmic) complexity.

- The software system is characterized in three dimensions:
  - ❑ Data dimension
  - ❑ Functional dimension
  - ❑ Control dimension

Software Process and Project Metrics

3-D Function Point Metric

- The **data** dimension
  - Evaluated in much the same way as the normal FP
  - Count the number of
    - internal data-structures
    - external data sources
    - user inputs
    - user outputs
    - user inquiries
  - each being assigned a complexity attribute of low, average, or high.

Software Process and Project Metrics

3-D Function Point Metric

- The **functional** dimension
  - ❑ Evaluated by identifying all distinct information transformations in the system (or module).
  - ❑ Transformations imply a change in the semantic content of the data, not simply a movement of data from one place to another
  - ❑ Specifically, a transformation is
    - a series of processing steps
    - that are governed by a set of semantic constraints
    - (Pressman calls them semantic statements – they could equally be called semantic predicates)

Software Process and Project Metrics

3-D Function Point Metric

- The **functional** dimension
  - For example, consider a search algorithm:
    - a transformation taking a list as input and producing an position locator as output
    - It has several processing steps that probe elements in the list and then move to a different location
    - the semantic constraint is that the element being probed should be identical to the key being sought
  - Transformations may have many processing steps and many semantic constraints

Software Process and Project Metrics

3-D Function Point Metric

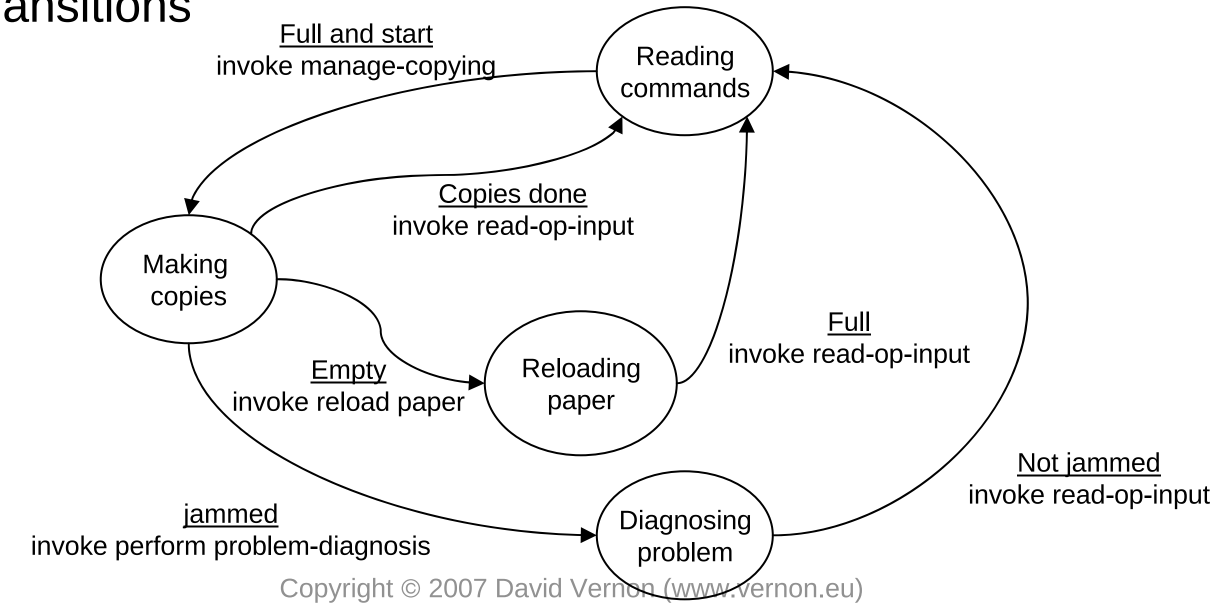
- The **functional** dimension
  - ❑ Depending on the number of steps and constraints we characterize the complexity of *each* transformation as *low*, *average*, or *high*, according to the following table

	Semantic Constraints (Statements / Predicates)			
Processing Steps		**1-5**	**6-10**	**>10**
	1-10	**Low**	**Low**	**Average**
	11-20	**Low**	**Average**	**High**
	>20	**Average**	**High**	**High**

Software Process and Project Metrics

3-D Function Point Metric

- The **control** dimension
  - ❑ Measured by counting the number of transitions between states (typically using a state transition diagram for the system or module being analyzed)
  - ❑ For example, the following state transition diagram has 6 state transitions



Software Process and Project Metrics

3-D Function Point Metric

- Finally, you compute the 3D FP index by completing the following table and summing the sub-totals):

			Complexity Weighting				
Measurement Element	**Count**		**Simple**	**Average**	**High**		**Sub-Total**
Internal data structures		×	**7**	**10**	**15**	**=**	
External data		×	**5**	**7**	**10**	**=**	
Number of user inputs		×	**3**	**4**	**6**	**=**	
Number of user outputs		×	**4**	**5**	**7**	**=**	
Number of user inquiries		×	**3**	**4**	**6**	**=**	
Transformations		×	**7**	**10**	**15**	**=**	
Transitions		×	**1**	**1**	**1**	**=**	

Software Process and Project Metrics

Reconciling Different Metrics

- The relationship between lines of code and function points depends on the programming language used to implement the software
- The following table provides a rough indication of the number of lines of code required to build one function point in various languages

Programming Language	LOC/FP (average)
Assembly language	320
C	128
Cobol	105
Fortran	105
Pascal	90
Ada	70
OO languages	30
4GLs	20
Code generators	15
Spreadsheets	6
Graphical languages (icons)	4

Software Process and Project Metrics

Metrics for Software Quality

- The quality of a system or product depends on
  - ❑ The requirements that describe the problem;
  - ❑ The analysis and design that models the solution;
  - ❑ The code that leads to an executable program;
  - ❑ The tests that exercise the software to uncover errors.
- A good software engineering uses measurement to assess the quality of all four components
- To accomplish this real-time quality assessment, the engineer must use technical measures to evaluate quality in an objective (rather than a subjective) way
- Chapters 18 and 23 of Pressman cover these technical measures where metrics are presented for the analysis model, specification quality, design model, source code, testing, and maintenance, with variants for object-oriented systems

Software Process and Project Metrics

Metrics for Software Quality

- Project managers must also assess quality as the project progresses
  - ❑ Typically he will collect and assimilate into project-level results the individual measures and metric generated by software engineers
  - ❑ The main focus at project level is on errors and defects

Software Process and Project Metrics

Metrics for Software Quality

DRE (Defect Removal Efficiency) is an error-based quality metric which can be used for both process and project quality assurance

$$DRE = E / (E + D)$$

where

E = the number of errors found before delivery of the software to the end user

D = the number of defects found after delivery

The ideal value of DRE is 1

Software Process and Project Metrics

Metrics for Software Quality

- The DRE metric can also be used to assess quality within a project to assess a team’s ability to find errors before they are passed to the next framework activity.
- In this case, we redefine it as follows

$$DRE_i = E_i / (E_i + E_{i+1})$$

where

E_i = the number of errors found during software engineering activity i

E_{i+1} = the number of errors found during software engineering activity $i+1$ that are traceable to errors that were not discovered in software engineering activity i

Software Project Planning

- **Software Project Planning**
  - ❑ Resources
  - ❑ Estimation
  - ❑ Decomposition
  - ❑ The COCOMO Estimation Model

Software Project Planning

- **General Issues**

  - The software project management process begins with a set of activities collectively called *project planning*

  - The first of these activities is *estimation*
    - Best-case and worst-case scenarios should be considered so that the project can be bounded

  - The software project planner must estimate several things before a project begins:

    - How long it will take
    - How much effort will be required
    - How many people will be involved
    - The hardware resources required
    - The software resource required
    - The risk involved

Software Project Planning

- **General Issues**

  - A good project manager is someone with
    - The ability to know what will go wrong before it actually does
    - The courage to estimate when the future is unclear and uncertain
  - Leadership **(**asd@asd.com in comp.software-eng)
    - **Leaders:**
      - lead by example
      - don't ask anything of anyone they wouldn't do themselves
      - are called on to make difficult and unpopular decisions
      - keep the team focused
      - reward/support their team in whatever they do
      - keep/clear unnecessary issues out of the way of the team
  - Project leaders: TAKE the blame when things go wrong and SHARE the credit when things go right.

Software Project Planning

- **General Issues**

Issues that affect the uncertainty in project planning include:

❑ Project complexity (e.g. real-time signal processing vs. analysis of examination marks)

❑ Project size (as size increases, the interdependency between its component grows rapidly)

❑ Structural uncertainty (the completeness and finality of the requirements)

The software planner should demand completeness of function, performance, and interface definitions (all contained in the system specification)

Software Project Planning

- **Software Scope**

  - ❑ Function – the actions and information transformations performed by the system

  - ❑ Performance – processing and response time requirements

  - ❑ Constraints – limits placed on the software by, e.g., external hardware or memory restrictions

  - ❑ Interfaces – interactions with the user and other systems

  - ❑ Reliability – quantitative requirements for functional performance (mean time between failures, acceptable error rates, etc.)

Software Project Planning

- Scope can only be established by detailed discussions and reviews with the client

- To get the process started, some basic questions must the addressed:

  - ❑ Who is behind the request for this work?
  - ❑ Who will use the solution?
  - ❑ What will be the economic benefit of a successful solution?
  - ❑ Is there another source for the solution?
  - ❑ How would you (the client) characterize a ‘good’ output that would be generated by a successful solution?
  - ❑ What problems will this solution address?
  - ❑ Can you show me or describe to me the environment in which the solution will be used?
  - ❑ Are there any special performance issues or constraints that will affect the way the solution is approached?

Software Project Planning

- To get the process started, some basic questions must the addressed:
  - ❑ Are you the right person to answer these questions?
  - ❑ Are your answers official?
  - ❑ Are my questions relevant to the problem you have?
  - ❑ Am I asking too many questions?
  - ❑ Is there anyone else who can provide additional information?
  - ❑ Is there anything else that I should be asking you?
- Follow up with more problem-specific and project-specific meetings

Software Project Planning

- You need to identify completely the
  - Data
  - Function
  - Behaviour of the system
  - Constraints on system operation
  - System performance requirements
- To do this properly, you will have to engage in a process of decomposition, sub-dividing the overall system into functional sub-units

Software Project Planning

- Software Project Planning also requires the estimation of resources

  - Environmental Resources: Development (hardware and software) tools
  - Reusable software component (i.e. pre-existing reusable software)
  - People (human resources)

- Each resource is specified by four characteristics
  - Description of the resource
  - Statement of availability
  - Time at which the resource will be required
  - Duration for which the resource will be required

Software Project Planning

- ***Human Resources***

  Need to identify:

  - ❑ the skill sets (e.g. databases, CGI, java, OO)
  - ❑ the development effort (see later for techniques for estimating effort)

- ***Reusable Software Resources***

  - ❑ Off-the-shelf components – can be acquired in house or from a third party and which can be used directly in the project
  - ❑ Full-experience components – existing specification, designs, code, test data developed in house in previous projects but may require some modification; members of the project team have full experience in the area represented by these components
  - ❑ Partial-experience components – as above but require substantial modification; members of the project team have partial experience in these areas
  - ❑ New components – software that must be built by the team specifically for this project.

Software Project Planning

- ***Environmental Resources***

  - ❑ This refers to the software engineering environment

  - ❑ more than the development tools such as compilers, linkers, libraries, development computers

  - ❑ Also refers to the final target machine and any associated hardware (e.g. a navigation system for a ship will require access to the ship during final tests)

Software Project Planning

◆ Software Project Estimation

- ❑ Software is the most expensive part of a computer-based system
- ❑ A large cost over-run may mean the difference between profit and loss (and bankruptcy).
- ❑ Software cost and effort estimation is not an exact science but we can do much better than guesstimates by using systematic methods (to be described below).

Software Project Planning

- **Software Project Estimation**

  Generally, we use one or both of two approaches:

  ❑ Decomposition Techniques

  ❑ Empirical Estimation Techniques

Software Project Planning

Decomposition Techniques for Estimation

- The first step in estimation is to predict the size of the project
- Typically, this will be done using either LOC (the direct approach) or FP (the indirect approach)
- Then we use historical data (on similar types of projects) about the relationship between LOC or FP and time or effort to predict the estimate of time or effort for this project.

Software Project Planning

Decomposition Techniques for Estimation

- If we choose to use the LOC approach, then we will have to decompose the project quite considerably into as many component as possible and estimate the LOC for each component

- The size *s* is then the sum of the LOC of each component

- If we choose to use the FP approach, we don't have to decompose quite so much

Software Project Planning

Decomposition Techniques for Estimation

- In both cases, we make three estimates of size:
  - s_{opt} an optimistic estimate
  - s_m the most likely estimate
  - s_{pess} an optimistic estimate

- and combine them to get a *three-point* or *expected value EV*

$$EV = (s_{opt} + 4s_m + s_{pess})/6$$

- *EV* is the value that is used in the final estimate of effort or time

Software Project Planning

LOC Example

Consider the following project to develop a Computer Aided Design (CAD) application for mechanical components

- ❑ The software is to run on an engineering workstation
- ❑ It must interface with computer graphics peripherals (mouse, digitizer, colour display, laser printer)
- ❑ It is to accept 2-D and 3-D geometric data from an engineer
- ❑ The engineer will interact with and control the CAD system through a graphic user interface
- ❑ All geometric data and other supporting information will be maintained in a CAD database
- ❑ Design analysis modules will be develop to produce the required output which will be displayed on a number of graphics devices.

Software Project Planning

LOC Example

After some further requirements analysis and specification, the following major software functions are identified:

- ❑ User interface and control facilities (UICF)
- ❑ 2D geometric analysis (2DGA)
- ❑ 3D geometric analysis (3DGA)
- ❑ Database management (DBM)
- ❑ Computer graphics display functions (CGDF)
- ❑ Peripheral control (PC)
- ❑ Design analysis modules (DAM)

Software Project Planning

Function	Optimistic LOC	Most Likely LOC	Pessimistic LOC	Estimated LOC
UICF	2000	2300	3000	2367
2DGA	4000	5400	6500	5350
3DGA	5500	6600	9000	6817
DBM	3300	3500	6000	3883
CGDF	4250	4900	5500	4892
PC	2000	2150	2950	2258
DAM	6900	8400	10000	8417
Estimate LOC				**33983**

Software Project Planning

LOC Example

- Historical data indicates that the organizational average productivity for systems of this type is 630 LOC/per-month and the cost per LOC is $13

- Thus, the LOC estimate for this project is

  - 33983 / 620 = 55 person months
  - 33983 * 13 = $441700

Software Project Planning

FP Example

Measurement Parameter	Optimistic	Likely	Pess.	Est. Count	Weight	FP-count
Number of user inputs	20	24	30	24	4	97
number of user outputs	12	15	22	16	5	78
number of user inquiries	16	22	28	22	4	88
number of files	4	4	5	4	10	42
number of external interfaces	2	2	3	2	7	15
Count						**321**

Software Project Planning

FP Example

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
On-line data entry	4
Input transactions over multiple screens	5
Master file updated on-line	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5

Software Project Planning

FP Example

- Estimated number of FP is
- Count total * (0.65 + 0.01 * ΣF_i) = 372
- Historical data indicates that the organizational average productivity for systems of this type is 6.5 FP/per-month and the cost per FP is $1230.
- Thus, the FP estimate for this project is
  - 372 / 6.5 = 58 person months
  - 372 * 1230 = $45700

Software Project Planning

Empirical Estimation Models

The general form of empirical estimation models is:

$$E = A + B \times (ev)^{C}$$

Where *A*, *B*, and *C* are empirically derived constants. *E* is effort in person months and *ev* is the estimation variable (either LOC or FP).

Software Project Planning

Empirical Estimation Models

Here are some of the many model proposed in the software engineering literature:

$$E = 5.2 \times (KLOC)^{0.91}$$

$$E = 5.5 + 0.73 \times (KLOC)^{1.16}$$

$$E = 3.2 \times (KLOC)^{1.05}$$

$$E = 5.288 \times (KLOC)^{1.047}$$

$$E = -13.39 + 0.0545 \times FP$$

$$E = 585.7 + 15.12 \times FP$$

Software Project Planning

Empirical Estimation Models

- Note: for any given value of LOC or FP, they all give a different answer! Thus, all estimation models need to be calibrated for local needs

The COCOMO Model

- COCOMO stands for COnstructive COst Model
- There are three COCOMO models:

Model 1:
The Basic COCOMO model which computes software development effort and cost as a function of program size expressed in LOC.

Software Project Planning

Model 2:

The Intermediate COCOMO model which computes software development effort and cost as a function of program size and a set of cost drivers that include subjective assessments of product, hardware, personnel, and project attributes

Model 3:

The Advanced COCOMO model which incorporates all the characteristics of the intermediate version with an assessment of all the cost drivers' impact on each step (analysis, design, etc.) of the software engineering process

Software Project Planning

Basic COCOMO Model

$$E = a_b KLOC^{b_b}$$

$$D = c_b E^{d_b}$$

where:

E is the effort applied in person-months

D is the development time in chronological months

KLOC is the estimated number of delivered lines of code for the project (expressed in thousands).

a_b, b_b, c_b, d_b are given in the following table.

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Software Project Planning

Basic COCOMO Model

- Note that there are three different types of project:
  - ❑ *Organic*: relatively small simple software projects in which small teams with good application experience work to a set of less than rigid requirements
  - ❑ *Semi-detached*: an intermediate sized project in which teams with mixed experience work to meet a mix of rigid and less than rigid requirements
  - ❑ *Embedded*: a software project that must be developed within a set of tight hardware, software, and operational constraints

Software Project Planning

Basic COCOMO Model

- For example, the basic COCOMO model the CAD system would yield an estimate of effort as follows:

$$E = 2.4 \times KLOC^{1.05} = 2.4 \times 33.2^{1.05} = 95 \text{ Person-months}$$

$$D = 2.5 \times E^{0.35} = 2.5 \times 95^{0.35} = 12.3 \text{ months}$$

Software Project Planning

Intermediate COCOMO Model

$$E = a_i KLOC^{b_i} \times EAF$$

where:

- *E* is the effort applied in person-months
- *EAF* is an effort adjustment factor
- *KLOC* is the estimated number of delivered lines of code for the project (expressed in thousands).
- a_i, b_i, are given in the following table.

