KHALIFA
UNIVERSITY

Year 5
Software Engineering 2
(Module 514)

Academic Session 2008-2009
Semester 2

COURSE NOTES

Professor D. Vernon

Course Outline

Aims

To provide a working knowledge of techniques for the estimation, design, building, and quality assurance of software projects.

Objectives

On completion of this module, it is expected that you will be able to:

1. Understand the relevance of measure and metric to software process improvement.
2. Compute FP, KLOC and other metrics.
3. Estimate the cost, effort, and duration of software projects using empirical techniques.
4. Estimate the cost, effort, and duration of software projects using the basic COCOMO Model.
5. Understand software quality assurance attributes, standards, and metrics for analysis.
6. Analyse software requirements and design solutions using both structured and object-oriented analysis and design techniques.
7. Understand the social, ethical and professional issues in software projects.

Course Contents

Software process and project metrics: measures vs. metrics, process improvement, LOC metrics, FP metrics, metrics for quality (correctness, maintainability, integrity, usability, defect removal efficiency). Software project planning: resources, estimation, decomposition, COCOMO estimation model. Project scheduling and tracking: human resources and effort, task definition, task networks, schedules. **Software quality assurance (SQA)**: quality attributes, software reviews, statistical quality assurance, software quality standards, McCall's quality factors, FURPS, metrics for analysis, design, implementation, testing, maintenance. **Object-oriented analysis, design, and testing:** OOD concepts, analysis and design issues, classes, inheritance, and polymorphism, testing strategies, metrics. **Social, ethical and professional issues:** code of ethics, copyright & security.

Recommended Textbooks

Software Engineering – A Practitioner's Approach
Roger S. Pressman
McGraw-Hill, 5th edition, 2001

Software Engineering
I. Sommerville,
Addison Wesley Longman, 6th edition, 2000

Recommended References

The Mythical Man-Month : Essays on Software Engineering
P. F. Brooks
Anniversary Edition, 1995
Addison-Wesley

Software Engineering: Methods and Management Analysis
A. von Mayrhauser and V. R. Basili
Academic Press, 1990

Software Engineering Concepts
R. Fairley
McGraw-Hill, 1995

1. A Review of Software Engineering

Software Engineering – A Definition[1]

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

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.

It is also concerned with the:

- Processes
- Methods
- Tools

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

Motivation

Consider the following statement[2]

"The scale of the software-dependent industry, often termed the secondary sector, extends far beyond the conventional software sector. It is estimated that more than half of the world's [1] current software is "embedded" in other products, where it largely determines their functionality. This percentage is set to increase dramatically over the next decade [2].

Software currently implements key, market-differentiating capabilities in sectors as diverse as automobiles, air travel, consumer electronics, financial services, and mobile phones and is adding major value in domestic appliances, house construction, medical devices, clothing and social care.

Future competitive advantage will be based on the *characteristics* of products sold or services provided. Many of those characteristics, such as *functionality, timeliness, cost, availability, reliability, interoperability, flexibility, simplicity* of use or other qualities, will be largely software-determined so that excellence in design, manufacturing or marketing will be to no avail without appropriate software. For example, 90% of the innovation in a modern car is software-based. Innovation, whether continuous or disruptive, will be delivered through quality software, and will determine the success of more and more products and services."

[1] In Germany the figure is greater than 70% (Source BMBF study 2001)
[2] "By 2010, there will be 16 billions Embedded Programmable Microcomponents (8 billions in 2003) or, in the average, 3 embedded devices per person worldwide" Artemis Technology position paper, June 2004 ; Available at: http://www.cordis.lu/ist/artemis/background.htm

[1] A. Finkelstein and J. Kramer, "Software Engineering: A Road Map" in "The Future of Software Engineering", Anthony Finkelstein (Ed.), ACM Press 2000.
[2] www.lero.ie: the Irish Software Engineering Research Centre

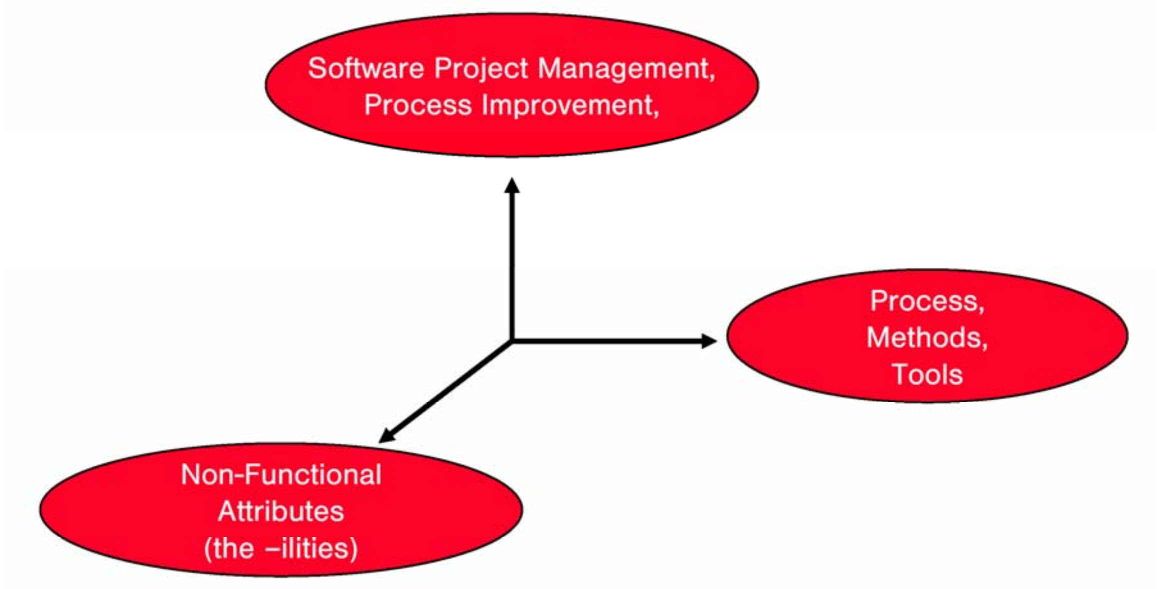
The Three Dimensions of Software Engineering

Software engineering is a rich, multi-faceted, and evolving field. It is often useful to think of it in three dimensions, each dimension being concerned with one particular aspect.

The first dimension contains all of the tools, techniques, methods, and processes required to develop software.

The second contains the management techniques required to organize software projects successfully, to monitor the effectiveness of the development, and to improve the development process.

The third addresses the way in which the non-functional attributes of the software being developed in achieved. Non-functional attributes refer *not* to what the software does (its function) but instead to the manner in which it does it (its dependability, security, composability, portability, interoperability … these are sometimes referred to as the '-ilities').



The Three Dimensions of Software Engineering

The Three Components of Software Engineering

There is also another way of looking at software engineering. This is sometimes referred to as the three Ps: People, Product and Process.

People are a very important aspect of software engineering and software systems. People use the system being developed, people design the system, people build the system, people maintain the system, and people pay for the system. Software engineering is as much about the organization and management of people as it is about technology.

There are typically two types of software **product**.

Generic products: these are stand-alone systems that are produced by a development organization and sold in the open market to any customer who wants to buy it.

Bespoke (customized) products: these are systems that are commissioned by a specific customer and developed specially by some contractor to meet a special need.

Most software expenditure is on generic products by most development effort is on bespoke systems.

The trend is towards the development of bespoke systems by integrating generic components (that must themselves be interoperable). This requires two of the non-functional properties (one of the ‘ilities’) mentioned earlier: composability and interoperability.

The **software process** is a structured set of activities required to develop a software system:

- Specification
- Design
- Validation
- Evolution

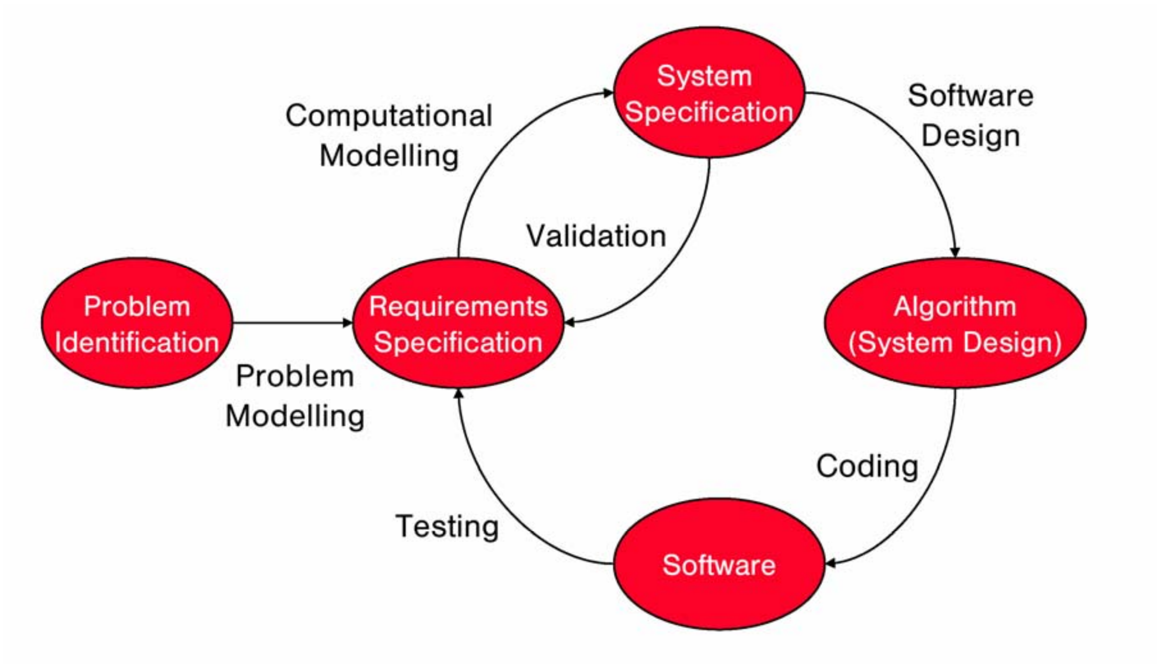
These activities vary depending on the organization and the type of system being developed. There are several different process models and the correct model must be chosen to match the organization and the project.

Let’s now look more deeply at the different types of software process.

The Basic Software Development Process

To start with, we will recall the software development process that we used when learning how to program.

This process has a number of activities and a number of (resultant) outcomes. In the following diagram, the activities are the arrows and the outcomes are the bubbles.



We will follow the diagram sequentially.

First we have the **Problem Identification** outcome. This is actually the result of a lot of background work in the general area of the problem. Call it experience. It is an ability to look at a domain (e.g. telecommunications or engine control) and to identify the issue that needs to be addressed and the problem to be solved (*e.g.* elimination of noise or cross-talk on a communication channel, or engine control for temperature-dependent fuel efficiency). It also required an understanding of the theoretical issues by which we can model this problem.

This gives us our first activity: **problem modelling**.

Working with the end-users of our software system, we can then state clearly the **requirements** of the system: what the user needs it to do (*e.g.* a bank may want to offer face recognition as a security feature on it's ATMs: the requirements will state the expected behaviour of the system).

From this, we can abstract the problem from the problem space and **model it computationally**: this means we can identify the theoretical tools we need to solve the problem (statistical analysis for the elimination of noise on the communication channel; characterization of the relationship between fuel consumption and engineer cylinder temperature for the engine control; the extraction of facial features from images and the statistical classification techniques used to match these feature with faces in a database).

The result of this process of modeling and abstraction is a complete statement of the **computational model**: a definition of

- the information that the system requires to solve the problem (the inputs)
- the explicit techniques used to process/analyze/transform that information (the computational model)
- the information that will be produced (the outputs)

It will also identify how the information needs to be presented to the system in great detail and how the information will be output (or presented to the user), again in great detail.

Ideally, it will also include a decomposition of the major functional blocks involved in the processing/analysis/transformation (i.e. a modular decomposition of the computational model). This information is collectively known as the '**system specification'**.

Once you have this specification, before proceeding you must return and see if it actually matches what the user needs: i.e. you need to **validate** that the system specification satisfies the requirement (you would be surprised how often it doesn't). If it does, you can proceed to the next activity: software design. If it doesn't, then either the requirements were wrong and need to be changed, or the specification was wrong, and needs to be changed, or, more likely, both were wrong and need to be changed.

Software design is a process that allows you to turn your computational model into a workable algorithm to solve the problem. Typically, it will require

- a functional decomposition of the overall system into manageable components (at the very least: data input, data validation, data processing / analysis / transformation, data output),
- explicit algorithms for each block (either using pseudo-code or flow-charts)
- a definition of the data-structures that will represent the input information, the output information, and any internal information representations,
- a definition of which functional components access and/or change which data-structures.

You now have your system design. At this point you can begin **coding**, i.e. programming. Note that you will never try to code the entire system in one go. You will always code up one module at a time, and one data-structure at a time, building your system incrementally.

Similarly, you **test** your software incrementally, as you add more and more modules.

You will need to test for

- verification (given certain inputs, is it computing the right answer)
- validation (is it doing what we want, i.e. is it satifying the requirements)
- evaluation (how well is it doing it compared to other systems or establishes metrics).

Although not shown on the diagram, there is another process or activity called **maintenance**, which is concerned with continually updating the system to allow for changes in requirements and corrections to errors.

Finally, there is one extremely important activity that applies to all outcomes: **documentation**. It is essential that every outcome is fully described and documented in great detail.

The Engineering Process Model[3]

We can see that the basic software development process is very similar to the standard 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

Generic Software Process Models

There are many different software process models. Here are just four (we will cover one additional one – the spiral model – in more depth shortly).

- The waterfall model
  Separate and distinct phases of specification and development; (more below)

- Evolutionary development
  Specification and development are interleaved (more below)

- Formal transformation
  A mathematical system model is formally transformed to an implementation

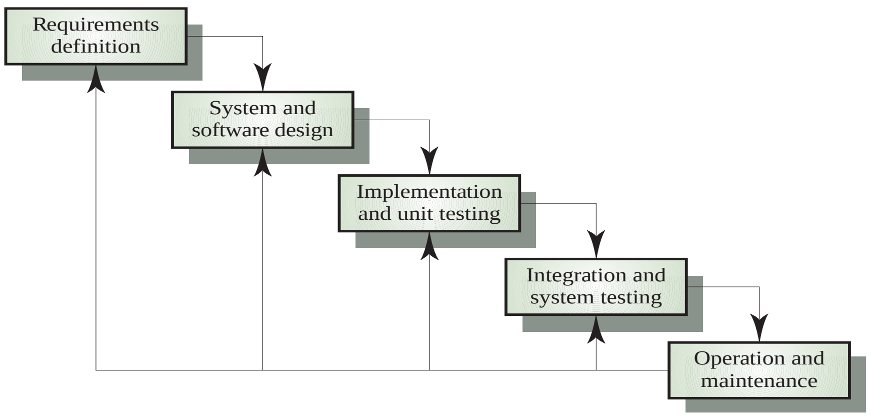
- Reuse-based development
  The system is assembled from existing components

The Waterfall Model

The waterfall model has separate and distinct phases of specification and development. These are:

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

Each one follows the other sequentially and doesn't begin until the previous one is finished. The drawback of the waterfall model is the difficulty of accommodating change after the process is underway (e.g. what happens if you find a flaw in the design after you have started implementation?).


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

[3] Please refer also to Chapter 1 of 'Software Engineering', I. Sommerville.

The Evolutionary Development Model

There are two types of evolutionary development:

1. Exploratory prototyping
   The objective is to work with customers and to evolve a final system from an initial outline specification (i.e. build and adapt the system incrementally). Here, you should start with well-understood requirements

2. Throw-away prototyping
   In this case, the objective is to understand (find out) the system requirements. The idea is that you quickly build a prototype and see if it suits the client's needs. If it does, you continue with development; if, as is more likely, it doesn't, you throw it away and start again with a better idea of what is needed. Here, you normally start with poorly understood requirements

There are problems with both of these approaches.

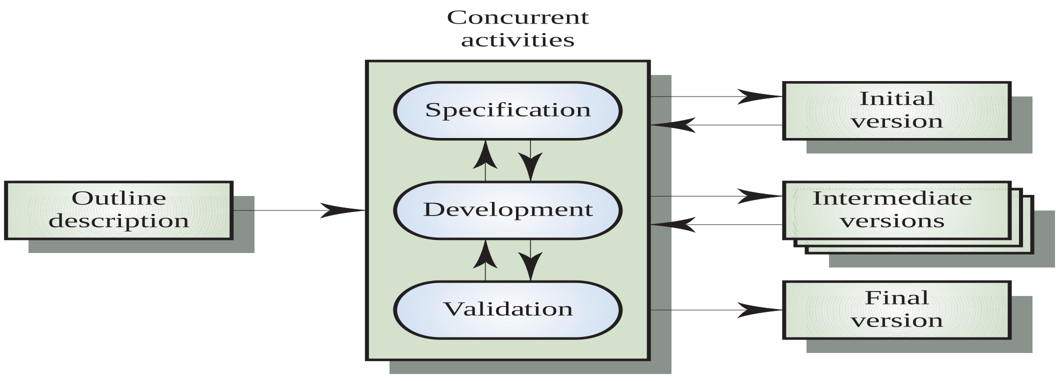
First, there is a lack of process visibility (we say more about this in a moment, just note here that it means that it is difficult to assess the real progress through documented analysis).