Software Project	$\boldsymbol{a_i}$	$\boldsymbol{b_i}$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Software Project Planning

Intermediate COCOMO Model

- The EAF typically has values in the range 0.9 to 1.4 and is computed on the basis of 15 cost driver attributes
- There are four categories of attributes:
  - ❑ Product attributes
  - ❑ Hardware attributes
  - ❑ Personnel attributes
  - ❑ Project attributes
- Each of the 15 attributes are rated on a scale of 1-6 and these are then use to compute an EAF based on published tables of values.

Software Project Planning

The Software Equation

- A multivariable model that assumes a specific distribution of effort over the life of a software development project
- Based on productivity data from over 4000 software engineering projects

Software Project Planning

The Software Equation

$$E = \frac{B \times LOC^3}{P^3 t^4}$$

E is the effort applied in person-months or person-years

t is the project duration in months or years

B is a special skills factor that increases slowly as the need for integration, testing, quality assurance, documentation, and management skills grows. *Typical values are:*

5-15 KLOC (small projects)	*B* = 0.16
> 70KLOC	*B* = 0.39

P is a productivity parameter

Software Project Planning

P is a productivity parameter that reflects:

- Overall process maturity and management practices
- Extent to which good software engineering practices are used
- Level of programming languages used
- State of the software environment
- Skills and experience of the software teams
- Complexity of the application

Typical values are:

Real-time embedded software	P=2000
Telecommunication and system software	P=10,000
Scientific software	P=12,000
Business systems applications	P=28,000

Note that the software equation has two independent variables: *LOC* and *t*

Project Scheduling and Tracking

- **Human Resources and Effort**
- **Task Definition**
- **Task Networks**
- **Schedules**

Project Scheduling and Tracking

- **Project Scheduling**

  Software development projects are very often delivered late. Why?

  - **Unrealistic deadline**
  - **Changing customer requirements (without rescheduling)**
  - **Underestimate of amount of effort required**
  - **Unforeseen risks**
  - **Unforeseen technical difficulties**
  - **Unforeseen human difficulties**
  - **Poor communication between project staff**
  - **Failure by project management to monitor and correct for delays**

**“How do software projects fall behind?
One day at a time”**

Fred Brooks, author of *The Mythical Man-Month*

Project Scheduling and Tracking

- **Aggressive (i.e. unrealistic) deadlines are a fact of life in the software business**

- **The project manager must:**

  Define all the project tasks
  Identify the ones that are on the critical path
  Allocate effort and responsibility
  Track progress

- **The project schedule will evolve: it will start as a macroscopic schedule and become progressively more detailed as the project proceeds**

Project Scheduling and Tracking

- **Critical Path**

  The chain of tasks that must be completed on schedule if the project as a whole is to be completed on schedule

  Consequently, the critical path determines the duration of the project

Project Scheduling and Tracking

Basic principles of software project scheduling

Compartmentalization
(decomposition and modularity)

Interdependency
(cf. ordering and concurrency)

Time Allocation
(effort, start date, end date)

Effort Validation
(globally and at any point in time)

Defined Responsibilities
(specific member of the team)

Defined Outcomes
(code, documentation, presentations, reports)

Defined Milestones
(checkpoints - group of tasks complete & outcomes reviewed)

Project Scheduling and Tracking

- **The Relationship between People and Effort**

  - ❑ **Common myth:**

    “if we fall behind schedule, we can always add more staff and catch up later in the project”

  - ❑ **Adding people late in the project often causes the schedule to slip further**

  - ❑ **New people have to learn, current people have to instruct them, and while they are doing so no work gets done.**

Fred Brooks, author of the *Mythical Man-Month* (1975) put it thus:

> **“Adding man-power to a late software project makes it later”.**

Project Scheduling and Tracking

- **The Relationship between People and Effort**

The relationship between the number of people working on a project and the productivity is not linear

Recall the software equation:

$$E = \frac{B \times LOC^3}{P^3 t^4}$$

Consider a small telecommunications project requiring an estimated 10k lines of code. In this case, P = 10000 and B = 0.16.

If the time available = 6 months, effort is approximately equal to 2.5 person-years or a team of at least 5 people

If the time available = 9 months, then the estimated effort is approximately equal to 0.5 person-years, i.e. a one-person team.

Project Scheduling and Tracking

◆ Distribution of Effort

- ❑ **Estimate total effort and then allocate to each component task.**

- ❑ **A common rule-of-thumb (i.e. guideline) is to distribute the effort 40%-20%-40%:**

  40% for specification, analysis, and design
  20% for implementation
  40% for testing

- ❑ **Often recommended that the specification, analysis, and design phase account for more than 40%**

Project Scheduling and Tracking

◆ Defining a Task Set for the Software Project

- **A task set is a collection of software engineering work tasks, milestones, and deliverables (outcomes) that must be accomplished to complete a project**
- **Task sets are designed to accommodate different type of project:**

1. ***Concept development*** **projects**
2. ***New application development*** **projects**
3. ***Application enhancement*** **projects**
4. ***Application maintenance*** **projects**
5. ***Reengineering*** **projects**

Project Scheduling and Tracking

- **Defining a Task Set for the Software Project**
  - ❑ **Decide on the degree of rigor with which the software development process will be applied.**
  - ❑ **There are four different levels of rigor:**

Casual

(umbrella tasks and documentation minimized)

Structured

(significant software quality assurance)

Strict

(all umbrella activities, robust documentation)

Quick Reaction

(focus on those task absolutely necessary to attain a good quality outcome)

Project Scheduling and Tracking

To select the appropriate task set for a project, complete the following table

Adaptation Criteria	Grade (1-5)	Weight	Entry Point Multiplier					X
			Concept	New Dev	Enhanc.	Maint.	Reeng.	
Size of the project		1.2	0	1	1	1	1	
Number of potential users		1.1	0	1	1	1	1	
Mission criticality		1.1	0	1	1	1	1	
Application longevity		0.9	0	1	1	0	0	
Stability of requirements		1.2	0	1	1	1	1	
Ease of customer-developer communication		0.9	1	1	1	1	1	
Maturity of application technology		0.9	1	1	0	0	1	
Performance constraints		0.8	0	1	1	0	1	
Embedded/non-embedded characteristics		1.2	1	1	1	0	1	
Project staffing		1.0	1	1	1	1	1	
Reengineering factors		1.2	0	0	0	0	1	
Average Score (TSS – Task Set Selector)								

Project Scheduling and Tracking

TSS Value	**Degree of Rigor**
TSS < 1.2	Casual
1.0 < TSS < 3.0	Structured
TSS > 2.4	Strict

Note the overlap in value ranges: this is not an exact science!

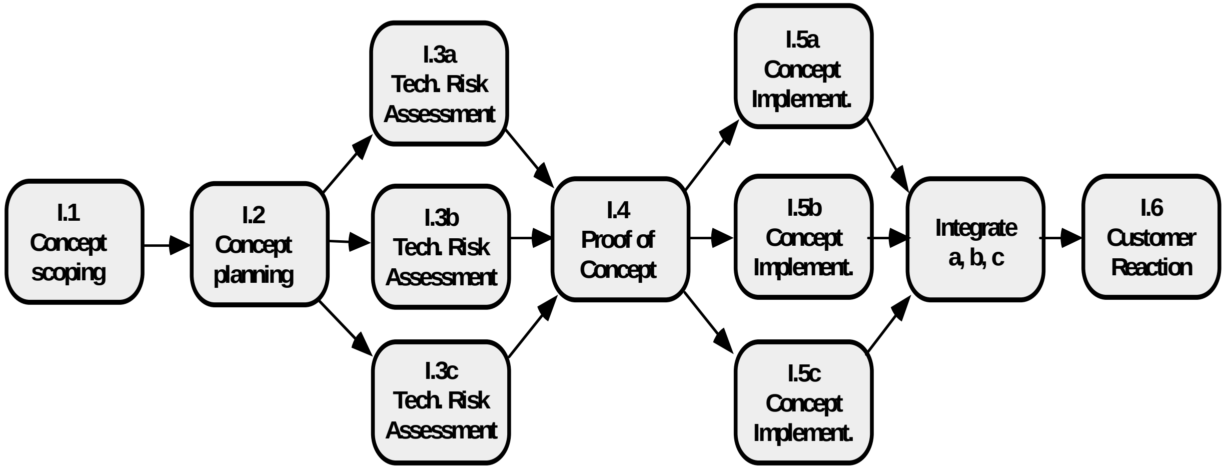
Use your judgment

Project Scheduling and Tracking

- Selecting Software Engineering Tasks
  - To develop a project schedule, a task set must be distributed on the project time line
  - The task set will vary according to the type of project and the degree of rigor
  - Depending on the project time, you will choose an appropriate process model (e.g. waterfall, evolutionary, spiral)
  - This process is forms the basis for a macroscopic schedule for a project
- Then proceed to refine each of the major tasks
  - breaking them into smaller composite tasks
  - aim to identify *clear*, *distinct*, *modular*, *independent* tasks with *well-defined inputs* (typically the result of other tasks) and *well-defined outputs* (typically feeding into other tasks).
- Much easier if you have a developed a rigorous and complete system specification

Project Scheduling and Tracking

- **Creating a Task Network**
  - This is a graphic representation of the task flow for a project and it makes explicit the dependencies, concurrency, and ordering of component tasks
  - It also allows the project manager to identify the critical path



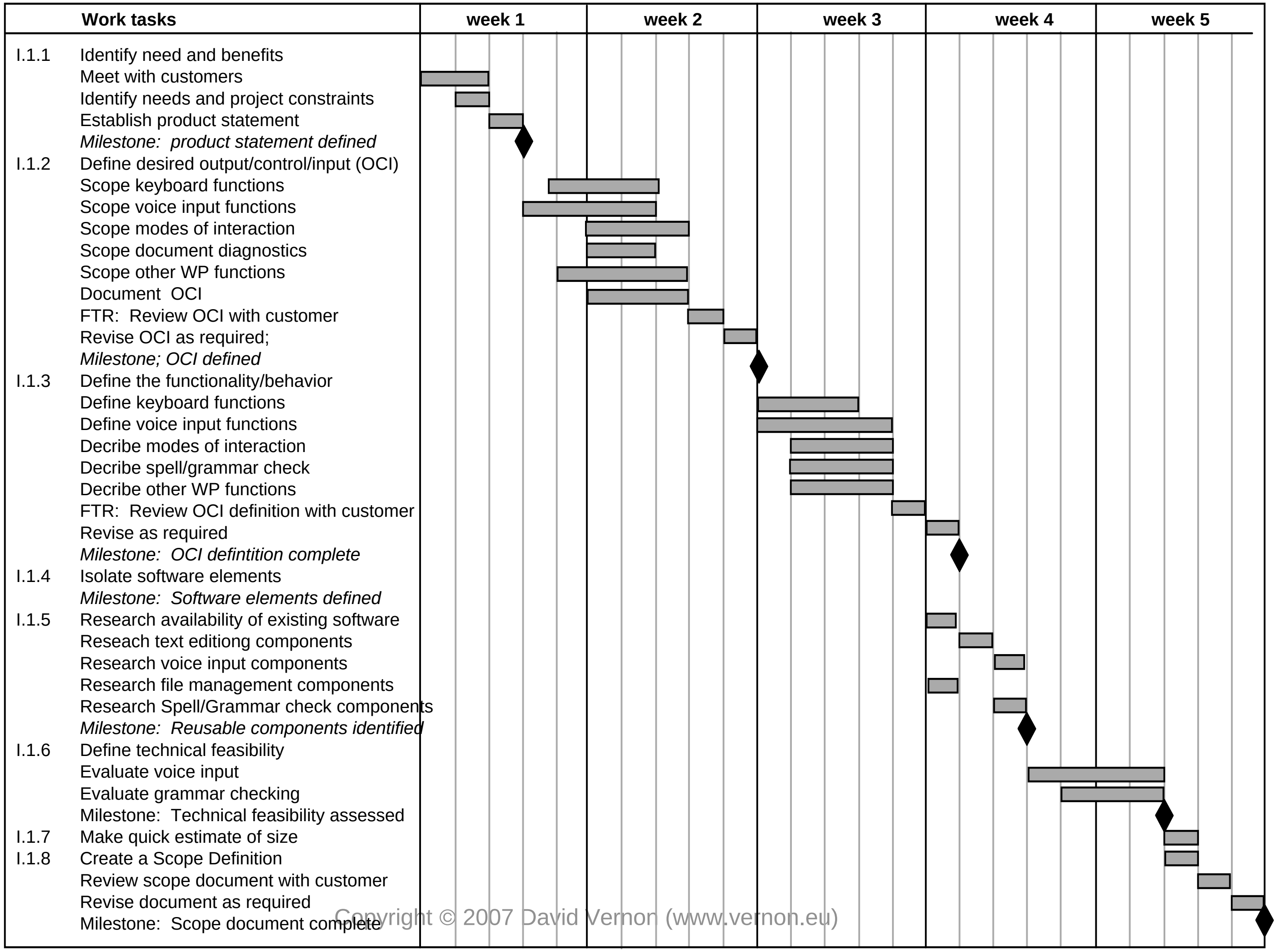
Three I.3 tasks are applied in parallel to 3 different concept functions

Three I.3 tasks are applied in parallel to 3 different concept functions

Project Scheduling and Tracking

- **Create the Project Schedule**
  - ❑ Remember always that it is a dynamic living entity that must be monitored, amended, revised, and validated
  - ❑ Typically, you will use a standard tool to effect this schedule for you.
  - ❑ Program evaluation and review technique (PERT)
    - Represents each task/subtask as a box containing its identity, duration, effort, start date, and end date (among other things)
    - It displays not only project timings but also the relationships among tasks (by arrows joining the tasks)
    - It identifies the tasks to be completed before others can begin and it identifies the *critical path*, *i.e.* the sequence of tasks with the longest completion time
  - ❑ Another standard tool is the timeline chart or GANTT chart
    - This represents tasks by horizontal bars and lines are used to show the dependencies.

Project Scheduling and Tracking



	Work tasks
I.1.1	Identify need and benefits
	Meet with customers
	Identify needs and project constraints
	Establish product statement
	Milestone: product statement defined
I.1.2	Define desired output/control/input (OCI)
	Scope keyboard functions
	Scope voice input functions
	Scope modes of interaction
	Scope document diagnostics
	Scope other WP functions
	Document OCI
	FTR: Review OCI with customer
	Revise OCI as required;
	Milestone; OCI defined
I.1.3	Define the functionality/behavior
	Define keyboard functions
	Define voice input functions
	Decribe modes of interaction
	Decribe spell/grammar check
	Decribe other WP functions
	FTR: Review OCI definition with customer
	Revise as required
	Milestone: OCI defintition complete
I.1.4	Isolate software elements
	Milestone: Software elements defined
I.1.5	Research availability of existing software
	Reseach text editiong components
	Research voice input components
	Research file management components
	Research Spell/Grammar check components
	Milestone: Reusable components identified
I.1.6	Define technical feasibility
	Evaluate voice input
	Evaluate grammar checking
	Milestone: Technical feasibility assessed
I.1.7	Make quick estimate of size
I.1.8	Create a Scope Definition
	Review scope document with customer
	Revise document as required
	Milestone: Scope document complete

Software Quality Assurance

- **Quality attributes**
- **Software reviews**
- **Statistical quality assurance**
- **Software quality standards**

Software Quality Assurance

- **Software Quality Assurance (SQA) is an activity that is applied throughout the software process**

- **SQA encompasses:**

  1. **A quality management approach**
  2. **Effective engineering technology (methods and tools)**
  3. **Formal technical reviews**
  4. **Multi-layered testing strategy**
  5. **Control of software documentation**
  6. **Procedures for assuring compliance with software development standards**
  7. **Measurement and reporting mechanisms**

Software Quality Assurance

- ***Quality control***
  - **the use of inspections, reviews, and tests to ensure that each work product (module, function, document) meets its requirements.**
  - **Control: we measure, assess, and correct through feedback**
  - **Later in the course, we will look in detail at measures and metrics specifically developed for distinct stages in the software process.**

Software Quality Assurance

- ***Quality assurance***
  - ❑ **consists of the auditing and reporting functions of management**
  - ❑ **The goal of quality assurance is to provide management with the data necessary to be informed about product quality**

Software Quality Assurance

- **A definition of software quality:**

  'Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software'

- **Some important points:**
  - ❑ **Software requirements are the foundation from which quality is measured**
  - ❑ **Specified standards define a set of development criteria that guide the manner in which software is engineered**
  - ❑ **There is always a set of implicit requirements (the non-functional attributes, such as dependability, etc.).**

Software Quality Assurance

- **Typically, there will be two main groups involved in SQA:**

1. ***The software engineering team.*** **quality is achieved by applying appropriate measures and metrics, conducting formal technical reviews, and performing well-planned software testing**

2. ***The SQA group*****. This group serves as the in-house customer representative; their goal is to assist the engineering team by performing independent quality audits.**

Software Quality Assurance

- **Software Reviews**
  - ❑ **Used to ‘filter out’ errors at various stages of the development process**
  - ❑ **Many types of reviews, including**
    - **informal discussions**
    - **customer presentations**
    - ***formal technical reviews FTR* (sometimes referred to as a walkthrough) SQA activity performed by software engineers**
  - ❑ **The objectives of the FTR are:**
    - **To uncover errors in function, logic, or implementation for any representation of the software**
    - **To verify that the software under review meets its requirements**
    - **To ensure that the software has been represented according to pre-defined standards**
    - **To achieve software that is developed in a uniform manner**
    - **To make projects more manageable.**