Second, the resultant systems are often poorly structured.

Third, special skills (e.g. in languages for rapid prototyping) may be required

Typically, this approach is useful in the following situations:

- For small or medium-size interactive systems
- For parts of large systems (e.g. the user interface)
- For short-lifetime systems



Risk Management

One of the principal tasks of a manager is to minimise risk of failure associated with each activity.

The 'risk' inherent in an activity is a measure of the uncertainty of the outcome of that activity.

High-risk activities (usually but not always) cause schedule and cost overruns.

Risk is related to the amount and quality of available information. The less information, the higher the risk.

Different process models have different levels of risk:

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 and the same process model does not have to be used for all subsystems: you can (and should) choose the process model which is best suited to the type of sub-system you are developing.

For example, you might use the prototyping/evolutionary model for high-risk specifications and the waterfall model for well-understood developments.

The Spiral Model of the Software Process

This was developed by B. Boehm and it tries to explicitly deal with the risk associated with each activity in the software development process.

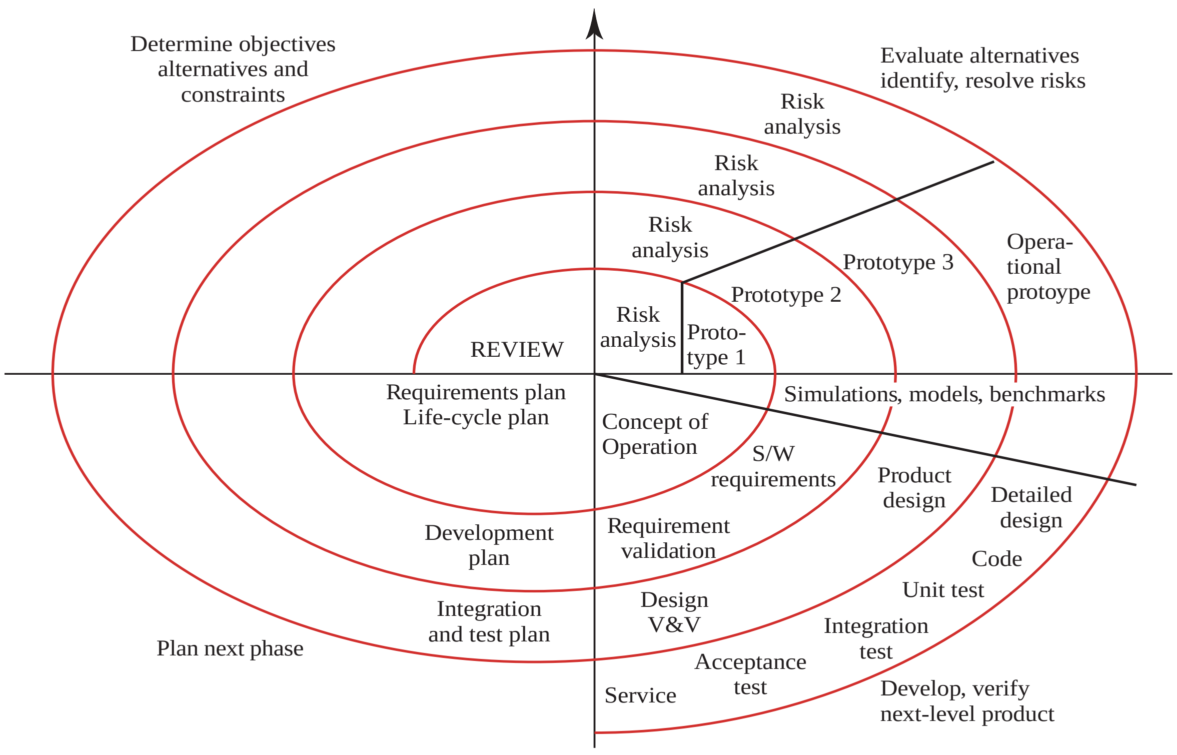
There are four phases in each revolution around the spiral; each revolution is concerned with just one activity in the software development process.

1. *Objective setting*
   Specific objectives for the project phase are identified

2. *Risk assessment and reduction*
   Key risks are identified, analysed and information is sought to reduce these risks

3. *Development and validation*
   An appropriate model is chosen for the next phase of development.

4. *Planning*
   The project is reviewed and plans drawn up for the next round of the spiral



The following is a template for the various activities in a single revolution around the spiral (remember, these activities are concerned with minimizing the risk associated with this particular part of the software development process in question):

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

Here's how these activities might be used for software development task to improve software quality:

- Objectives
  - Significantly improve software quality

- Constraints
  - Within a three-year timescale
  - No large-scale capital investment
  - No 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

Note that the spiral model is a hybrid model: you can use whatever conventional model you want with it:

- If you are using the waterfall model (for well-understood systems), then each phase of the waterfall will probably correspond to one revolution
- If you are using the evolutionary model (for uncertain systems, such as user interfaces), then each phase may correspond to a distinct evolutionary iteration or prototype.

The spiral model has many 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

But it also has a few 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 – you can't see and feel them like you can with a manufactured product like cars, keyboards, cameras – 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

The waterfall model is still the most widely used deliverable-based model. Each phase in the process must be completed before the next phase begins and this provides a natural breakpoint to create and check the appropriate documentation.

The following are some of the documents that might be produced using the waterfall process model:

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

Finally, let's review the visibility associated with each process model:

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.

‘Software Engineering is Not Enough’

Software engineering courses and textbooks tend to emphasize the management aspects and process aspects of software development (there is not one single program listing in the textbook[4] for this course).

While software engineering is certainly important, it is not everything.

The following points, taken from a recent article in IEEE Software[5], make the argument.

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.’

‘Software development … is the hardest and least understood part of the software engineering life cycle. 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.’

‘There is no magic method that guarantees a well-engineered product.’

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. When you have written a program in its entirety, you have forced yourself to understand every aspect of the problem and the solution.’

Documentation is Essential

‘There is rarely such a thing as too much documentation … the only hope for understanding programs is good documentation of control structure blocks and detailed descriptions of the purpose and use of data structures.’

‘Documentation – often exceeding the source code in size – is a requirement, not an option.’

You Must Validate Data

‘Good developers understand that they cannot trust user inputs. Each time an input enters the system it must be validated to prevent failure or corruption of internal data.’

‘Developers quickly learn that every program has two parts: the code that performs the desired function and the code that handles failure [and validation]’

‘You must painstakingly validate each parameter of each call, often meaning a lot of If statements and calls to validation routines.’

‘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.’

[4] R. Pressman, Software Engineering: A Practitioner’s Approach, McGraw-Hill, New York 1997.
[5] J. A. Whittaker and S. Atkin, “Software Engineering Is Not Enough”, IEEE Software, July/August 2002, pp. 108-115.

Failure is Inevitable – You Need To Handle It

‘Exception handlers … raising exceptions is not the same thing as programming constraints. Constraints actually prevent failure. Exceptions, on the other hand, let the failure occur and then trap it. The trapped failure can then be handled by a special routine that the developer provides.’

‘Failure recovery is often difficult.’

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

‘Before design, developers must pursue two activities: familiarizing themselves with the problems they are to solve (we’ll call this *problem-domain expertise*) and studying the tools they will use to solve them (we’ll call this *solution-domain expertise*). Expertise in both domains is crucial for project success.’

‘Learning the problem domain means more than simply talking to users and gathering requirements.’

‘The solution domain … consists of the tools that a team of developers and testers employ to build the software product … 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.’

2. Software Process and Project Metrics[6]

Measurement

Measurement is fundamental to any engineering discipline.

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

Measurement is applied to:

The software ***process*** to improve it

A software ***project*** to assist in

- ♦ Estimation
- ♦ Quality control
- ♦ Productivity assessement
- ♦ Project control

A software ***product*** to assess its quality

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.

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

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.)

[6] Please refer also the Chapter 4 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman.

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 metric 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 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.

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 styles; they don't easily accommodate non-procedural languages.

Function-Oriented Metrics

Since functionality cannot be measured directly, it must be derived indirectly using other direct measures. The *function point* metric is the most common function-oriented metric.

Function points are computed as follows.

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 respository of information
External interface	a machine readable interface such as port, disk, tape, CD

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

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

Extended Function Point Metrics

The function point metric was originally designed to be applied to business information systems applications which typically focus on information processing and transaction processing. 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.

3D Function Point Metric

The key idea with the 3D Function Point metric is to extend the standard FP to include not only the complexity of the data processing but also the functional (algorithmic) complexity. The software system is characterized in three dimensions:

1. The **data** dimension. This is evaluated in much the same way as the normal FP. In this case, you count the number of internal data-structures, external data sources, user inputs, user outputs, and user inquiries, each being assigned a complexity attribute of low, average, or high.

2. The **functional** dimension. This is 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.

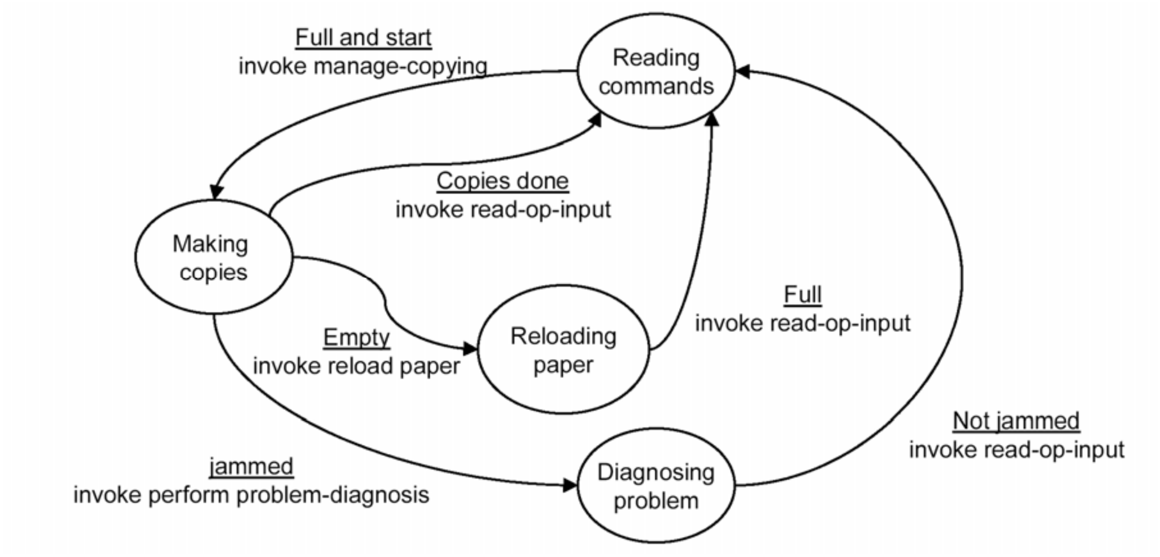
   Specifically, a transformation is a series of processing steps that are governed by a set of semantic constraints (Pressman calls them semantic statements – they could equally be called semantic predicates). For example, a search algorithm (i.e. a transformation taking a list as input and producing an position locator as output) would have 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.

   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

3. The **control** dimension. This is measured by counting the number of transitions between states. For this, you need a state transition diagram for the system or module being analysed. For example, the following state transition diagram has 6 state transitions.



Finally, you compute the 3D FP index by completing the following table:

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	=	

Complexity Weighting

The 3D Function Point index is equal to the sum of the sub-totals.

Reconciling Difference Metrics Approaches

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

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.

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.

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.

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*.

We will return to metrics for software quality later in the course (please refer also the Chapter 4 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman).

3. Software Project Planning – Resources, Estimation, Decomposition, and the COCOMO Estimation Model[7]

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

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

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 Scope

Scope addresses:

- 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 – interations with the user and other systems
- Reliability – quantitative requirements for functional performance (mean time between failures, acceptable error rates, etc.)

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?

[7] Please refer also the Chapter 4 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman.

- 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?
- 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?

These questions will break the ice and prepare for more problem-specific and project-specific meetings.

The bottom line is that you need to identify completely the data, function, and behaviour of the system, its constraints, and its performance requirements.

In order to do this properly, you will inevitably have to engage in a process of decomposition, sub-dividing the overall system into functional sub-units.

Resources

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

1. Description of the resource
2. Statement of availability
3. Time at which the resource will be required
4. Duration for which the resource will be required

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.

Environmental Resources

This refers to the software engineering environment. Note that this refers to more than the development tools such as compilers, linkers, libraries, development computers. It 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 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).

Generally, we use one or both of two approaches:

1. Decomposition Techniques
2. Empirical Estimation Techniques

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.

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.

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.

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.

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)

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**

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

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**

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

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

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).

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

$$E = 5.2 \times (KLOC)^{0.91}$$
$$E = 5.5 + 0.73 \times (KLOC)^{1.16}$$
$$E = 3.2 \times (KLOC)^{1.05}$$
$$E = 5.288 \times (KLOC)^{1.047}$$
$$E = -13.39 + 0.0545 \times FP$$
$$E = 585.7 + 15.12 \times FP$$

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.
Model 2:	The Intermediate COCOMO model which computes software development effort and cost as a function of program size and a sset 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.

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

Note that there are three different types of project:

1. *Organic*: relatively small simple software projects in which small teams with good application experience work to a set of less than rigid requirements.

2. *Semi-detached*: an intermediate sized project in which teams with mixed experience work to meet a mix of rigid and less than rigid requirements.

3. *Embedded*: a software project that must be developed within a set of tight hardware, software, and operational constraints.

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

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

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

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	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

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:

1. Product attributes
2. Hardware attributes
3. Personnel attributes
4. 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.

The Software Equation

The software equation is a multivariable model that assumes a specific distribution of effort over the life of a software development project. The model is based on productivity data from over 4000 software engineering projects.

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

Where:

E is the effort applied in person-years (**NB not person-months**)

t is the project duration in years (**NB not months**)

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 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*.

4. Project Scheduling and Tracking: Human Resources and Effort, Task Definition, Task Networks, Schedules [8]

Project Scheduling

Software development projects are very often delivered late. Why? There are several possible reasons:

- 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*

Note: 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 critical (i.e. on the critical path[9]),
- 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.

Basic principles of software project scheduling

Compartmentalization: Decompose the product and the process into a set of manageable activities/tasks.

Interdependency: Determine the interdependency between each component task (this then determines how tasks are ordered and what tasks can be undertaken in parallel).

Time Allocation: Allocate an estimate of effort to each task. Allocate a start date and an end date.

Effort Validation: make sure the amount of allocated effort does not exceed the available effort (globally and at any point in time).

Defined Responsibilities: Assign each task to a specific member of the team.

Defined Outcomes: Define the outcome (output) for each task – code, documentation, presentation, reports, etc. These are often referred to as deliverables.

Defined Milestones: Set milestones (checkpoints when a group of tasks is complete and the collective outcomes have been reviewed for quality).

[8] Please refer also the Chapter 7 of ‘Software Engineering – A Practitioner’s Approach’, R. S. Pressman.

[9] 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.

The Relationship between People and Effort

There is a common myth: "if we fall behind schedule, we can always add more staff and catch up later in the project". Unfortunately, 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. In addition, the communication overhead and complexity increases.)

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

"Adding man-power to a late software project makes it later".

In essence, 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 10000 lines of code. In this case, P = 10000 (telecommunications and systems software) and B = 0.16.

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

On the other hand, if the time available for the project is extended to 9 months, then the estimated effort is approximately equal to 0.5 person-years, i.e. a one-person team. Even if this estimate is debatable, it does show the highly non-linear relationship between the time available, the number of people to be deployed, and the effort required.

Distribution of Effort

Once we have an estimate of the total effort required for a project, we then need to distribute it across 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.

Note that it is often recommended that the specification, analysis, and design phase account for **more** than 40%.

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 (explore new business application or technology)
2. *New application development* projects (usually the result of a client request)
3. *Application enhancement* projects (major modification of existing systems, typically providing extended functionality)
4. *Application maintenance* projects (correct, adapt, extend existing software in a way that is not immediately obvious to the user).
5. *Reengineering* projects (rebuilding a legacy system)

You also need to decide on the degree of rigor with which the software development process will be applied. There are four different levels of rigor:

1. Casual (all process framework activities but umbrella tasks and documentation minimized).
2. Structured (all process framework activities, with significant software quality assurance)
3. Strict (all process framework activities; all umbrella activities, robust documentation)
4. Quick Reaction (apply the process framework but only focus on those task absolutely necessary to attain a good quality outcome).