Software Quality Assurance

- **Every review meeting should adhere to the following guidelines:**
  - **Between three and five people should be involved**
    - **producer of the work**
    - **project leader**
    - **review leader**
    - **two or three reviewers**
    - **review recorder**
  - **Attendees should prepare in advance (but not spend more than 2 hours of work per person)**
  - **Duration of the review should be less than 2 hours**
- **A FTR should focus on one (small) part of the overall project.**

Software Quality Assurance

- **At the end of the review, all attendees must decide whether to**
  - ❑ **Accept the work projects without further modification;**
  - ❑ **Reject the work product due to severe errors (there will have to be another review once the errors are corrected).**
  - ❑ **Accept the work provisionally (minor errors have been encountered and no new review is required once they are corrected).**
- **The FTR team then signs off on this decision.**
- **In all cases, the review meeting must be fully documented**
  1. **Review Issues List (issues raised; action items)**
  2. **Review Summary (1 page: what was reviewed, who reviewed it, what were the findings and conclusions).**

Software Quality Assurance

- **Some guidelines for FTRs**
  - ❑ **Review the product, not the producer**
  - ❑ **Set an agenda and maintain it**
  - ❑ **Limit debate and rebuttal**
  - ❑ **Identify problem areas, but don’t attempt to solve every problem noted**
  - ❑ **Take written notes**
  - ❑ **Limit the number of participants and insist on advance preparation**
  - ❑ **Develop a checklist for each work product that is likely to be reviewed**
  - ❑ **Allocate resources and time schedule for FTRs**
  - ❑ **Conduct meaningful training for all reviewers**
  - ❑ **Review your early reviews**

Software Quality Assurance

- **Statistical Quality Assurance**
  - □ **The goal of statistical quality assurance is to try to assess quality in a quantitative manner**
  - □ **Information about software defects is collected and categorized**
  - □ **Each defect is traced to its underlying cause (e.g non-conformance to specification, design error, violation of standards, poor communication with customer)**
  - □ **Use the *Pareto principle* (80% of the defects can be traced to 20% of all possible causes): identify the critical 20%**
  - □ **Fix the problems in the 20%**
  - □ **This is a feedback process**

Software Quality Assurance

- ## Statistical Quality Assurance

  The following is a list of typical sources of defects

  - ❑ Incomplete or erroneous specification (IES)
  - ❑ Misinterpretation of customer communication (MCC)
  - ❑ Intentional deviation from specification (IDS)
  - ❑ Violation of programming standards (VPS)
  - ❑ Error in data representation (EDR)
  - ❑ Inconsistent module interface (IMI)
  - ❑ Error in design logic (EDL)
  - ❑ Incomplete or erroneous testing (IET)
  - ❑ Inaccurate or incomplete documentation (IID)
  - ❑ Error in programming language translation of design (PLT)
  - ❑ Ambiguous or inconsistent human-computer interface (HCI)
  - ❑ Miscellaneous (MIS)

Software Quality Assurance

- **Statistical Quality Assurance**
  - ❑ **Build the following table**
  - ❑ **Then focus on the sources giving the largest percentage of defects**

	Total		Serious		Moderate		Minor	
	Number	**%**	**Number**	**%**	**Number**	**%**	**Number**	**%**
IES								
MCC								
IDS								
VPS								
EDR								
IMI								
EDL								
IET								
IID								
PLT								
HCI								
MIS								

Software Quality Assurance

- **Statistical Quality Assurance**

Compute the error index

$$EI = \sum (i \times PI_i) / PS$$

i number of the current phase in the software process

PS size of the product (e.g. in kLOC)

PI_i phase index

$$PI_i = w_s (S_i / E_i) + w_m (M_i / E_i) + w_t (T_i / E_i)$$

E_i total number of errors uncovered during phase *i* of the software process

S_i number of serious errors

M_i number of moderate errors

T_i number of minor errors

W_s = 10, W_m = 3, W_t = 1, typically

Note that the error index weights errors that occur later in the software process more heavily

Software Quality Assurance

- **Statistical Quality Assurance**

  - ❑ **The key message here in using the Pareto Principal and SQA is:**

  - ❑ ***'Spend your time focusing on things that really matter, but first be sure you understand what really matters'.***

Software Quality Assurance

- **The Quality System**
  - ❑ **Software development organizations should have a *quality system* addressing the following tasks:**
    - **Auditing of projects to ensure that quality controls are being adhered to**
    - **Staff development of personnel in the SQA area**
    - **Negotiation of resources to allow staff in the SQA area to do their job properly**
    - **Providing input into the improvement of development activities**
    - **Development of standards, procedures, guidelines**
    - **Production of reports for top-level management**
    - **Review and improvement of the quality system**
  - ❑ **Details procedures for accomplishing these tasks will be set out in a *Quality Manual***
    - **Often, these procedures will follow international standards such as *ISO 9001***
  - ❑ **The quality manual is then used to create a *quality plan* for each distinct project**

Software Quality Assurance

- **The ISO 9001 Quality Standard**
  - ❑ **Adopted by more than 130 countries**
  - ❑ **Increasingly important as a way by which clients can judge (or be assured of) the competence of a software developer**
  - ❑ **Problems with ISO 9001: not industry-specific**

Software Quality Assurance

◆ The ISO 9001 Quality Standard

❑ **For the software industry, the relevant standards are**

- ***ISO 9001 Quality Systems – Model for Quality Assurance in Design, Development, Production, Installation, and Servicing***

  Describes the quality system used to support the development of a product which involves design

- ***ISO 9000-3 Guidelines for the Application of ISO 9001 to the Development, Supply, and Maintenance of Software.***

  Interprets ISO 9001 for the software developer

- ***ISO 9004-2. Quality Management and Quality System Elements – Part 2.***

  This document provides guidelines for the servicing of software facilities such as user support.

Software Quality Assurance

- **The ISO 9001 Quality Standard**

The requirements of the standard are grouped under 20 headings:

- **Management responsibility**
- **Quality system**
- **Contract review**
- **Design control**
- **Document control**
- **Purchasing**
- **Purchase supplied product**
- **Product identification and traceability**
- **Process control**
- **Inspection and testing**

Software Quality Assurance

- **The ISO 9001 Quality Standard**

The requirements of the standard are grouped under 20 headings:

- **Inspection, measuring and test equipment**
- **Inspection and test status**
- **Control of non-conforming product**
- **Corrective action**
- **Handling, storage, packaging and delivery**
- **Quality records**
- **Internal quality audits**
- **Training**
- **Servicing**
- **Statistical techniques**

Software Quality Assurance

- ***An excerpt from ISO 9001***

  4.11 Inspection, measuring and test equipment

  The supplier shall control, calibrate, and maintain inspection, measuring, and test equipment, whether owned by the supplier, on loan, or provided by the purchaser, to demonstrate the conformance of product to the specified requirements. Equipment shall be used in a manner which ensures that measurement uncertainty is known and is consistent with the required measurement capability

  - ❑ Very general statement & can be hard to interpret in a specific (software) domain
  - ❑ Very common for companies to invest in the services of an external consultant to help then achieve ISO 9001 certification (and other quality-oriented models such as CMM – the Software Engineering Institute Capability Maturity Model)
  - ❑ See www.q-labs.com for an example of the offerings of a typical international consulting firm

McCall’s Quality Factors & FURPS

- So far, the focus has been metrics that are applied at the process and project level
- We now we focus on measures that can be used to assess the quality of the software as it is being developed
- We will begin by introducing two software quality checklists (McCall and FURPS)

McCall’s Quality Factors & FURPS

- McCall’s software quality factors focus on three important aspects of a software product:
  - ❑ Its operational characteristics (*product operation*)
  - ❑ Its ability to undergo change (*product revision*)
  - ❑ Its adaptability to new environments (*product transition*)

McCall’s Quality Factors & FURPS

- The software quality factors are:
  - Product Operation
    - Correctness
    - Reliability
    - Efficiency
    - Integrity (security)
    - Usability
  - Product Revision
    - Maintainability
    - Flexibility
    - Testability
  - Product transition
    - Portability
    - Reusability
    - Interoperatibility

McCall’s Quality Factors & FURPS

- **It is very difficult to develop direct measures of these quality factors**

- **McCall proposes the *combination* of several metrics to provide these measures**

McCall’s Quality Factors & FURPS

- ***Auditability*** **(ease with which conformance with standards can be checked)**
- ***Accuracy*** **(precision of computations)**
- ***Communication commonality*** **(use of standard interfaces and protocols)**
- ***Completeness*** **(w.r.t. requirements)**
- ***Conciseness*** **(compact code)**
- ***Consistency*** **(uniformity of design and documentation)**
- ***Data commonality*** **(use of standard data structures and types)**
- ***Error tolerance*** **(cf graceful degradation)**

McCall’s Quality Factors & FURPS

- ***Execution efficiency*** **(run-time performance)**
- ***Expandability*** **(cf reuse)**
- ***Generality*** **(number of potential applications)**
- ***Hardware independence***
- ***Instrumentation*** **(degree to which the program monitors its own operation and identifies errors, cf autonomic computing)**
- ***Modularity*** **(functional independence of program components)**
- ***Operability*** **(ease of use)**
- ***Security***

McCall’s Quality Factors & FURPS

- ***Self-documentation*** **(usefulness/clarity of source code and internal documentation)**
- ***Simplicity*** **(ease of understanding of source code / algorithms)**
- ***Software system independence*** **(decoupling from operating system and libraries esp. DLLs)**
- ***Traceability*** **(ability to trace a design, representation, or code to initial requirements)**
- ***Training*** **(ease of learning by new users)**

McCall’s Quality Factors & FURPS

- Each metric is simply graded on a scale of 0 (low) to 10 (high)
- The measure for the software quality factors is then computed as a weighted sum of each metric:

$$F_q = \sum c_i \times m_i$$

where

F_q is the software quality factor

c_i are weighting factors

m_i are the metrics that affect the quality factor.

The weighting factors are determined by local organizational considerations.

McCall’s Quality Factors & FURPS

	Software Quality Factors										
Sofware Quality Metric	Correctness	Reliability	Efficiency	Integrity	Maintainability	Flexibility	Testability	Portability	Reusability	Interoperability	usability
Auditability				x			x				
Accuracy		x								x	
Communication commonality											
Completeness	x										
Conciseness			x		x	x					
Consistency	x	x			x	x					
Data commonality										x	
Error tolerance		x									
Execution efficiency			x								
Expandability						x					
Generality						x		x	x	x	
Hardware independence								x	x		
Instrumentation				x	x		x				
Modularity		x			x	x	x	x	x	x	
Operability			x								x
Security				x							
Self-documentation					x	x	x	x	x		
Simplicity		x			x	x	x				
Software system independence								x	x		
Traceability	x										
Training											x

McCall’s Quality Factors & FURPS

- **FURPS**
  - ❑ **Hewlett-Packard developed a set of software quality factors with the acronym FURPS**
  - ❑ **They define the following five major factors:**
    1. **Functionality**
    2. **Usability**
    3. **Reliability**
    4. **Performance**
    5. **Supportability**
  - ❑ **Each is assessed by evaluating a set of metrics, in much the same way as McCall’s Quality factors**

Technical Metrics

- Analysis
- Design
- Implementation
- Testing
- Maintenance

Technical Metrics

◆ *Metrics for the Analysis Model*

- ❑ These metrics examine the analysis model with the intent of predicting the size of the resultant system
- ❑ One of the most common size metrics is the Function Point (FP) metric
- ❑ The following metrics can be used to assess the quality of the requirements

Technical Metrics

Metrics for the Analysis Model

- ***Specificity metric (lack of ambiguity)***

$$Q_1 = n_{ui} / n_r$$

n_{ui} **is the number of requirements for which all reviewers had identical interpretations**

$n_r = n_f + n_{nf}$

n_r **is the total number of requirements**

n_f **is the number of functional requirements**

n_{nf} **is the number of non-functional requirements**

Technical Metrics

Metrics for the Analysis Model

- ***Completeness of functional requirements***

$$Q_2 = n_u / (n_i \times n_s)$$

n_u **is the number of unique functional requirements**

n_i **is the number of inputs**

n_s **is the number of states**

Technical Metrics

Metrics for the Design Model

- ***Structural Complexity of a module i***

$$S(i) = f_{out}^{2}(i)$$

$f_{out}(i)$ **is the fan-out of module *i***

(*i.e.* the number of modules that are called by module *i*)

Technical Metrics

Metrics for the Design Model

- ***Data Complexity of a module i***

$$D(i) = v(i) / [f_{out}(i) + 1]$$

$v(i)$ **is the number of input and output variables that are passed to and from module *i***

Technical Metrics

Metrics for the Design Model

- ***System Complexity of a module i***

$$C(i) = S(i) + D(i)$$

$v(i)$ **is the number of input and output variables that are passed to and from module *i***

Technical Metrics

Metrics for the Design Model

Morphology Metrics (based on functional decomposition graph)

- **size = $n + a$**

  n **is the number of nodes in the tree/graph (number of modules)**
  a **is the number of arcs (lines of control)**

- **depth = longest path from root to a leaf node**

- **width = maximum number of nodes at any one level**

- **arc-to-node ratio, $r = a/n$, provides a simple indication of the coupling of the architecture**

Technical Metrics

Metrics for the Design Model

Cohesion Metrics

(require knowledge of the detailed design of a module)

Recall:

cohesion is the degree to which a module performs a distinct task or function (without the need to interact with other modules in the program)

Technical Metrics

Metrics for the Design Model

Cohesion Metrics

First, we need to define five concepts:

Data slice: **a backward walk through a module looking for data values that affect the module location at which the walk began.**

Data tokens $D(i)$: **the variables in a module i**

Glue tokens $G(i)$: **the set of data tokens that lie on more than one data slice**

Technical Metrics

Metrics for the Design Model

Cohesion Metrics

Superglue tokens $S(i)$: **the data tokens that are common to every data slice in a module**

Stickiness: **the stickiness of a glue token is directly proportional to the number of data slices that it binds**

Technical Metrics

Metrics for the Design Model

Cohesion Metrics

Strong Functional Cohesion: $SFC(i) = S(i) \; / \; D(i)$

A module with no superglue tokens (*i.e.* no tokens that are common to *all* data slices) has zero strong functional cohesion – there are no data tokens that contribute to all outputs

As the ratio of *superglue* tokens to the total number of tokens in a module increases towards a maximum value of 1, the functional cohesiveness of the module also increases

Technical Metrics

Metrics for the Design Model

Cohesion Metrics

***Weak Functional Cohesion*:** $WFC(i) = G(i) / D(i)$

A module with no *glue* tokens (*i.e.* no tokens that are common to *more than one* data slices) has zero weak functional cohesion – there are no data tokens that contribute to more than one output

Cohesion metrics take on values in the range 0 to 1

Technical Metrics

Metrics for the Design Model

Coupling Metrics

(also require knowledge of the detailed design of a module)

Recall:

Coupling is the degree of interconnection among modules (and the interconnection of a module to global data)

Technical Metrics

Metrics for the Design Model

Coupling Metrics

There are (at least) four types of coupling:

1. **Data coupling – exchange of data between modules via parameters**
2. **Control flow coupling – exchange of control flags via parameters (this allows one module to influence the logic or flow of control of another module)**
3. **Global coupling – sharing of global data structures**
4. **Environmental coupling – the number of modules to which a given module is connected**

Technical Metrics

Metrics for the Design Model

Coupling Metrics

Let:

d_i = **number of input data parameters**

c_i = **number of input control parameters**

d_o = **number of output data parameters**

c_o = **number of output control parameters**

g_d = **number of global variables used as data**

g_c = **number of global variables used as control**

w = **number of modules called (fan-out)**

r = **number of modules calling the module under consideration (fan-in)**

Technical Metrics

Metrics for the Design Model

Coupling Metrics

Define a module coupling indicator m_c

$$m_c = k / M$$

$$M = d_i + d_o + a\,c_i + b\,c_o + g_d + c\,g_c + w + r$$

where $k = 1, a = b = c = 2$

These constants may be adjusted to suit local organizational conditions

Technical Metrics

Metrics for the Design Model

Coupling Metrics

The higher the value of m_c, the lower the overall coupling

> **For example, if a module has a single input data parameter and a single output data parameter, and if it access no global data, and is called by a single module, then**

$$m_c = 1 / (1 + 1 + 0 + 0 + 0 + 0 + 0 + 1) = 0.33$$

Is this the highest value of a module coupling indicator?