Adaptation criteria are used to determine the recommended degree of rigor to apply. There are 11 adaptation criteria:

1. Size of the project
2. Number of potential users
3. Mission criticality
4. Application longevity
5. Stability of requirements
6. Ease of customer-developer communication
7. Maturity of application technology
8. Performance constraints
9. Embedded/non-embedded characteristics
10. Project staffing
11. Re-engineering factors

Each criterion is assigned a grade (1-5): 1 is a project in which a small set of process tasks need to be applied and the overall methodological and documentation requirements are minimal. 5 is a project all process tasks have to be applied in full with strong requirements for methodology and documentation.

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

Adaptation Criteria	**Grade**	**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)								

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.

Selecting Software Engineering Tasks

In order to develop a project schedule, a task set must be distributed on the project time line.

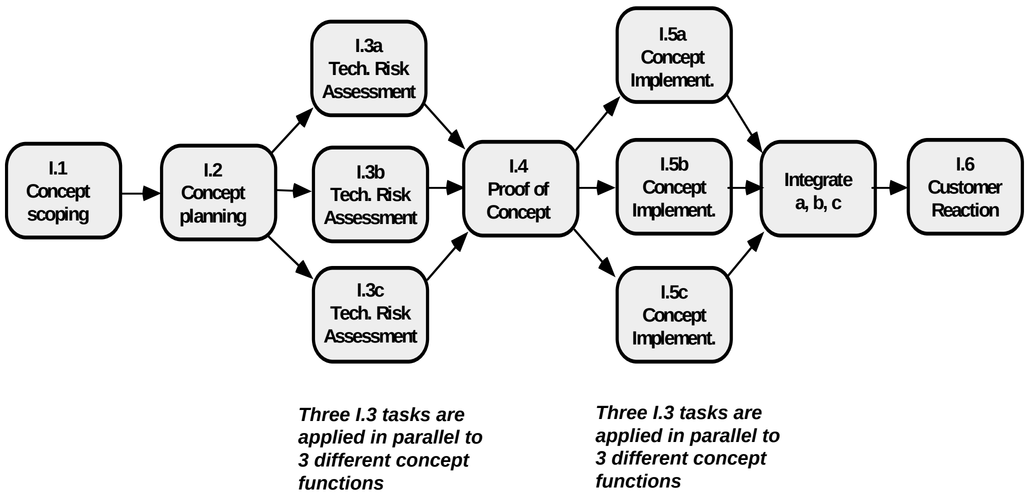
The task set will vary according to the type of project and the degree of rigor. Depending on the project time, you will choose an appropriate process model (e.g. waterfall, evolutionary, spiral). This process is forms the basis for a macroscopic schedule for a project.

Once this is complete, you must then proceed to refine each of the major tasks, breaking them into smaller composite tasks (remember: each sub-task also needs to have input and outcomes specified). This process of task decomposition is exactly the same one as we use when performing a function decomposition of the product during the specification phase of the software life-cycle, *i.e.* you should 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).

Note well that this process of task identification and decomposition will be greatly aided if you have a developed a rigorous and complete system specification (including all the supporting data, functional, and process models: entity relationship diagrams, functional decomposition diagrams [equivalently, the software architecture], data-flow diagrams, state-transition diagrams, etc.). If these haven't yet been created, then the schedule must be revisited and revised when they have.

Creating a Task Network

Once this is complete, the next step is to create 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. In addition, it allows the project manager to identify the critical path: the tasks that must be completed on schedule in order to avoid any delay in the overall project schedule.


I.1 Concept scoping
I.2 Concept planning
I.3a Tech. Risk Assessment
I.3b Tech. Risk Assessment
I.3c Tech. Risk Assessment
I.4 Proof of Concept
I.5a Concept Implement.
I.5b Concept Implement.
I.5c Concept Implement.
Integrate a, b, c
I.6 Customer Reaction
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

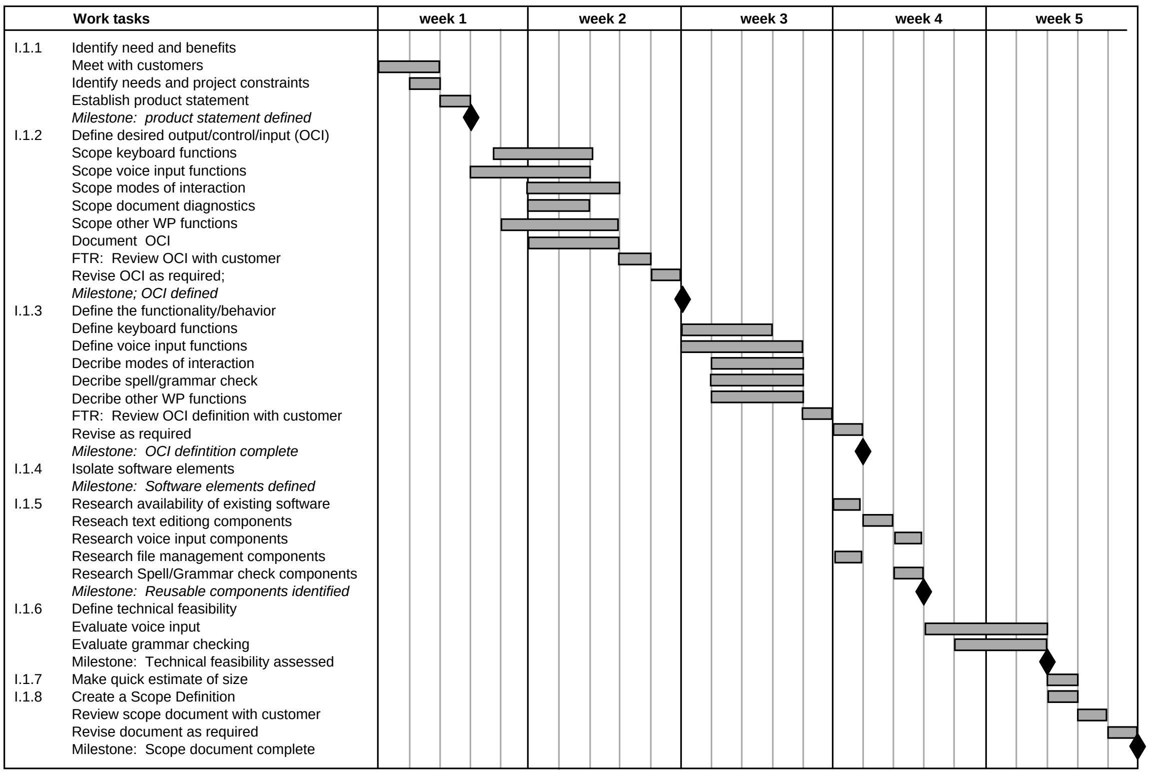
Create the Project Schedule

The final step is to create the project schedule (remembering always that it is a dynamic living entity that must be monitored, amended, revised, and validated),

All of the information for the schedule now exists at this point: the identity and purpose of each task, the relative sequencing of each task, the estimates of effort and time for each task.

Typically, you will use a standard tool to effect this schedule for you. One such approach is the Program evaluation and review technique (PERT). A PERT chart 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. The critical path is important as it defines the absolute minimum time needed to complete the project. Note that the critical path can change as task start-dates and durations are changed.

Another standared tool is the timeline chart or GANTT chart. This represents tasks by horizontal bars and lines are used to show the dependencies.



5. Software quality assurance (SQA): quality attributes, software reviews, statistical quality assurance, software quality standards[10]

Software Quality Assurance

Software Quality Assurance (SQA) is an activity that is applied throughout the software process. SQA encompasses:

- A quality management approach
- Effective engineering technology (methods and tools)
- Formal technical reviews
- Multi-layered testing strategy
- Control of software documentations
- Procedures for assuring compliance with software development standards
- Measurement and reporting mechanisms

Quality control is the central issue in quality assurance: the use of inspections, reviews, and tests (used throughout the development cycle) to ensure that each work product (module, function, document) meets its requirements. Quality control implies a feedback loop to correct and improve the process that lead to the work product; 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.

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.

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 – lack of conformance to requirements is lack of quality.

- 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.).

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.

[10] Please refer also the Chapter 8 of ‘Software Engineering – A Practitioner’s Approach’, R. S. Pressman.

Software Reviews

Software reviews are used to 'filter out' errors at various stages of the development process. There are many types of reviews, including informal discussions, customer presentations, and *formal technical reviews FTR* (sometimes referred to as a walkthrough). It is a software quality assurance activity that is 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.

Every review meeting should adhere to the following guidelines:

- Between three and five people should be involved (typically, the producer of the work, the project leader, the review leader, two or three reviewers, and a 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.

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. Typically, two documents will be produced:

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).

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

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%

Clearly, this is a feedback process.