Technical Metrics

Metrics for the Design Model

Coupling Metrics

In order to have the coupling metric move upward as the degree of coupling increases, we can define a revised coupling metric C as:

$$C = 1 - m_c$$

In this case, the degree of coupling increases non-linearly between a minimum value of 0 to a maximum value that approaches 1.0

Technical Metrics

Metrics for the Design Model

Complexity Metrics

(require knowledge of the detailed design of a module)

- **The most widely used complexity metric is the *cyclomatic complexity*, sometimes referred to as the McCabe metric (after its developer, Thomas McCabe)**

- **Cyclomatic complexity defines the number of *independent paths* in the *basis set* of a program**

Technical Metrics

Metrics for the Design Model

Complexity Metrics

- **An independent path is any path through the program that introduces at least one new set of processing statements or a new condition**

- **A basis set is a set of independent paths which span (i.e. include) every statement in the program**

Technical Metrics

Metrics for the Design Model

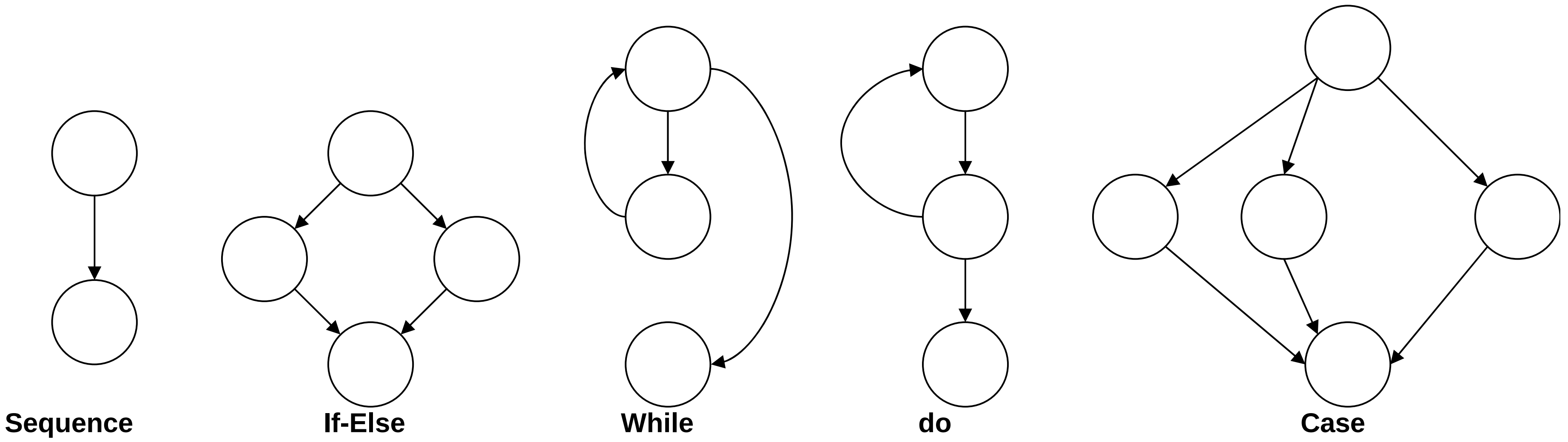
Complexity Metrics

- **If we represent a program as a flow graph**
  - ❑ statements represented as nodes
  - ❑ flow of control represented as edges
  - ❑ areas bounded by edges and nodes are regions
  - ❑ area outside the graph counting as one region
- **then an independent path must move along at least one edge that has not been traversed in any of the other independent paths**

Technical Metrics

Metrics for the Design Model

Complexity Metrics



Technical Metrics

Metrics for the Design Model

Complexity Metrics

We can define *cyclomatic complexity* in two (equivalent) ways:

(1) The cyclomatic complexity $V(G)$ of a flow graph G is defined as:

$$V(G) = E - N + 2$$

Where E is the number of flow graph edges and N is the number of flow graph nodes

Technical Metrics

Metrics for the Design Model

Complexity Metrics

We can define *cyclomatic complexity* in two (equivalent) ways:

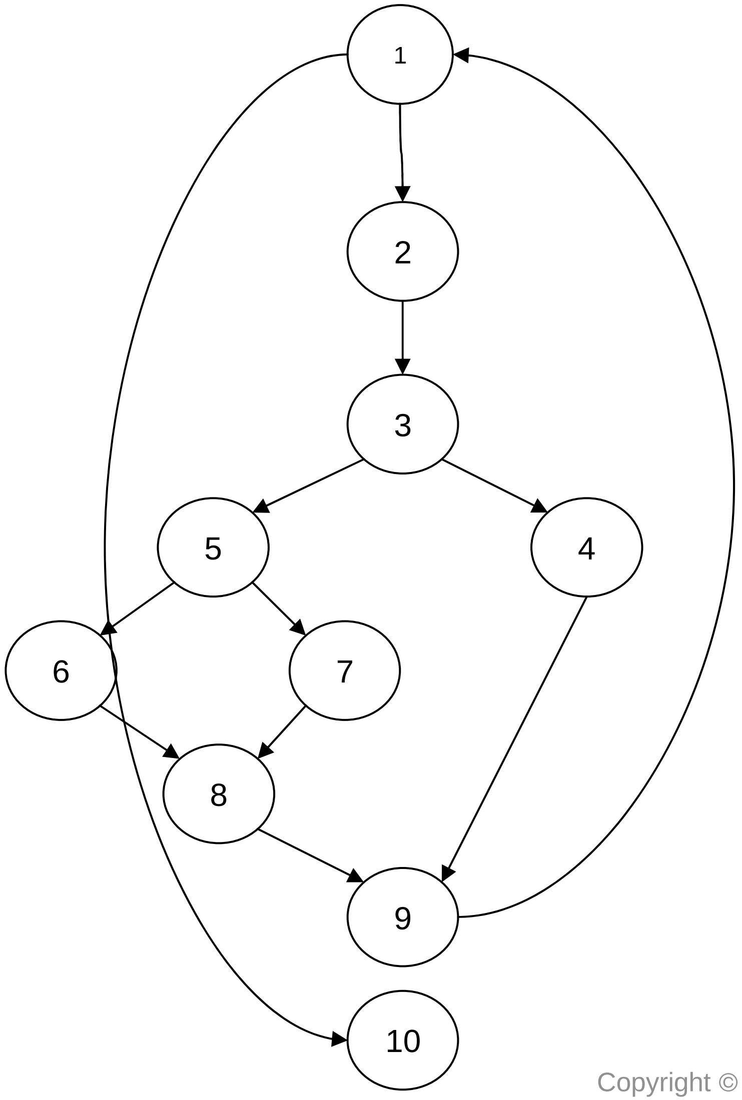
(2) The cyclomatic complexity $V(G)$ of a flow graph G is defined as:

$$V(G) = p + 1$$

Where P is the number of predicate nodes contained in the flow graph G

A *predicate node* is a node that contains a condition and is characterized by two or more edges emanating from it

Technical Metrics



```
// Sample Code
1    While records remain
2       read record
3       if record field 1 == 0
4           process record
             store in buffer
             increment counter
         else
5            if record field 2 == 0
6                reset counter
              else
7                process record
                  store in file
8            endif
9      endif
10   End while
```

Technical Metrics

Metrics for the Design Model

Interface Design Metrics

- The layout appropriateness metric, LA:
  - ❑ Assess the effectiveness of the interface
  - ❑ More specifically, the spatial organization of the GUI
  - ❑ Real effectiveness is best measured by user feedback

Technical Metrics

Metrics for the Design Model

Interface Design Metrics

- Define the cost C of performing a series of actions with a given layout:

$$C = \mathrm{S}\ (f(k)\ /\ m(k))$$

where

$f(k)$ is the frequency of a transition k

(between pair of widgets)

$m(k)$ is the cost of a transition k

(e.g distance a cursor must travel)

The summation is made for all k, i.e. all transitions

Technical Metrics

Metrics for the Design Model

Interface Design Metrics

- **The LA metric is then defined as follows:**

$$LA = 100 \text{ x } C_o / C_i$$

where

C_o **is the cost of the optimal layout**
C_i **is the cost of the current layout**

Technical Metrics

Metrics for the Design Model

Interface Design Metrics

- **To compute the optimal layout for a GUI**
  - ❑ **the screen is divided into a grid, with each square representing a possible position for a widget**
  - ❑ **If there are N positions in the grid and K widgets, then the number of possible layouts is** $[N! / K! \times (N-K)!] \times K!$
  - ❑ **This is the number of possible combinations of *K* widgets in *N* spaces times the number of permutations of widgets**
  - ❑ **If N is large, you will need to use a non-exhaustive approach to finding the optimal layout (e.g. tree search or dynamic programming)**
- ***LA* is used to assess different proposed GUI layouts and the sensitivity of a particular layout to changes in task descriptions (*i.e.* changes in the sequence or frequency of transitions)**

Technical Metrics

Metrics for Source Code

- Halstead's theory of software science defines a number of metrics for source code

- First, define the following measures

n_1 the number of distinct operators that appear in a given program (including the language constructs such as for, while, do, case, …).

n_2 the number of distinct operands that appear in a given program

N_1 the total number of occurrences of operators in a given program

N_2 be the total number of occurrences of operands in a given program

Technical Metrics

Metrics for Source Code

- **Program length metric**

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

- **Program volume metric**

$$V = N \log_2 (n_1 + n_2)$$

- **Halstead defines a volume ratio L as the ratio of volume of the most compact form of a given program to the volume of the actual program:**

$$L = 2 / n_1 \text{ x } n_2 / N_2$$

Technical Metrics

Metrics for Testing

- Most material in the SW Eng. literature focuses on the testing process, rather than characteristics (quality) of the tests themselves
- In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases
- FP metrics can be used as a predictor of overall testing effort, especially when combined with past experience in required testing
- The cyclomatic complexity metric defines the effort required for basis path (white box) testing
  - it also helps identify the functions or modules that should be targeted for particularly rigorous testing

Technical Metrics

Metrics for Maintenance

- **IEEE Standard 982.1-1988 suggests a software maturity index (*SMI*) that provides an indication of the stability of a software product based on changes that occur for each release of the product**

$$SMI = [\, M_T - (F_a + F_c + F_d)] / M_T$$

Technical Metrics

Metrics for Maintenance

$$SMI = [\, M_T - (F_a + F_c + F_d)] / M_T$$

where

M_T **= the number of modules in the current release**

F_a **= the number of modules in the current release that have been changed**

F_a **= the number of modules in the current release that have been added**

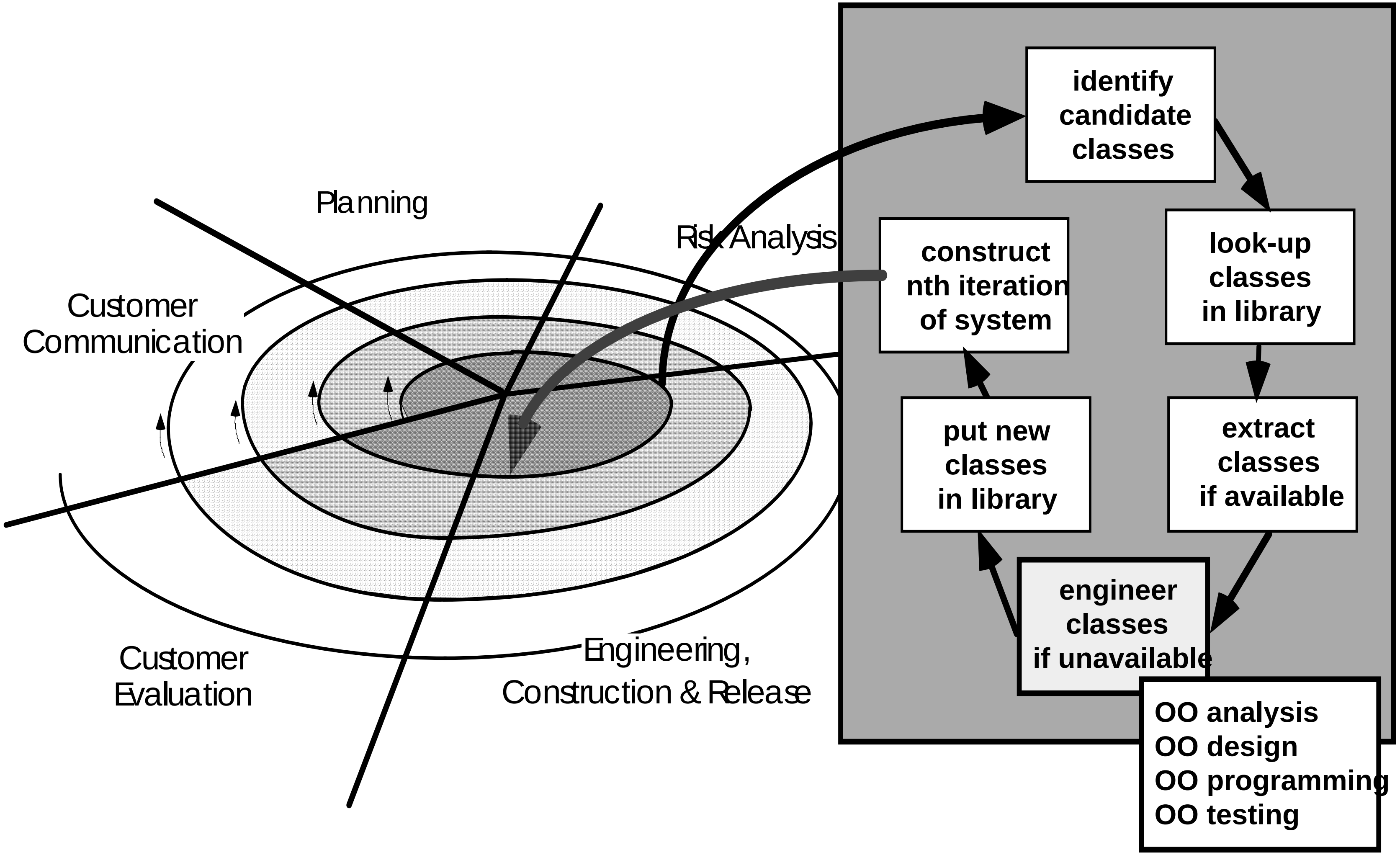
F_a **= the number of modules from the preceding release that were deleted in the current release**

OO classes, inheritance, and polymorphism

- Object technologies include the analysis, design, and testing phases of the development life-cycle, not just OOP

- OO systems tend to evolve over time so an evolutionary process model, such as the spiral model, is probably the best paradigm for OO software engineering

OO classes, inheritance, and polymorphism


Planning
Risk Analysis
Customer Communication
Customer Evaluation
Engineering, Construction & Release
identify candidate classes
look-up classes in library
extract classes if available
engineer classes if unavailable
put new classes in library
construct nth iteration of system
OO analysis
OO design
OO programming
OO testing

OO classes, inheritance, and polymorphism

- **What is an object-oriented approach?**

  One definition:

  It is the exploitation of class objects, with private data members and associated access functions.

OO classes, inheritance, and polymorphism

- **Key Concept: *Class***

  - ❑ A class is a ‘template’ for the specification of a particular collection of entities (e.g. a widget in a Graphic User Interface).

  - ❑ More formally, ‘a class is an OO concept that encapsulates the data and procedural abstractions that are required to describe the content and behaviour of some real-world entity’.

OO classes, inheritance, and polymorphism

- Key Concept: *Attributes*
  - ❑ Each class will have specific attributes associated with it (e.g. the position and size of the widget).
  - ❑ These attributes are queried using associated access functions (e.g. set_position)

OO classes, inheritance, and polymorphism

- **Key Concept: *Object***

  - □ **An object is a specific instance (or instantiation) of a class (e.g. a button or an input dialogue box).**

OO classes, inheritance, and polymorphism

- **Key Concept: *Data Members***
  - ❑ **The object will have data members representing the class attributes (e.g. int x, y;)**

OO classes, inheritance, and polymorphism

- Key Concept: *Access function*
  - ❑ The values of these data members are accessed using the access functions (e.g. set_position(x, y);)
  - ❑ These access functions are called methods (or services).
  - ❑ Since the methods tend to manipulate a limited number of attributes (i.e. data members) a given class tends to be cohesive.
  - ❑ Since communication occurs only through methods, a given class tends to be decoupled from other objects.

OO classes, inheritance, and polymorphism

- **Key Concept: *Encapsulation***
  - ❑ **The object (and class) encapsulates the data members (attributes), methods (access functions) in one logical entity.**

OO classes, inheritance, and polymorphism

- **Key Concept: Data Hiding**
  - ❑ **Furthermore, it allows the implementation of the data members to be hidden (why? Because the only way of getting access to them – of seeing them – is through the methods.) This is called data hiding.**

OO classes, inheritance, and polymorphism

- **Key Concept:** ***Abstraction***
  - ❑ **This separation, though data hiding, of physical implementation from logical access is called *abstraction*.**

OO classes, inheritance, and polymorphism

- **Key Concept: *Messages***
  - **Objects communicate with each other by sending messages (this just means that a method from one class calls a method from another method and information is passed as arguments).**

OO classes, inheritance, and polymorphism

- **Ellis and Stroustrup define OO as follows:**

  ‘The use of derived classes and virtual functions is often called object-oriented programming’

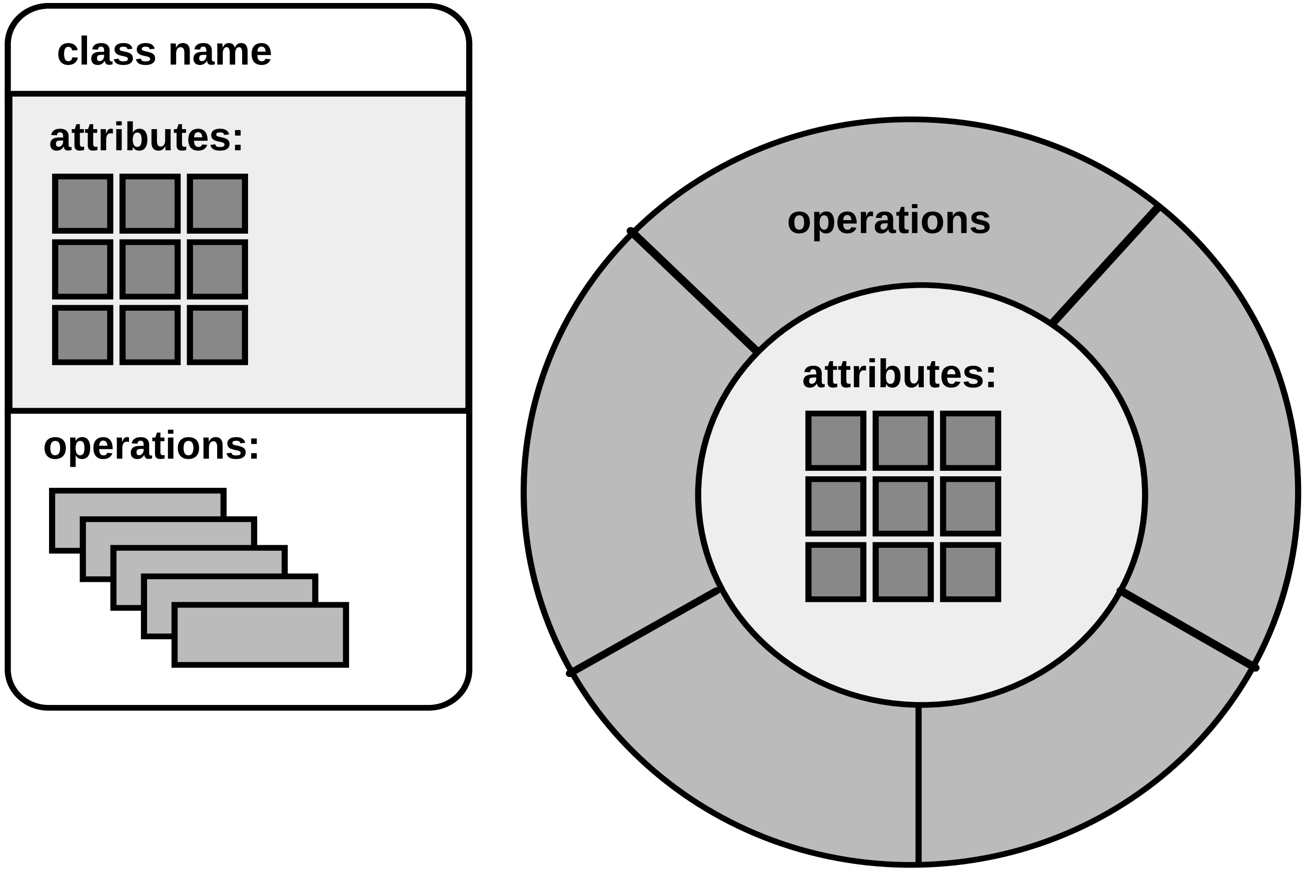
OO classes, inheritance, and polymorphism

- **Key Concept: Inheritance**
  - ❑ We can define a new class as a sub-class of an existing class
    - e.g. button is a sub-class of the widget class; a toggle button is a sub-class of the button class
  - ❑ Each sub-class inherits (has by default) the data members and methods of the parent class (the parent class is sometimes called a super-class)
    - For example, both the button and toggle button classes (and objects) have set_position() methods and (private) position data members x and y
  - ❑ A sub-class is sometimes called a derived class
  - ❑ The C++ programming language allows multiple inheritance, i.e. a sub-class can be derived from two or more super-classes and therefore inherit the attributes and methods from both
    - Multiple inheritance is a somewhat controversial capability as it can cause significant problems for managing the class hierarchy.