The following is a list of typical sources of errors and 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)

The goal is to build a table of data detailing the number and percentage of defects that can be traced to each of these sources and 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								

In addition, one can compute the error index (*EI*) as follows:

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

Where i is the number of the current phase in the software process (requirements, specification, …), PS is the size of the product (e.g. in kLOC), and PI_i is the phase index (a weighted metric of the number of serious, moderate, and minor errors uncovered at phase i):

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

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

S_i is the number of serious errors

M_i is the number of moderate errors

T_i is the number of minor errors

W_s = 10, typically

W_m = 3, typically

W_t = 1, typically

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

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’.

The Quality System

Software development organizations should have a *quality system* (or *quality management 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.

The ISO 9001 Quality Standard

The ISO 9001 is an international standard that has been adopted by more than 130 countries. It is becoming increasingly important as a way by which clients can judge (or be assured of) the competence of a software developer.

One of the problems with ISO 9001 is that it is not industry-specific – it can be used by any type of industry.

For the software industry, the relevant standard are:

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

  This is a standard which 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.*

  This is a specific document which 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.

The requirement of the standard are grouped under 20 headings:

1. Management responsibility
2. Quality system
3. Contract review
4. Design control
5. Document control
6. Purchasing
7. Purchase supplied product
8. Product identification and traceability
9. Process control
10. Inspection and testing
11. Inspection, measuring and test equipment
12. Inspection and test status
13. Control of non-conforming product
14. Corrective action
15. Handling, storage, packaging and delivery
16. Quality records
17. Internal quality audits
18. Training
19. Servicing
20. Statistical techniques

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.

Note the generality of this statement: it could apply to the developer of any product. Consequently, it can be hard to interpret in a specific (software) domain. It is 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.

6. McCall's Quality Factors, FURPS, and Technical Metrics for Analysis, Design, Implementation, Testing, and Maintenance[11]

Introduction

So far in this course, we have been concerned with metrics that are applied at the process and project level. We now shift our focus to measures that can be used to assess the quality of the software as it is being developed to allow the software engineer to assess, control, and improve his development process.

We will begin by introducing two software quality checklists that explicitly highlight the factors that influence quality and their relationships to software quality metrics.

Recall the issues associated with quality identified in the previous section:

- Software requirements are the foundation from which quality is measured – lack of conformance to requirements is lack of quality.

- Specified standards define a set of development criteria that guide the manner in which software is engineered; if they are not followed, low quality software will probably be produced.

- There is always a set of implicit requirements (the non-functional attributes, such as dependability, etc.); if these are not addressed, low quality software will probably be produced.

McCall's Quality Factors

McCall and his colleagues proposed a categorization of factors that affect software quality. These 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*)

The software quality factors are:

- Product Operation
    - Correctness
    - Reliability
    - Efficiency
    - Integrity (security)
    - Usability
- Product Revision
    - Maintainability
    - Flexibility
    - Testability
- Product transition
    - Portability
    - Reusability
    - Interoperatibility

It is very difficult (and often impossible) to develop direct measures of these quality factors. McCall proposes the *combination* of several metrics to provide these measures.

[11] Please refer also the Chapter 18 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman.

The metrics he proposes are:

1. *Auditability* (ease with which conformance with standards can be checked)
2. *Accuracy* (precision of computations)
3. *Communication commonality* (use of standard interfaces and protocols)
4. *Completeness* (w.r.t. requirements)
5. *Conciseness* (compact code)
6. *Consistency* (uniformity of design and documentation)
7. *Data commonality* (use of standard data structures and types)
8. *Error tolerance* (cf graceful degradation)
9. *Execution efficiency* (run-time performance)
10. *Expandability* (cf reuse)
11. *Generality* (number of potential applications)
12. *Hardware independence*
13. *Instrumentation* (degree to which the program monitors its own operation and identifies errors, cf autonomic computing)
14. *Modularity* (functional independence of program components)
15. *Operability* (ease of use)
16. *Security*
17. *Self-documentation* (usefulness/clarity of source code and internal documentation)
18. *Simplicity* (ease of understanding of source code / algorithms)
19. *Software system independence* (decoupling from operating system and libraries esp. DLLs)
20. *Traceability* (ability to trace a design, representation, or code to initial requirements)
21. *Training* (ease of learning by new users)

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.

The metrics that are used for each software quality factor are identified from a checklist:

	Software Quality Factors										
Software Quality Metric	Correctness	Reliability	Efficiency	Integrity	Maintainability	Flexibility	Testability	Portablity	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

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 for Analysis, Design, Implementation, Testing, and Maintenance

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.

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

Completeness of functional requirements

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

n_u is the number of unique function requirements

n_i is the number of inputs

n_s is the number of states

Metrics for the Design Model

Structural Complexity of a module i

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

$f_{out}(i)$ is fan-out of module i (*i.e.* the number of modules that are called by module i)

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

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

Morphology Metrics (based on program architecture / functional decomposition graph)

- size = $n + a$

  n is the number of nodes in the tree (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

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).

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
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.

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.

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.

What is the cohesion of a module that produces only one output?

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).

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.

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)

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.

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

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

This is the highest value of a module coupling indicator (why?)

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 0.66 to a maximum value that approaches 1.0.

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

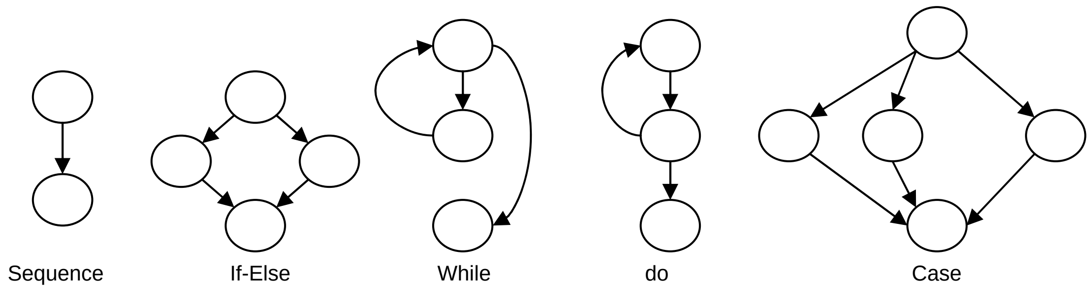
The most widely used complexity metric is the *cyclomatic complexity*, sometimes referred to as the McCabe metric (after its developer, Thomas McCabe).

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

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.

If we represent a program as a flow graph (i.e. a graph with statements represented as nodes and flow of control represented as edges; areas bounded by edges and nodes are called regions, with the 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.



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

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

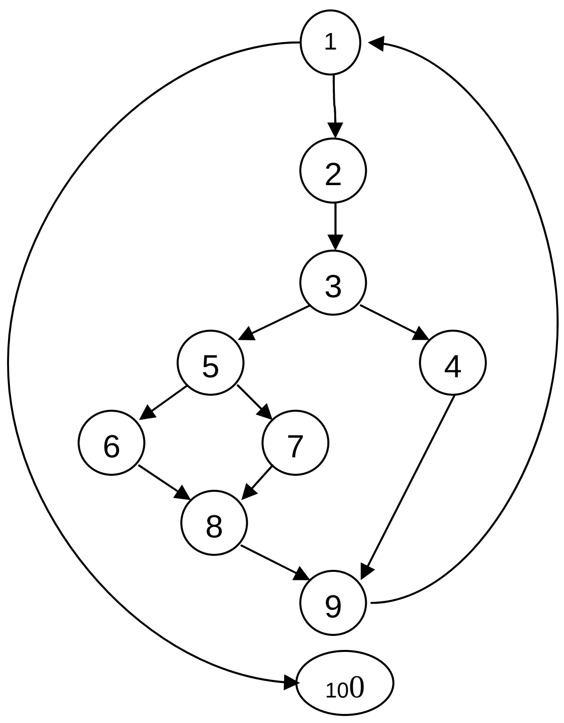
  $V(G) = E - N + 2$

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

- 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.



```
// 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
```

For example, the cyclomatic complexity of the above program is 4:

4 = 12 – 10 + 2
4 = 3 + 1

Interface Design Metrics

Graphic user interfaces, GUIs, require the placement of widgets (buttons, dialogue boxes, scroll bars, etc) on a screen. The layout appropriateness metric, LA, is one way of assessing the effectiveness of the interface (or, more specifically, the spatial organization of the GUI; effectiveness is best measured by user feedback).

Define the cost C of performing a series of actions with a given layout as the sum of the frequency of transition between a pair of widgets times the cost of that transition (e.g. the distance a cursor must travel to accomplish the transition). The summation is effected for all transitions required for a particular task.

$$C = \Sigma\ (f(k)\ /\ m(k))$$

where

$f(k)$ is the frequency of a transition k
$m(k)$ is the cost of a transition k
The summation is made for all k, i.e. all transitions.