OO classes, inheritance, and polymorphism

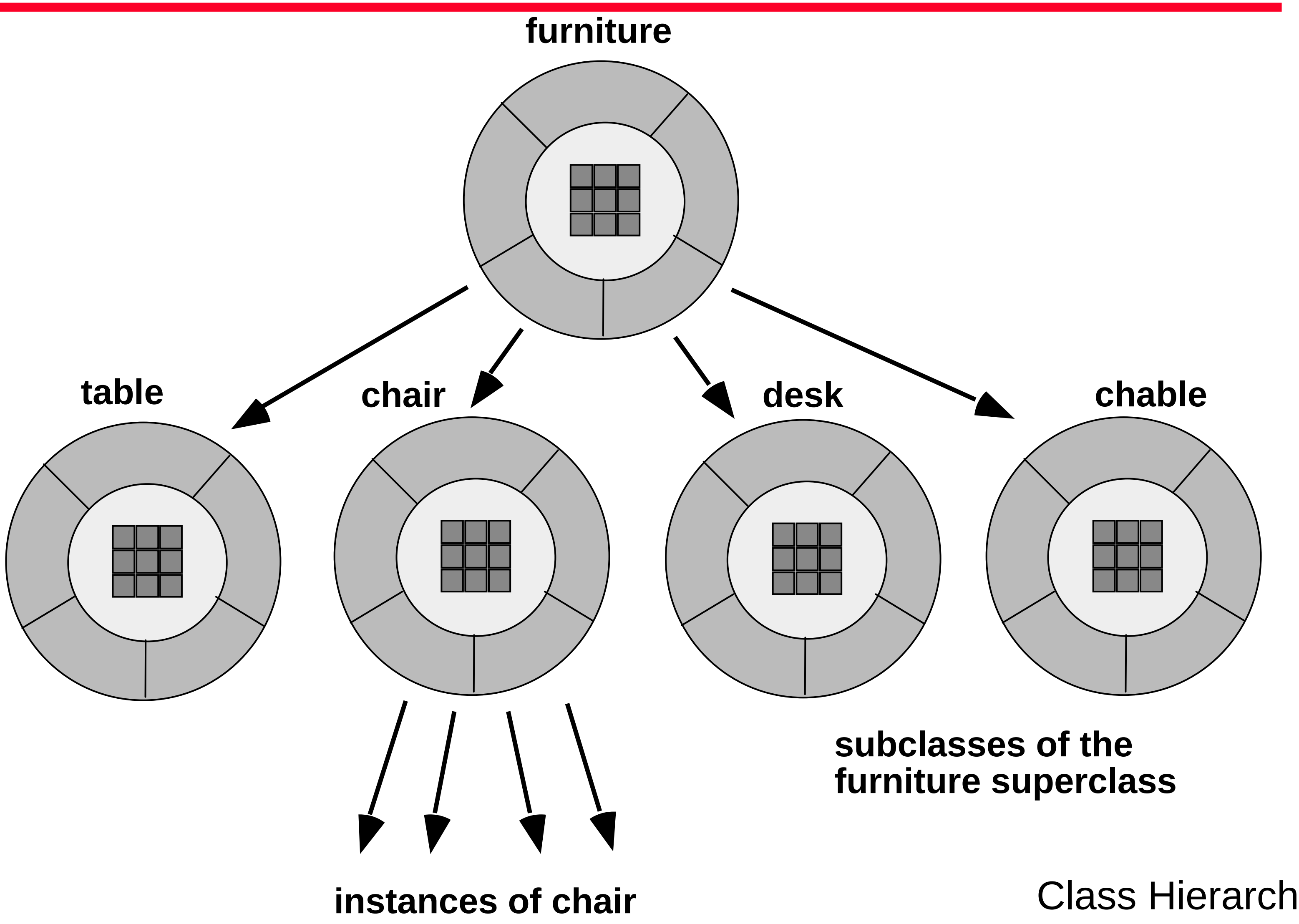
- **Key Concept: *Polymorphism***
  - ❑ If the super-class is designed appropriately, it is possible to re-define the meaning of some of the super-class methods to suit the particular needs of the derived class.
  - ❑ For example, the widget super-class might have a draw() method. Clearly, we need a different type of drawing for buttons and for input boxes.
  - ❑ However, we would like to use the one generic method draw() for both types of derived classes (i.e. for buttons and for input boxes).
  - ❑ OO programming languages allow us to do this. This is called polymorphism (literally: multiple structure) – the ability to define and choose the required method depending on the type of the derived class (or object) without changing the name of the method.
  - ❑ Thus, the draw() method has many structures, one for each derived class.

OO classes, inheritance, and polymorphism



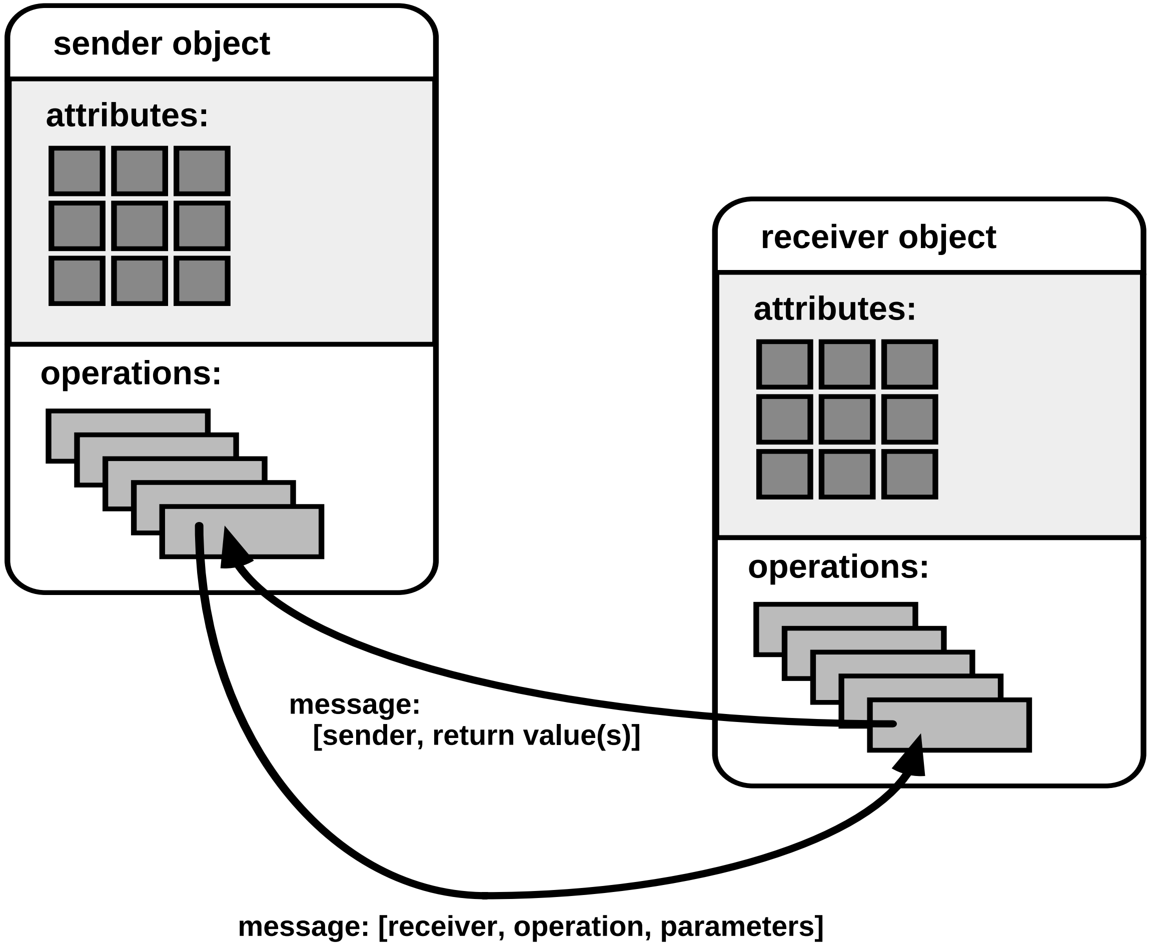
Two Views of a Class

OO classes, inheritance, and polymorphism



Class Hierarchy

OO classes, inheritance, and polymorphism



Message passing between objects

Object-Oriented Analysis - OOA

- **In order to build an analysis model, five basic principles are applied:**

  1. The information domain is modeled
  2. Module function is described
  3. Model behaviour is represented
  4. Models are partitioned (decomposed)
  5. Early models represent the essence of the problem; later models provide implementation details.

Object-Oriented Analysis - OOA

- **The goal of OOA is to define all classes (their relationships and behaviour) that are relevant to the problem to be solved. We do this by:**
  - ❑ Eliciting user requirements
  - ❑ Identifying classes (defining attributes and methods)
  - ❑ Specifying class hierarchies
  - ❑ Identifying object-to-object relationships
  - ❑ Modelling object behaviour

- ***These steps are reapplied iteratively until the model is complete***

Object-Oriented Analysis - OOA

- There are many OOA methods. For example:

- *The Booch Method*
  - ❑ Identify Classes and objects
  - ❑ Identify the semantics of classes and objects
  - ❑ Identify relationships among classes and objects
  - ❑ Conduct a series of refinements
  - ❑ Implement classes and objects

Object-Oriented Analysis - OOA

- ***The Coad and Yourdon Method***
  - ❑ Identify objects
  - ❑ Define a generalization-specification structure (gen-spec)
  - ❑ Define a whole-part structure
  - ❑ Identify subjects (subsystem components)
  - ❑ Define attributes
  - ❑ Define services

Object-Oriented Analysis - OOA

- ***The Jacobson Method (OOSE)***
  - ❑ **Relies heavily of use case modeling (how the user (person or devide) interacts with the product or system**

Object-Oriented Analysis - OOA

- ***The Rambaugh Method (Object Modelling Technique OMT)***
  - ❑ **Scope the problem**
  - ❑ **Build an object model**
  - ❑ **Develop a dynamic model**
  - ❑ **Construct a functional model**

Object-Oriented Analysis - OOA

- **There are seven generic steps in OOA:**

1. **Obtain customer requirements (identify scenarios or use cases; build a requirements model)**
2. **Select classes and objects using basic requirements**
3. **Identify attributes and operations for each object**
4. **Define structures and hierarchies that organize classes**
5. **Build an object-relationship model**
6. **Build an object-behaviour model**
7. **Review the OO analysis model against use cases / scenarios**

Object-Oriented Analysis - OOA

◆ ***Requirements Gathering and Use Cases***

- ❑ *Use cases* are a set of scenarios each of which identifies *a thread of usage* of the system to be constructed.
- ❑ They can be constructed by first identifying the *actors*: the people <u>or devices</u> that use the system (anything that communicates with the system).
- ❑ Note that an actor is not equivalent to a user: an actor reflects a particular role (a user may have many different roles, e.g. in configuration, normal, test, maintenance modes).

Object-Oriented Analysis - OOA

◆ *Requirements Gathering and Use Cases*

- ❑ Once the actors have been identified, on can then develop the use case, typically by answer the following questions:

  1. What are the main tasks or functions that are performed by the actor?
  2. What system information will the actor require, produce, or change?
  3. Will the actor have to inform the system about changes in the external environment?
  4. What information does the actor desire from the system?
  5. Does the actor wish to be informed about unexpected changes?

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***

  - ❑ CRC modeling provides a simple means for identifying and organizing the classes that are relevant to a system
  - ❑ *Responsibilities* are the attributes and operations that are relevant for the class (‘a responsibility is anything a class knows or does’)
  - ❑ *Collaborators* are those classes required to provide a class with the information needed to complete a responsibility (a collaboration implies either a request for information or a request for some action).

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***

  - ***Guidelines for Identifying Classes***
    - We said earlier that ‘a class is an OO concept that encapsulates the data and procedural abstractions that are required to describe the content and behaviour of some real-world entity’.
    - We can classify different types of entity and this will help identify the associated classes:
    - ***Device classes:***
      these model external entities such as sensors, motors, and key-boards.
    - ***Property classes*:**
      these represent some important property of the problem environment (e.g. credit rating)
    - **Interaction classes:**
      these model interactions what occur among other objects (e.g. purchase of a commodity).

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***

  - ***Guidelines for Identifying Classes***
    - In addition, objects/classes can be categorized by a set of characteristics:
    - ***Tangibility***
      does the class respresent a real device/physical object or does it represent abstract information?
    -
    - ***Inclusiveness***
      is the class atomic (it does not include other classes) or is it an aggregate (it includes at least one nested object)?
    -
    - ***Sequentiality***
      is the class concurrent (i.e. it has its own thread of control) or sequential (it is controlled by outside resources)?

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***

  - ❑ ***Guidelines for Identifying Classes***

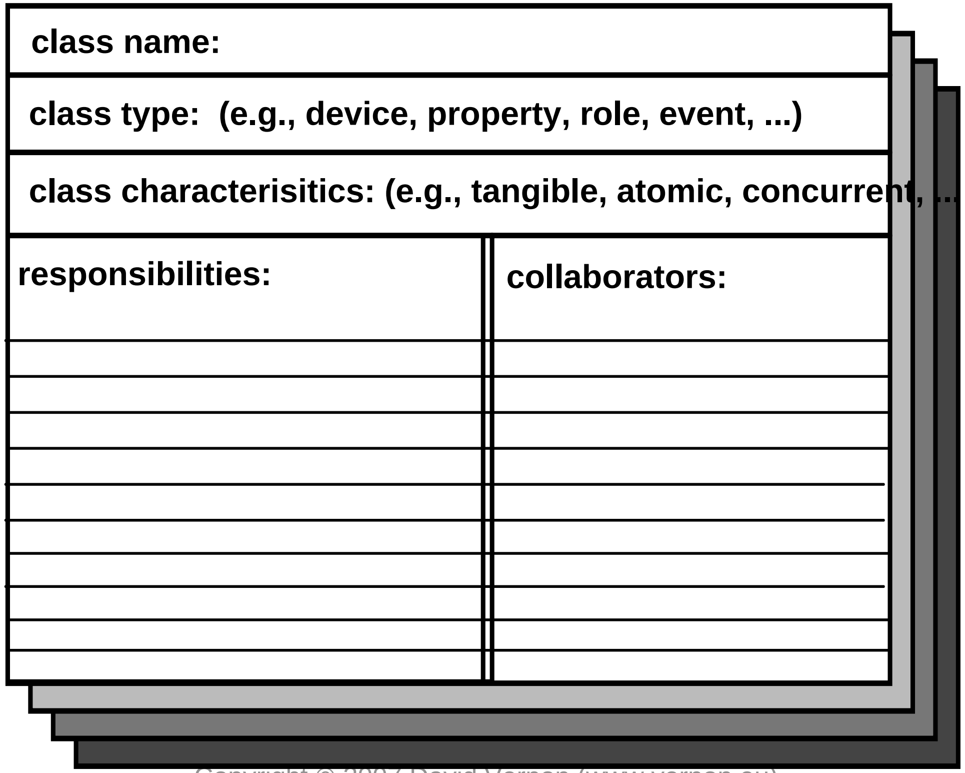
    - ***Persistence***
      is the class *transient* (i.e. is it created and removed during program operation); *temporary* (it is created during program operation and removed once the program terminates) or *permanent* (it is stored in, e.g., a database)?

    - ***Integrity***
      is the class corruptible (i.e. it does not protect its resources from outside influence) or it is guarded (i.e. the class enforces controls on access to its resources)?

    - For each class, we complete a CRC ‘index card’ noting these class types and characteristics, and listing all the collaborators and attributes for the class.