The LA metric is then defined as follows:

$$LA = 100 \times C_o\ /\ C_i$$

where

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

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 $[\ N!\ /\ K! \times (N\text{-}K)!\ ] \times K!$

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).

Metrics for Source Code

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

Define the following measures.

n_1 be the number of distinct operators that appear in a given program (including the language constructs such as for, while, do, case, …).
n_2 be the number of distinct operands that appear in a given program
N_1 be 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

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:

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

Metrics for Testing

Much has been written about software metrics for testing. However, most of this material 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.

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.

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

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

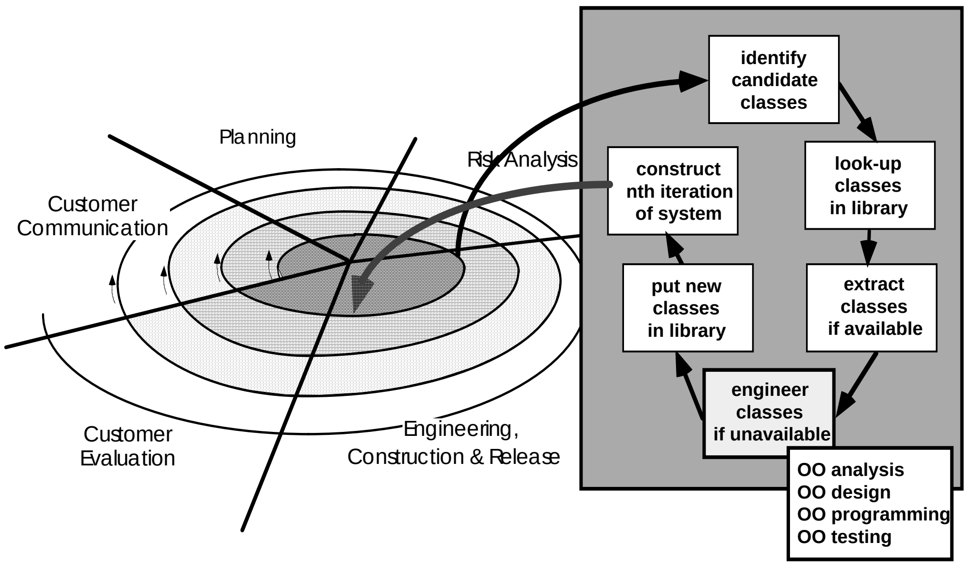
F_a = the number of modules in the current release that have been added

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

7. OO classes, inheritance, and polymorphism [12]

Object-oriented technologies (sometimes referred to as just object technologies) involve more that just object-oriented programming (using OO languages such as C++, Eiffel, Smalltalk). Object technologies also include the analysis, design, and testing phases of the development life-cycle.

It is extremely difficult to define all the necessary classes for a major system or product in a single iteration. Consequently, 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.


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

What is an object-oriented approach? There are many definitions of OO.

One definition is that it is the exploitation of class objects, with private data members and associated access functions (cf. concept of an abstract data type). This is a good start and introduces some key concepts:

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

[12] Please refer also the Chapter 19 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman.

Data Members	The object will have **data members** representing the class attributes (e.g. int x, y;)
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.
Encapsulation	The object (and class) **encapsulates** the data members (attributes), methods (access functions) in one logical entity.
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**.
Abstraction	This separation, though data hiding, of physical implementation from logical access is called **abstraction**.
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).

So far, so good. All this sounds very familiar and no different to what was covered in abstract data-types in the course on Algorithms and Data-Structures. However, there is a stronger meaning associated with OO. Ellis and Stroustrup, for example, define OO as follows:

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

The implication of this statement is that OO also includes two further concepts of **inheritance** and **polymorphism**.

A complete treatment of these concepts would require specific reference to an OO programming language and its constructs. However, a simple definition is still required.

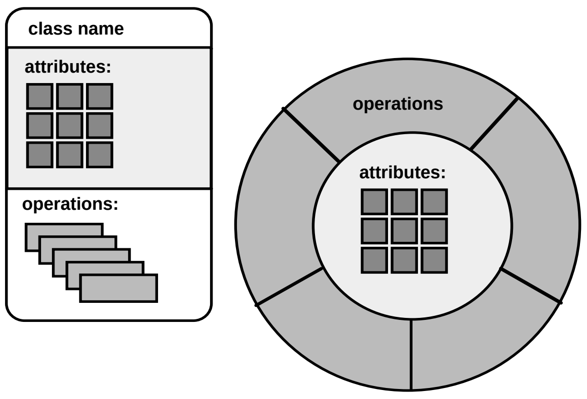
Inheritance	We can define a new class as a **sub-class** of an existing class (e.g. button is a sub-class of the widget class; a toggle button is a sub-class of the button class). Each sub-class **inherits** (has by default) the data members and methods of the parent class (the parent class is sometimes 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.

Polymorphism

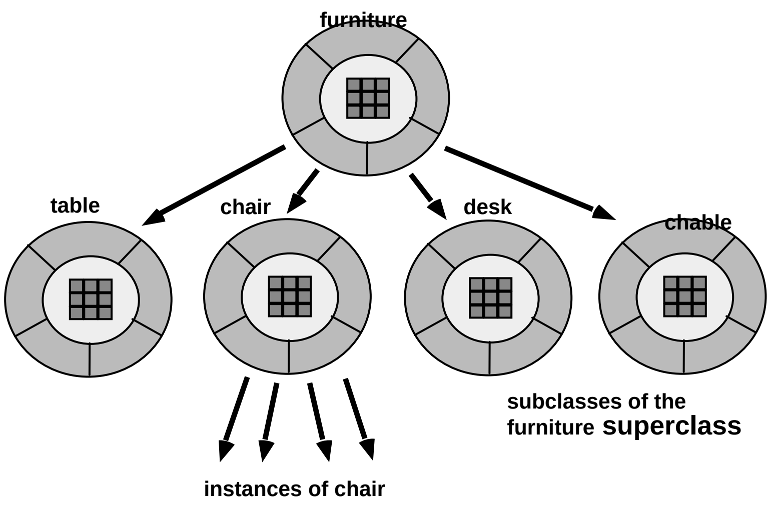
If the super-class is designed appropriately, it is possible to re-define the meaning of some of the super-class methods to suit the particular needs of the derived class. For example, the widget super-class might have a draw() method. Clearly, we need a different type of drawing for buttons and for 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).

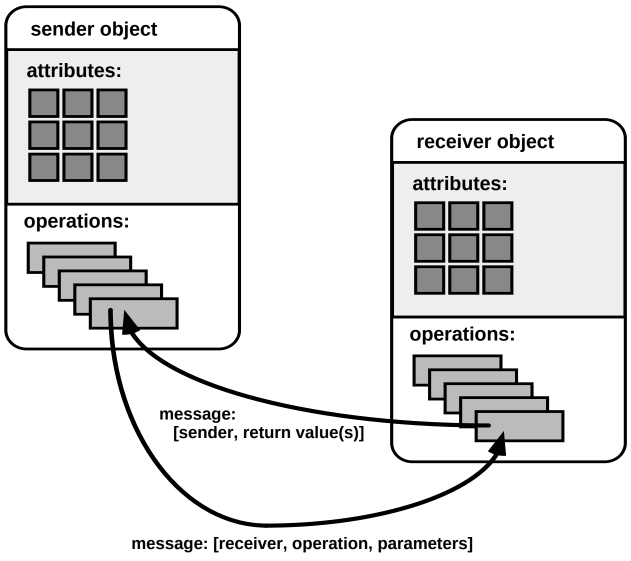
OO programming languages allow us to do this. This is called **polymorphism** (literally: multiple structure) – the ability to define and choose the required method depending on the type of the derived class (or object) without changing the name of the method. Thus, the draw() method has many structures, one for each derived class.



Two views of a class / object



A Class Hierarchy


sender object
attributes:
operations:
receiver object
attributes:
operations:
message: [sender, return value(s)]
message: [receiver, operation, parameters]

Message Passing Between Objects

8. Object-oriented analysis, design, and testing: OOD concepts, analysis and design issues [13]

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.

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.

There are many OOA methods. For exampe:

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

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

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

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

[13] Please refer also the Chapters 19-21 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman.

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

We will look at each of these in turn.

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 or devices 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).

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

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

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).

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'.