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***



class name:

class type: (e.g., device, property, role, event, ...)

class characterisitics: (e.g., tangible, atomic, concurrent, ...

responsibilities:

collaborators:

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***

  - ***Guidelines for assigning responsibilities to classes***

    - System intelligence should be evenly distributed.
    - Each responsibility should be stated as generally as possible.
    - Information and the behavior that is related to it should reside within the same class.
    - Information about one thing should be localized with a single class, not distributed across multiple classes.
    - Responsibilities should be shared among related classes, when appropriate.

Object-Oriented Analysis - OOA

- ***Class-Responsibility-Collaborator (CRC) Modelling***

  - ***Reviewing the CRC Model***

    - All participants in the review (of the CRC model) are given a subset of the CRC model index cards.
    - All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
    - The review leader reads the use-case deliberately. As the review leader comes to a named object, she passes the token to the person holding the corresponding class index card.
    - When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
    - If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards

Object-Oriented Analysis - OOA

- ***Defining Structures and Hierarchies***

  - ❑ The next step is to organize the classes identified in the CRC phase into hierarchies

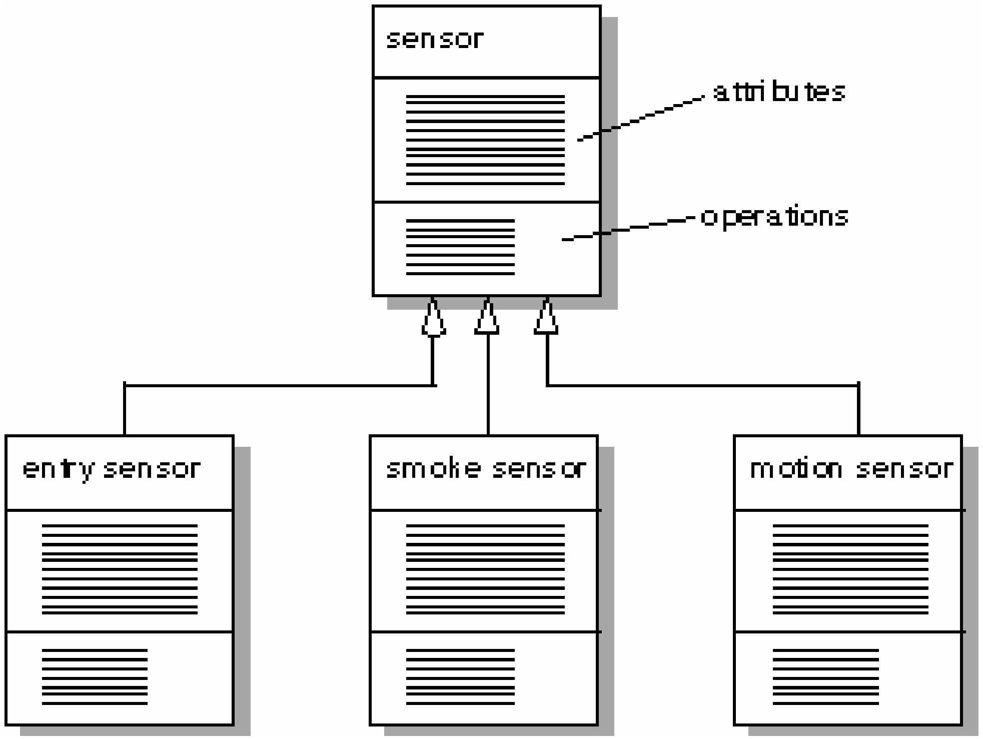
  - ❑ There are two types of hierarchy:

    1. Generalization-Specialization (Gen-Spec) structure

    2. Composite-Aggregate (Whole-Part) structure

Object-Oriented Analysis - OOA

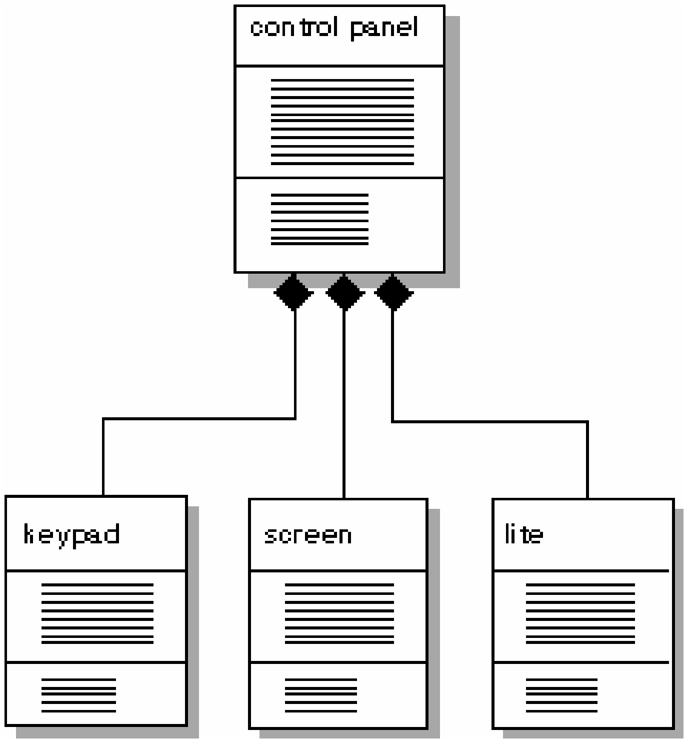
- ***Gen-Spec Hierarchy***



The relationship between classes in a Gen-Spec hierarchy can be viewed as a **'Is A'** relation.

Object-Oriented Analysis - OOA

- ***Composite-Aggregate Hierarchy***



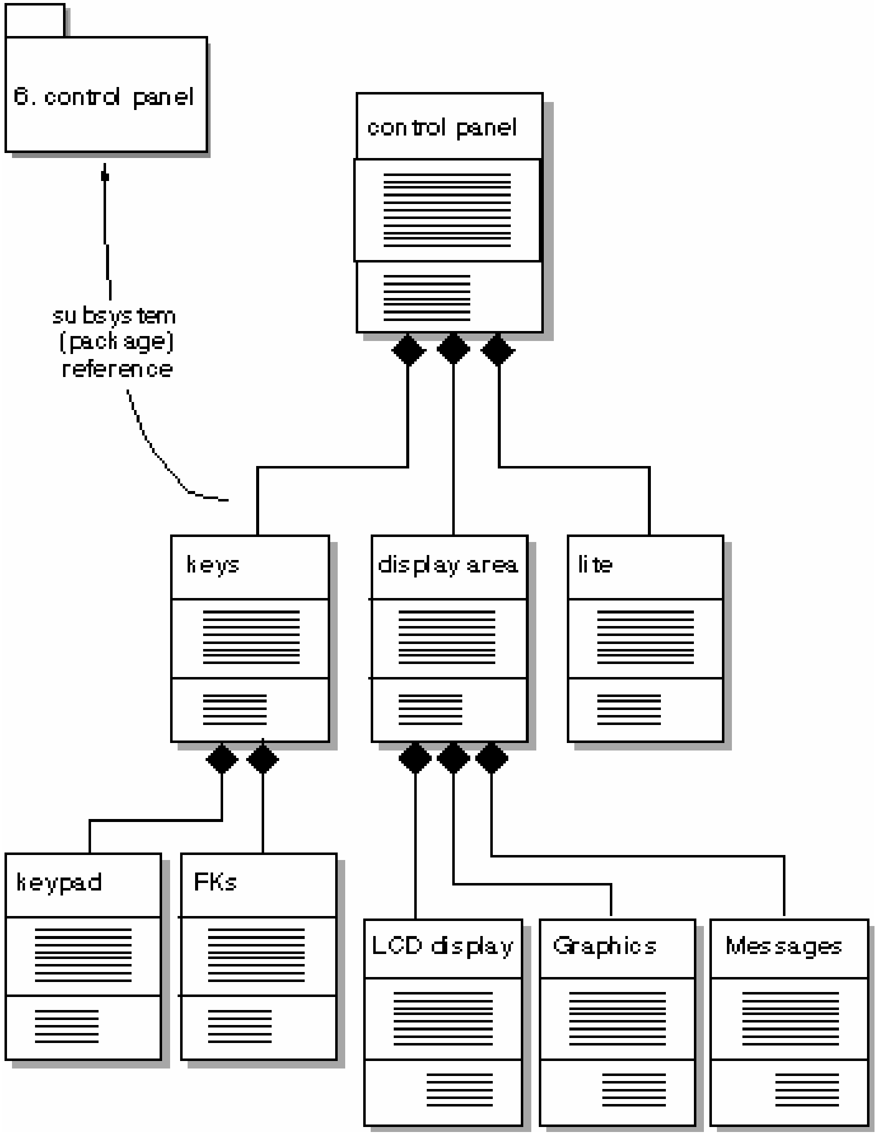
The relationship between classes in a composite-aggregate hierarchy can be viewed as a ‘**Has A**’ relation.

Object-Oriented Analysis - OOA

◆ *Defining Subjects and Subsystems*

- ❑ Once the class hierarchies have been identified, we then try to group them into subsystems or subjects.
- ❑ A subject / subsystem is a subset of classes that collaborate among themselves to accomplish a set of cohesive responsibilities.
- ❑ A subsystem / subject implements one or more contracts with its outside collaborators.
- ❑ A contract is a specific list of requests that collaborators can make of the subsystems.

Object-Oriented Analysis - OOA

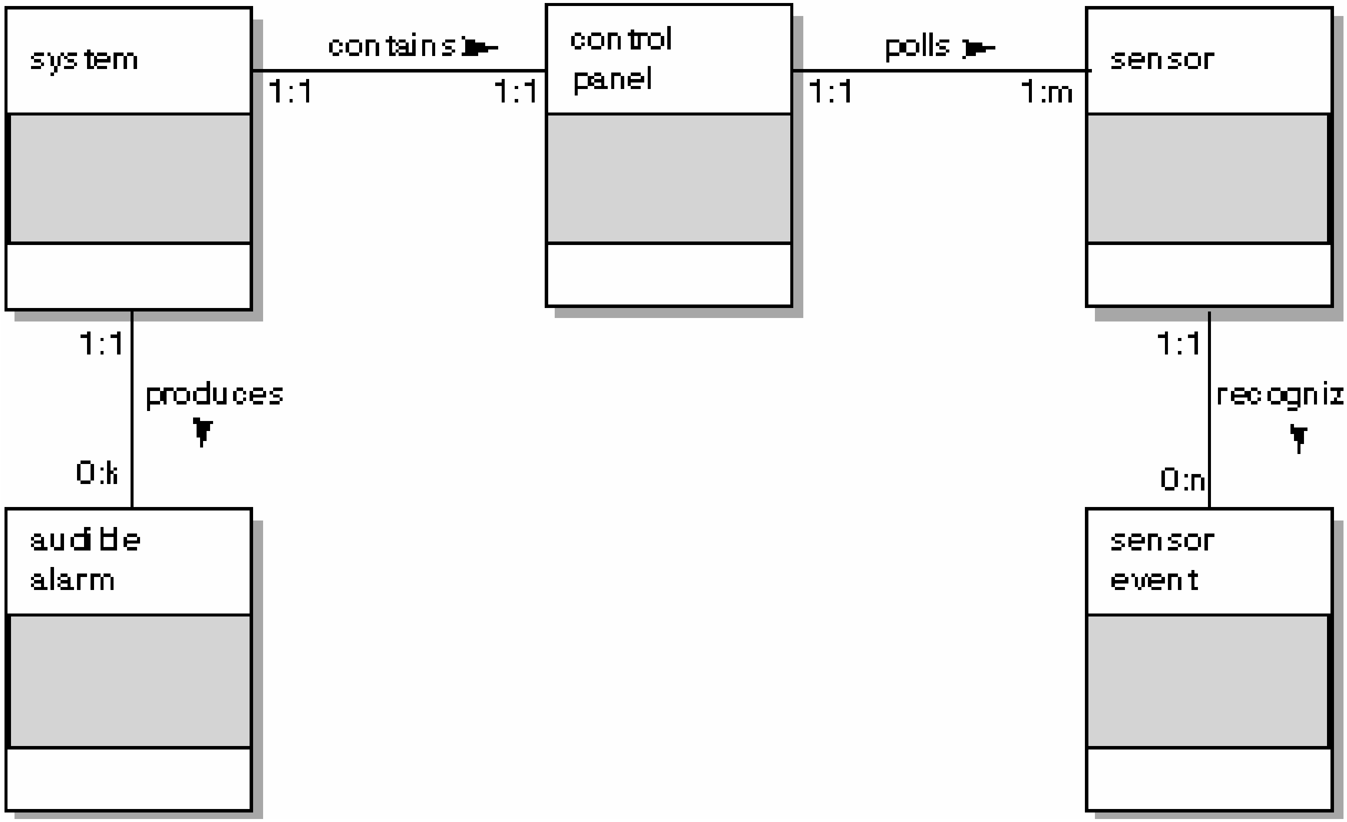


Object-Oriented Analysis - OOA

- ***The Object-Relationship Model***
  - ❑ The next step is to define those collaborator classes that help in achieving each responsibility.
  - ❑ This establishes a connection between classes. A relationship exists between any two classes that are connected
  - ❑ There are many different (but equivalent) graphical notations for representing the object-relationship model. All are the same as the entity-relationship diagrams that are used in modeling database systems and they depict the existence of a relationship (line) and the cardinality of the relationship (1:1, 1:n, n:n, etc).
  - ❑ In the following notation, the direction of the relation is also shown and the cardinality is show at both ends of the relationship line. A cardinality of zero implies a partial participation.

Object-Oriented Analysis - OOA

- ***The Object-Relationship Model***



Object-Oriented Analysis - OOA

◆ *The Object-Behaviour Model*

- ❑ The object-behaviour model indicates how an OO system will respond to external events or stimuli.

- ❑ To create the model, the you should perform the following steps:

    1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
    2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
    3. Create an event trace for each use-case.
    4. Build a state transition diagram for the system.
    5. Review the object-behavior model to verify accuracy and consistency.

Object-Oriented Analysis - OOA

- ***The Object-Behaviour Model***

  - ❑ In general, an event occurs whenever an OO system and an actor exchange information.

  - ❑ Note that an event is Boolean: an event is not the information that has been exchanged; it is the fact that information has been exchanged.

  - ❑ An actor should be identified for each event; the information that is exchanged should be noted and any conditions or constraints should be indicated.

  - ❑ Some events have an explicit impact on the flow of control of the use case, other have no direct impact on the flow of control.

Object-Oriented Analysis - OOA

◆ *The Object-Behaviour Model*

❑ For OO systems, two different characterizations of state must be considered

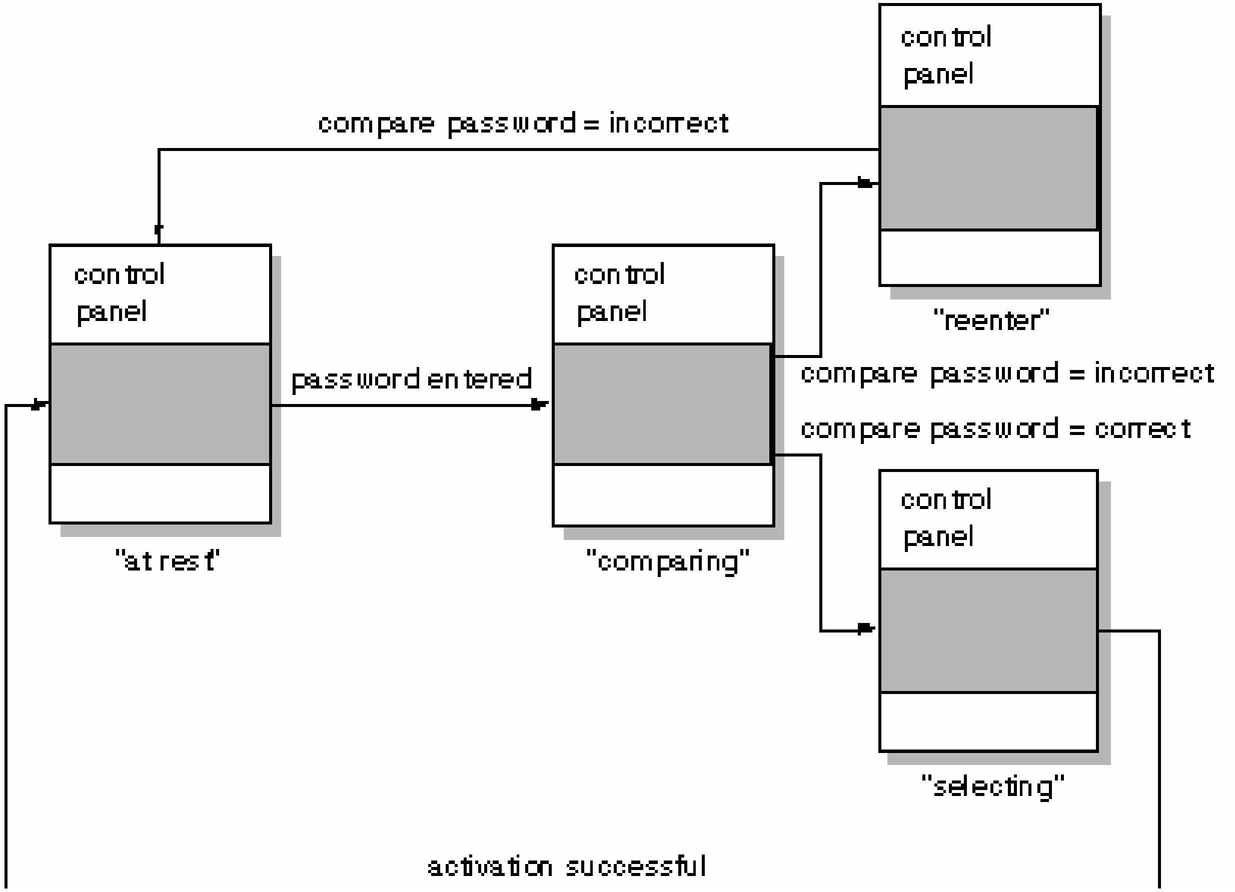
1. The state of each object as the system performs its function.
2. The state of the system as observed from outside as the system performs its function

❑ The state of an object can be both passive and active:

- The passive state is the current status of all an object’s attributes.
- The active state is the current status of the object as it undergoes a continuing transformation or process.

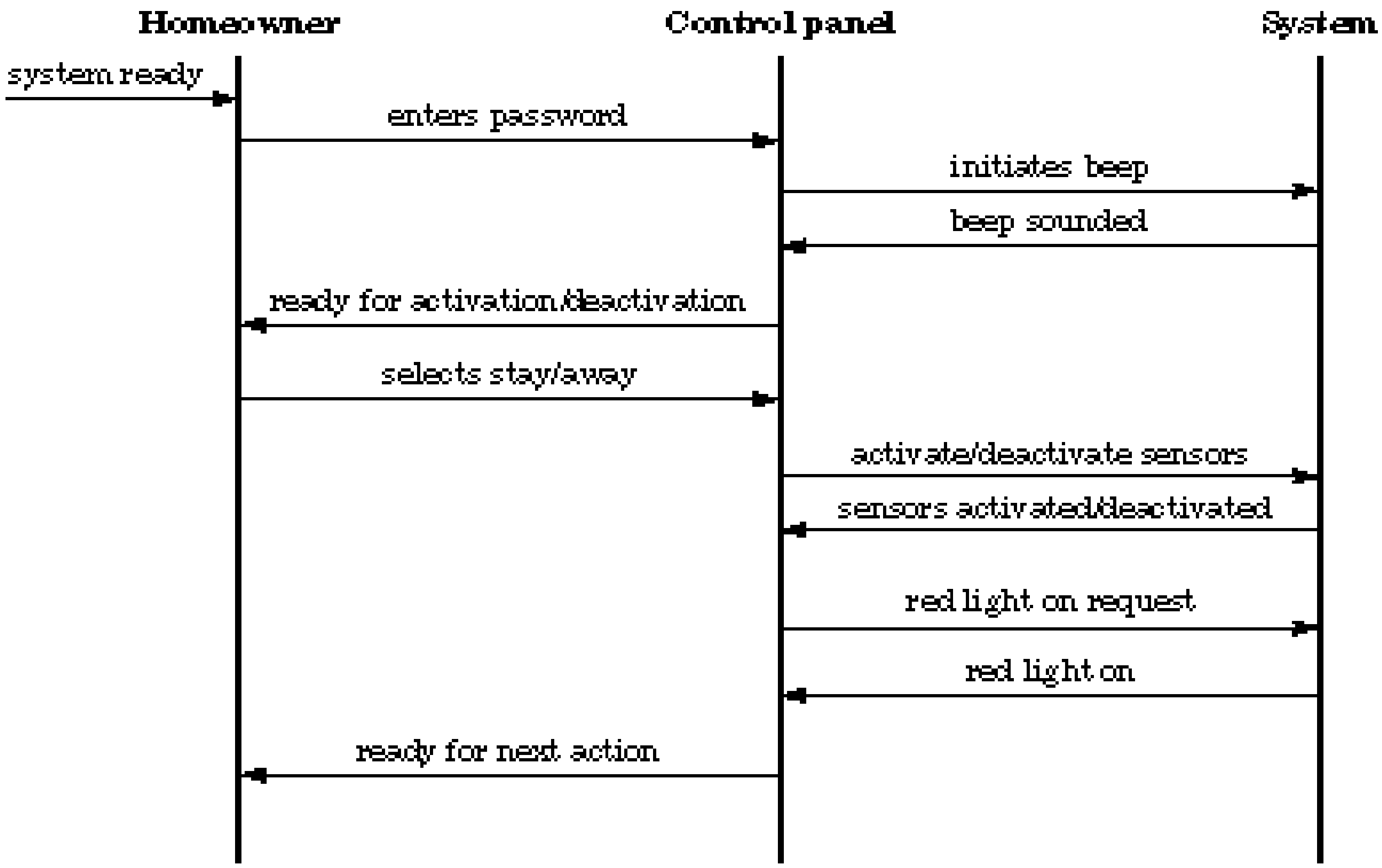
❑ An event (i.e. a trigger) must occur to force an object to make a transition from one active state to another.

Object-Oriented Analysis - OOA



A partial active state transition diagram for the object *control panel*

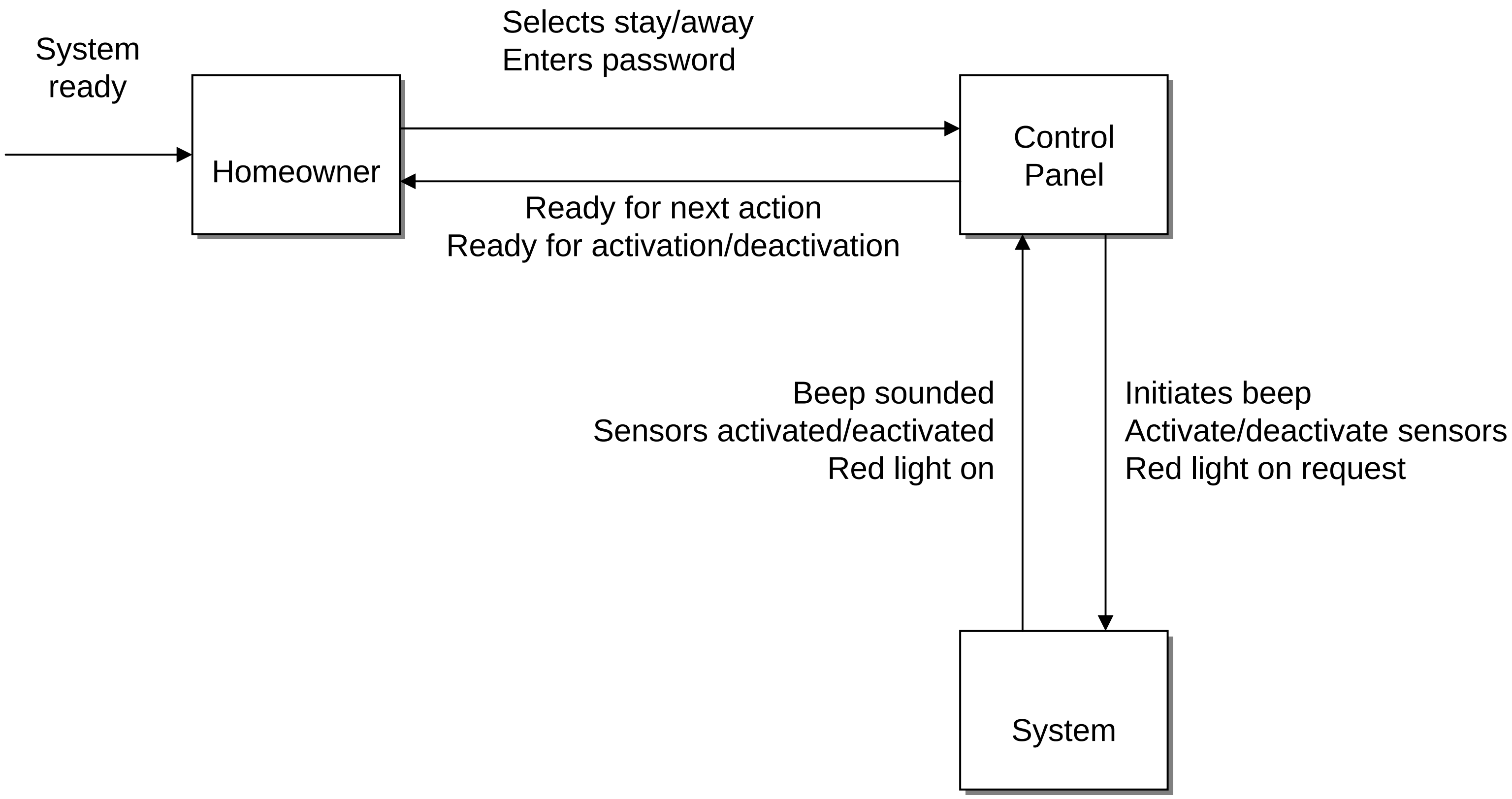
Object-Oriented Analysis - OOA



An *Event Trace* model: indicates how events cause transitions from object to object

An event trace is actually a shorthand version of the use case

Object-Oriented Analysis - OOA



System ready

Selects stay/away
Enters password

Homeowner

Control Panel

Ready for next action
Ready for activation/deactivation

Beep sounded
Sensors activated/eactivated
Red light on

Initiates beep
Activate/deactivate sensors
Red light on request

System

***Event flow diagram*: summary of all of the events that cause transitions between objects**

Object-Oriented Design - OOD

‘Designing object-oriented software is hard, and designing reusable object-oriented software is even harder … a reusable and flexible design is difficult if not impossible to get “right” the first time’

- OOD is a part of an iterative cycle of analysis and design
- Several iterations of which may be required before one proceeds to the OOP stage.

Object-Oriented Design - OOD

- There are many OOD approaches, almost all of which are the direct adjuncts of OOA approaches (e.g. the Booch method, the Coad and Yourdon Method, the Jacobson method, the Rambaugh method)

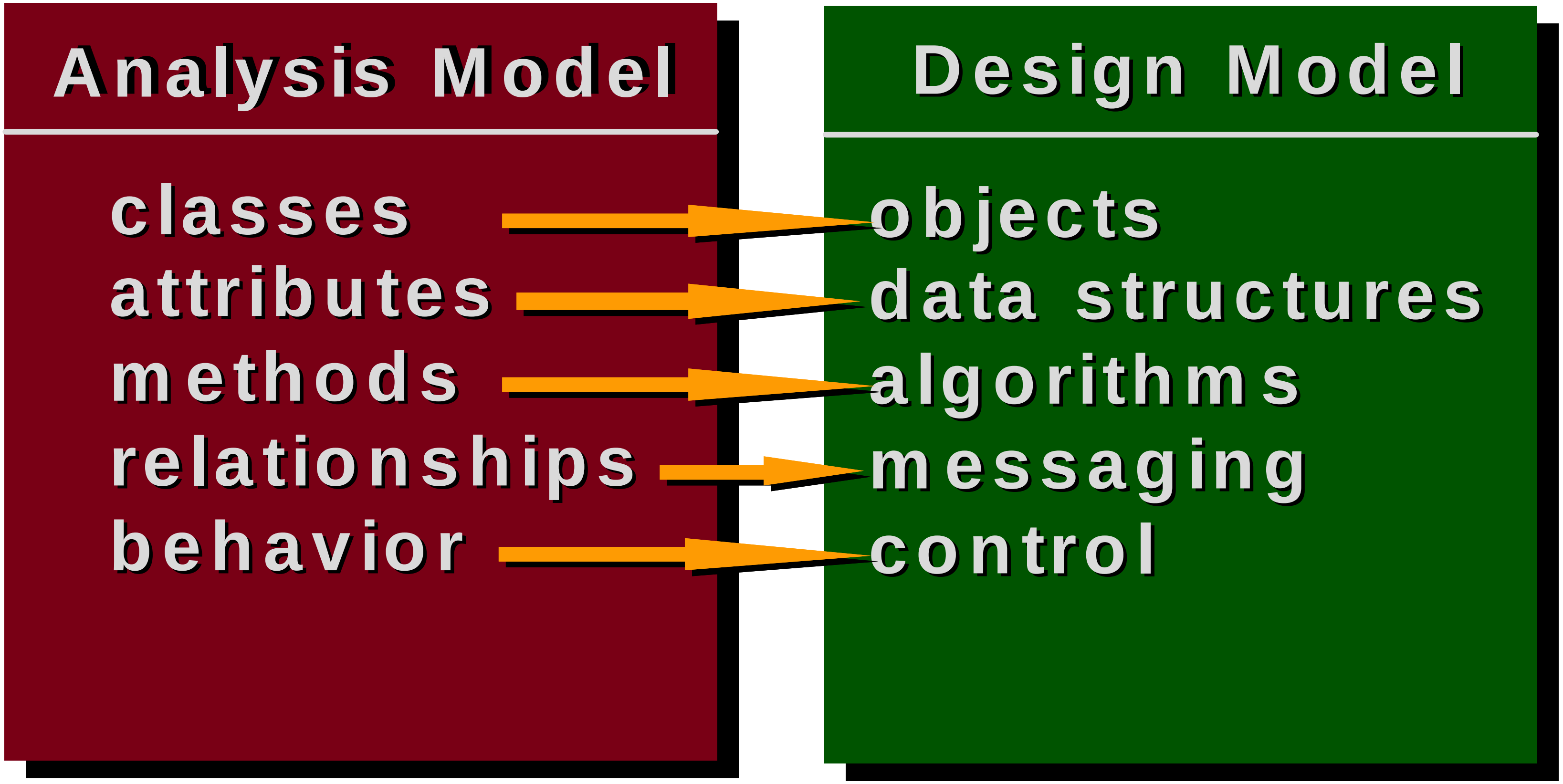
- The following gives just an overview of the issues that are common to all approaches.

- OOD is a critical process in the transition from OOA model to OO implementation (OO programming) because it requires you to set out the details of all aspects of the OOA model that will be needed when you come to write the code. At the same time, it allows you to validate the OOA model.

Object-Oriented Design - OOD

- Thus, the main goal in OOD is to make the OOA models less abstract by filling in all the details, but without going as far as writing the OO code.
- This will require you to state exactly:
  - ❑ how the attributes (data members) will be represented;
  - ❑ the algorithms and calling sequences required to effect the methods;
  - ❑ the protocols for effecting the messaging between the objects;
  - ❑ the control mechanism by which the system behaviour will be achieved (i.e. task management and HCI management).
- We focus on the algorithmic, representation, and interface issues;
  on how the system will be implemented, rather than on what will be implemented.

Object-Oriented Design - OOD



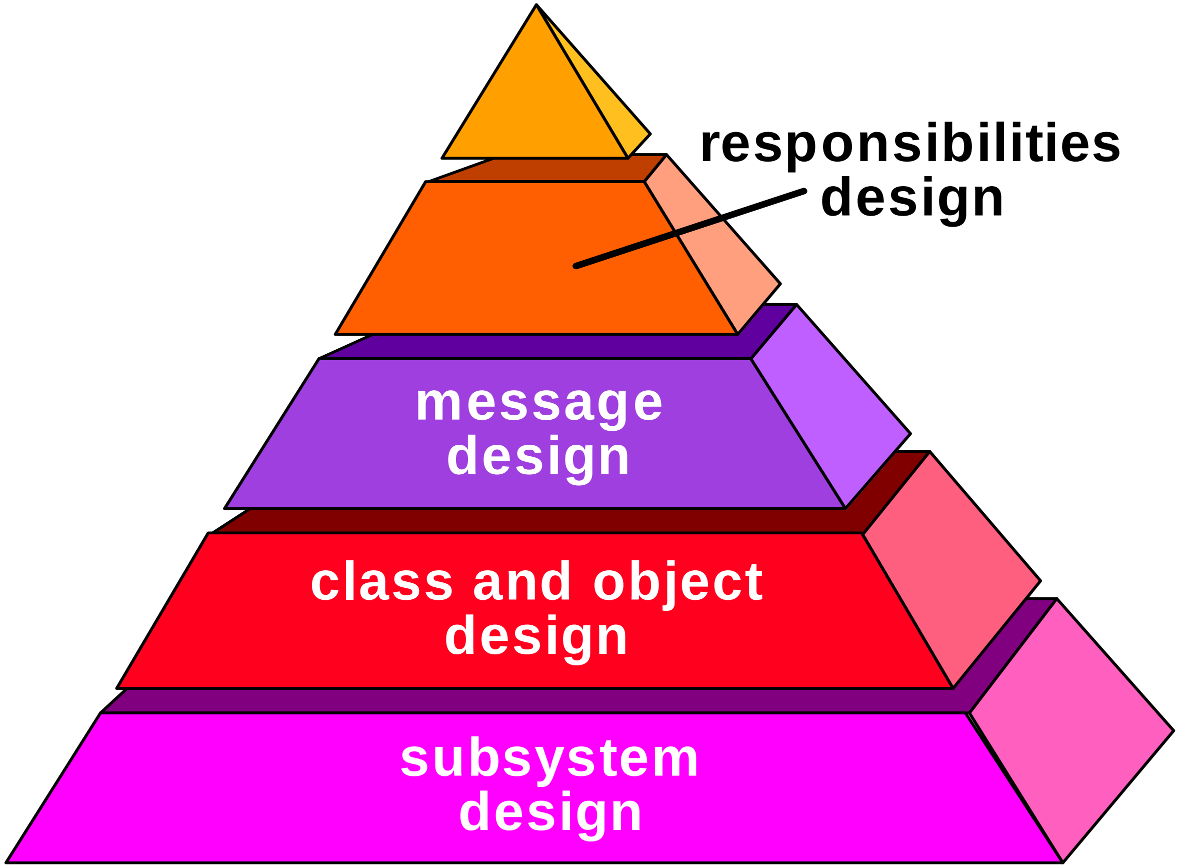
Analysis Model

classes
attributes
methods
relationships
behavior

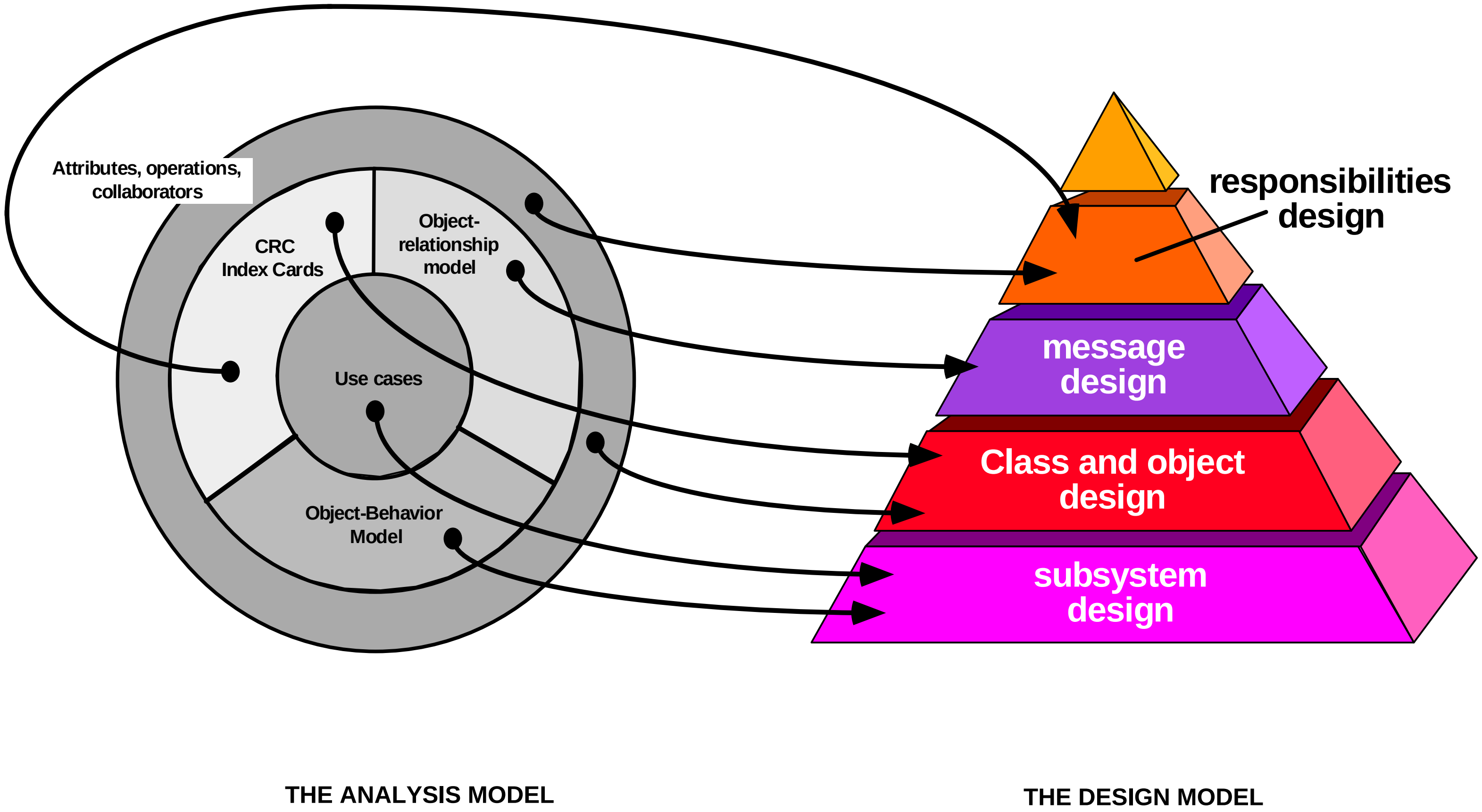
Design Model

objects
data structures
algorithms
messaging
control

Object-Oriented Design - OOD



Object-Oriented Design - OOD



Object-Oriented Design - OOD

- The software engineering will follow some standard steps in the design process:

  - ❑ Partition the analysis model into sub-systems
  - ❑ Identify concurrency that is dictated by the problem
  - ❑ Allocate subsystems to processors and tasks
  - ❑ Choose a basic strategy for implementing data management
  - ❑ Identify global resources and the control mechanism required to access them.
  - ❑ Design an appropriate control mechanism for the system
  - ❑ Consider how boundary conditions should be handled.
  - ❑ Review and consider trade-offs.

Object-Oriented Design - OOD

- Typically, there will be (at least) four different types of subsystem:

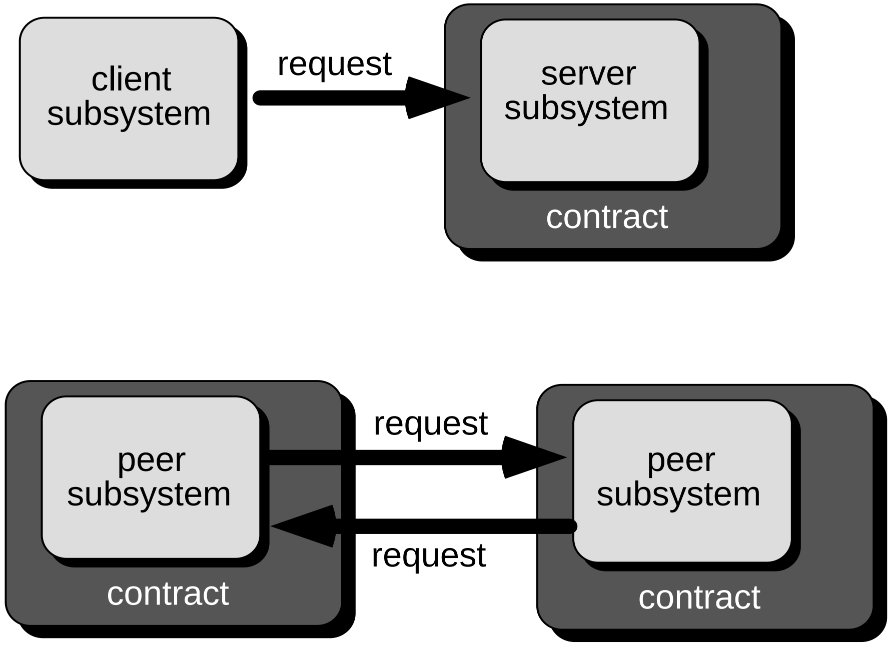
1. Problem domain subsystems: subsystems responsible for implementing customer/client requirements directly.

2. Human interaction subsystems: the subsystems that implement the user-interface (incorporating reusable GUI class libraries).

3. Task management subsystems: the subsystems that are responsible for controlling and coordinating concurrent tasks.

4. Data management subsystems: the subsystem(s) that is responsible for the storage and retrieval of objects.

Object-Oriented Design - OOD



Client/Server vs Peer-to-Peer Communication

Object-Oriented Design - OOD

- When designing a subsystem, the following guidelines may be useful:
  - ❑ The subsystem should have a well-defined interface through which all communication with the rest of the system occurs
  - ❑ With the exception of a small number of “communication classes,” the classes within a subsystem should collaborate only with other classes within the subsystem
  - ❑ The number of subsystems should be kept small
  - ❑ A subsystem can be partitioned internally to help reduce complexity

Object-Oriented Design - OOD

- We also have to design individual objects and classes
- Two distinct aspects
  - ❑ A *protocol description:* establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs.
  - ❑ An *implementation description:* shows implementation details for each operation implied by a message that is passed to an object. This will include:

    1. information about the object's private part
    2. internal details about the data structures that describe the object’s attributes
    3. procedural details that describe operations

Object-Oriented Testing - OOT

- Begin by evaluating the correctness and consistency of the OOA and OOD models
- Recognize that the testing strategy changes
  - ❑ the concept of the ‘unit’ broadens due to encapsulation
  - ❑ integration focuses on classes and their execution across a ‘thread’ or in the context of a usage scenario
  - ❑ validation uses conventional black box methods
- test case design draws on conventional methods (black-box testing and white-box testing) but also encompasses special features

Object-Oriented Testing - OOT

- ***Testing the CRC Model***
  1. Revisit the CRC model and the object-relationship model
  2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator’s definition
  3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source
  4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes
  5. Determine whether widely requested responsibilities might be combined into a single responsibility.
- Steps 1 to 5 are applied iteratively to each class and through each evolution of the OOA model.

Object-Oriented Testing - OOT

- ***OOT Strategy***

  - ❑ Encapsulation and inheritance make testing more complicated
  - ❑ Encapsulation:
    - the data members are effectively hidden and the test strategy needs to exercise both the access methods and the hidden data-structures
  - ❑ Inheritance (and polymorphism):
    - the invocation of a given method depends on the context (*i.e.* the derived class for which that method is called).
    - Consequently, you need to have a new set of tests for every new context (i.e. every new derived class).
    - Multiple inheritance makes the situation even more complicated.

Object-Oriented Testing - OOT

- ***OOT Strategy***

  - ❑ In conventional testing, we begin by unit testing and then proceed, incrementally, to test larger and larger sub-systems (i.e. integration testing). This approach has to be adapted for OO:

  - ❑ class testing is the equivalent of unit testing:
    - operations within the class are tested
    - the state behavior of the class is examined (very often, the state of an object is persistent).
  - ❑ integration testing requires three different strategies:
    - ***thread-based testing***—integrates the set of classes required to respond to one input or event (incremental integration may not be possible).
    - ***use-based testing***—integrates the set of classes required to respond to one use case
    - ***cluster testing***—integrates the set of classes required to demonstrate one collaboration

Object-Oriented Testing - OOT

- ***OO Test Case Design***

  - ❑ Each test case should be uniquely identified and should be explicitly associated with the class to be tested

  - ❑ The purpose of the test should be stated

  - ❑ A list of testing steps should be developed for each test and should contain:
    1. a list of specified states for the object that is to be tested
    2. a list of messages and operations that will be exercised as a consequence of the test
    3. a list of exceptions that may occur as the object is tested
    4. a list of external conditions (*i.e.,* changes in the environment external to the software that must exist in order to properly conduct the test)
    5. supplementary information that will aid in understanding or implementing the test.

Object-Oriented Testing - OOT

- ***OO Test Methods***
  - *Random testing*
    - Identify operations applicable to a class
    - Define constraints on their use
    - Identify a series of random but valid test sequences (a valid operation sequence for that class/object, *i.e.* a sequence of messages or method invocations for that class)

Object-Oriented Testing - OOT

- ***OO Test Methods***
  - *Partition Testing*
    - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software (*i.e.* input and output are categorized, and test cases are designed to exercise each category)
  - State-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - Attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - Category-based partitioning
    - categorize and test operations based on the generic function each performs

Object-Oriented Testing - OOT

- ***OO Test Methods***
  - ❑ *Inter-Class Testing*
    - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes
    - For each message that is generated, determine the collaborator class and the corresponding operator in the server object
    - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits
    - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

Object-Oriented Metrics

- **Project Metrics**
  - ❑ Since the achievement of re-use is such an important part of the OO strategy, LOC estimates of project size make little sense
  - ❑ FP estimates can be used effectively because the required information domain counts are readily available
  - ❑ There are a number of other OO-specific metrics.   These include the following

Object-Oriented Metrics

◆ Project Metrics

Number of scenario scripts

A scenario script is a detailed sequence of steps that describe the interaction between the user and the application. Each script is organized into triplets of the form {initiator, action, participant}

- *Initiator* is the object that requests some service
- *Action* is the result of the request
- *Participant* is the server object that satisfies the request

Number of key classes.

Key classes are highly-independent components (objects).

Object-Oriented Metrics

- **Project Metrics**

Number of support classes

Support classes are required to implement the system, but are not immediately related to the problem domain (e.g. GUI classes, database access classes, communication classes).

Average number of support classes per key class

Number of subsystems

A subsystem is an aggregation of classes that support a function that is visible to the end user of a system.

Object-Oriented Metrics

- **Project Estimation**

  - ❑ Although normal estimation techniques do apply for OO approaches, the historical database of OO projects is relatively small and so the empirical formulae may not be as accurate as you might wish them to be

  - ❑ Consequently, it may be useful to estimate project effort and cost using an OO-specific technique in addition to the normal approach (the more estimates you have, the better)

Object-Oriented Metrics

- **Project Estimation**

One OO-specific approach estimates effort as follows:

$$E = w * (k + s)$$

E is the estimated project effort.

W is the average number of person-days per class (typically, 15-20 person-days).

k is the number of key classes (estimated from the analysis phase)

s is the number of support classes (estimated as *m* * *k*, where m is a multiplier identified from the following table:

Interface Type	***m***
No GUI	2.0
Text-based user interface	2.25
GUI	2.5
Complex GUI	3.0

Object-Oriented Technical Metrics

- ***Design Metrics***

The CK (Chidamber and Kemerer) Metric Suite

❑ ***Weighted Methods per Class (WMC)***

$$WMC = \Sigma\ c_i$$

where:

c_i is the normalized complexity measure for method *i* (using, *e.g.* cyclomatic complexity measure)

Note that it is not straightforward to decide on the number of methods in a class

A low value of WMC implies a good design (cf. reusability, testability).

Object-Oriented Technical Metrics

- ***Design Metrics***

The CK (Chidamber and Kemerer) Metric Suite

- ❑ ***Depth of the Inheritance Tree (DIT)***

  DIT is the maximum length from the base class(root) to the derived classes (leaf nodes)

  A large DIT indicates leads to greater design complexity (but significant reuse).

Object-Oriented Technical Metrics

- ***Design Metrics***

The CK (Chidamber and Kemerer) Metric Suite

❑ ***Number of Children (NOC)***

NOC is the number of derived classes for a given class.

A lager value for NOC implies increased reuse, but also increased effort to test and possible diluted abstraction in the parent class.

Object-Oriented Technical Metrics

◆ *Design Metrics*

The CK (Chidamber and Kemerer) Metric Suite

❑ ***Coupling between Object Classes (CBO)***

CBO is the number of collaborators for a class.

‘As CBO increases, it is likely that the reusability of a class will decrease’.

Object-Oriented Technical Metrics

- ***Design Metrics***

The CK (Chidamber and Kemerer) Metric Suite

❑ ***Response Set for a Class (RFC)***

RFC is the set of methods that can potentially be executed in response to a message received by an object of that class.

As RFC increases, the effort required for testing increases.

Object-Oriented Technical Metrics

- ***Design Metrics***

The CK (Chidamber and Kemerer) Metric Suite

❑ ***Lack of Cohesion in Methods (LCOM)***

LCOM is the number of methods that access one or more of the same attributes.

In general, high values for LCOM imply that the class might be better designed by breaking it into two ore more separate classes.

Object-Oriented Technical Metrics

- ***Design Metrics***

The Lorenz and Kidd Metrics

❑ ***Class Size (CS)***

The overall class size can be determined by the total number of operations (both inherited and locally defined) that are encapsulated within the class and by the number of attributes (inherited and locally defined).

Large values of CS may indicate that the class has too much responsibility.

Object-Oriented Technical Metrics

- ***Design Metrics***

The Lorenz and Kidd Metrics

- ❑ ***Number of Operations Overridden by a subclass (NOO)***

  If NOO is large, the designer may have violated the abstraction implied by base class (note that this does not apply to pure virtual functions and abstract classes in C++).

Object-Oriented Technical Metrics

- ***Design Metrics***

The Lorenz and Kidd Metrics

❑ ***Number of Operations Added by a Subclass (NOA)***

As NOA increases, the subclass drifts away from the abstraction implied by the superclass.

In general, as the depth of the class hierarchy increases, the value for NOA at lower levels in the hierarchy should go down.

Object-Oriented Technical Metrics

- ***Design Metrics***

The Lorenz and Kidd Metrics

- ❑ ***Specialization Index (SI)***

$$SI = (NOO \times level) / M_{total}$$

where

level is the level in the hierarchy at which the class resides
M_{total} is the total number of methods for that class.

The higher the value of SI, the more likely that the class hierarchy has classes that do not conform to the superclass abstraction.

Object-Oriented Technical Metrics

- ***Design Metrics***

The Lorenz and Kidd Metrics

❑ ***Average Operation Size (OS)***

Either LOC or the number of messages sent by the operation (method) can be used to measure OS.

❑ ***Operation Complexity (OC)***

Any complexity measure can be used; OC should be as low as possible (to maximize cohesion of responsibility).

❑ ***Average Number of Parameters per Operation (NP)***

In general, NP should be kept as low as possible.

Object-Oriented Technical Metrics

- ***Test Metrics***

Encapsulation

- ❑ ***Lack of Cohesion in Methods (LCOM)***

- ❑ ***Percent Public and Protected (PAP)***

  The percentage ratio of public (and protected) attributes to private attributes; high values indicate likelihood of side effects among classes.

- ❑ ***Public Access to Data Members (PAD)***

  The number of classes (or methods) that can access another class's attributes. High values indicate the potential for side effects

Object-Oriented Technical Metrics

- ***Test Metrics***

Inheritance

❑ ***Number of Root Classes (NOR)***

The number of class hierarchies. Testing effort rises with NOR.

❑ ***Fan In (FIN)***

For OO, fan-in is an indication of multiple inheritance (the fan in is the number of base classes from which a sub-class is derived).

❑ ***Number of Children (NOC) and Depth of Inheritance Tree (DIT)***

Social, ethical, and professional issues

Code of Ethics

“A code of ethics functions like a technical standard, only it’s a standard of behaviour”

Joseph Herkert

former president of the IEEE Society on the Social Implications of Technology

Social, ethical, and professional issues

IEEE Code of Ethics

We, the members of the IEEE … agree:

- to accept responsibility in making engineering decisions consistent with the safety, health and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;
- to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;
- to be honest and realistic in stating claims or estimates based on available data;
- to reject bribery in all its forms;
- to improve the understanding of technology, its appropriate application, and potential consequences;

Social, ethical, and professional issues

IEEE Code of Ethics

- to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;
- to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;
- to treat fairly all persons regardless of such factors as race, religion, gender, disability, age, or national origin;
- to avoid injuring others, their property, reputation, or employment by false or malicious action;
- to assist colleagues and co-workers in their professional development and to support them in following this code of ethics.

Social, ethical, and professional issues

- **Software Engineering Code of Ethics and Professional Practice**

  Short Version

  Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Social, ethical, and professional issues

1. PUBLIC - Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.

Social, ethical, and professional issues

5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.

8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

The Software Engineer’s Dilemma

Customer to a software engineer:

“I know you believe you understood
what you think I said,
but I am not sure you realize
that what you heard is not what I meant”

R. Pressman