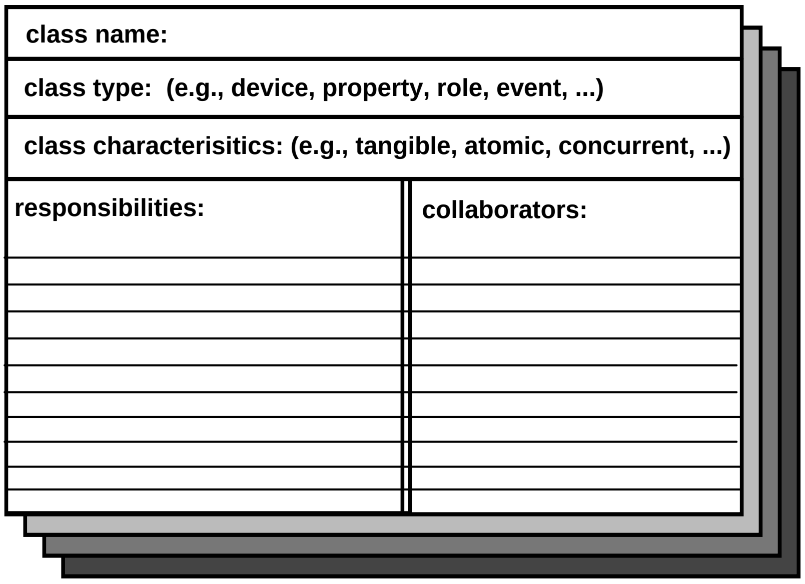
We can classify different types of entity and this will help identify the associated classes:

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).

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)?
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.


class name:
class type: (e.g., device, property, role, event, ...)
class characterisitics: (e.g., tangible, atomic, concurrent, ...)
responsibilities:
collaborators:

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.

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.

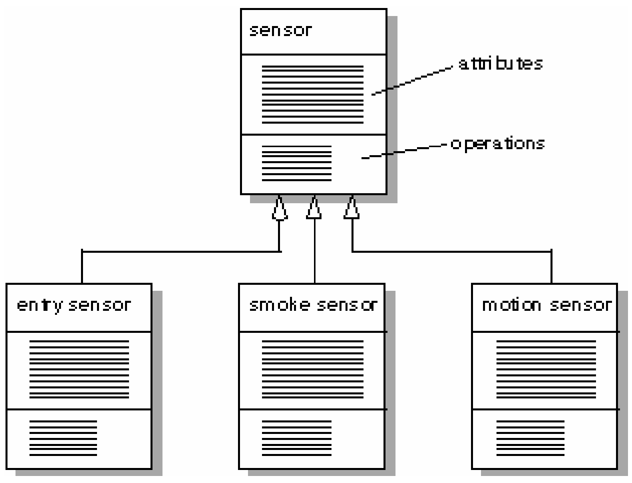
Defining Structures and Hierarchies

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

There are two types of hierarchy:

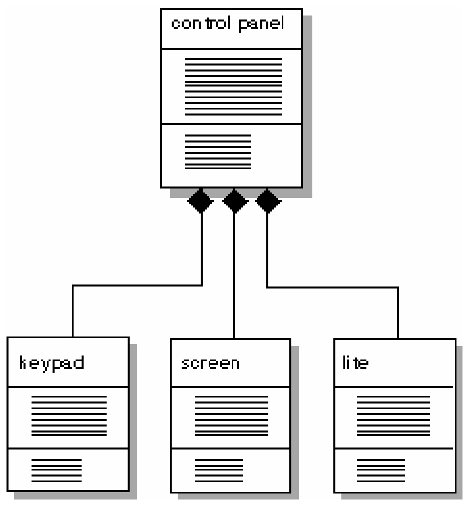
1. Generalization-Specialization (Gen-Spec) structure
2. Composite-Aggregate (Whole-Part) structure

The Gen-Spec structure can be represented by the following type of diagram:



The relationship between classes in a Gen-Spec hierarchy can be viewed as a ‘I**s A**’ relation. For example, an entry sensor ‘Is A’ sensor. This indicates a relationship by inheritance (entry sensor is a sub-class of sensor and inherits the base class methods and members).

The Composite-Aggregate structure can be represented by the following type of diagram



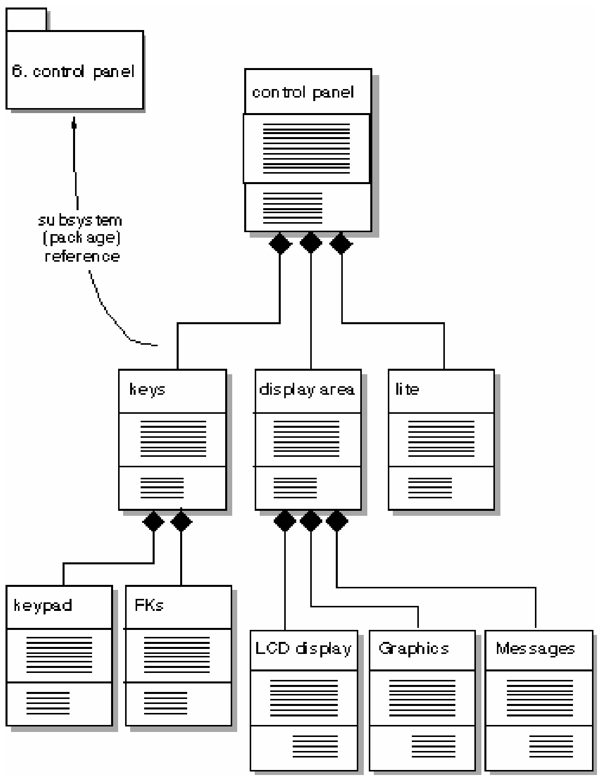
The relationship between classes in a Composite-Aggregate hierarchy can be viewed as a ‘**Has A’** relation. For example, a control panel ‘Has A’ key-pad. This indicates a relationship by nesting (key-pad is a component class of control panel).

Note that there are alternative styles of diagrams for both types of diagram.

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.

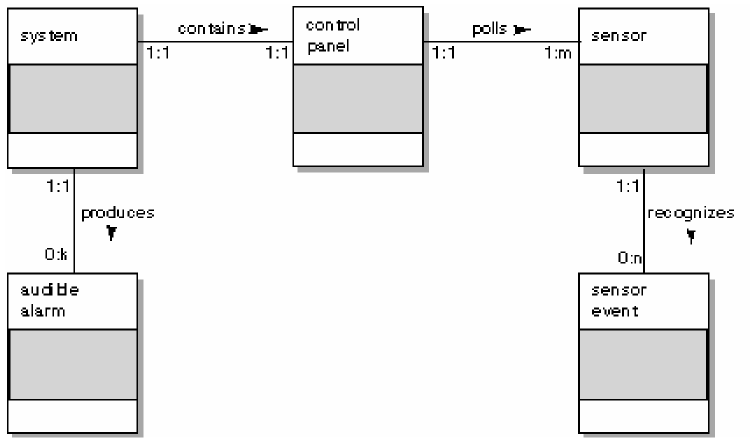


The Object-Relationship Model

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.



The object-relationship model makes explicit the message paths between classes.

The Object-Behaviour Model

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

To create the model, 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.

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.

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

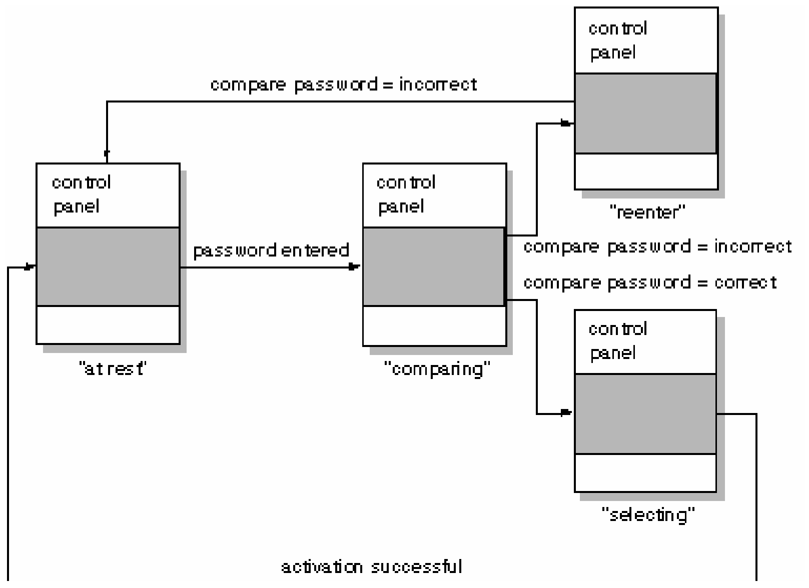
- The state of each object as the system performs its function.
- The state of the system as observed from outside as the system performs its function.

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.

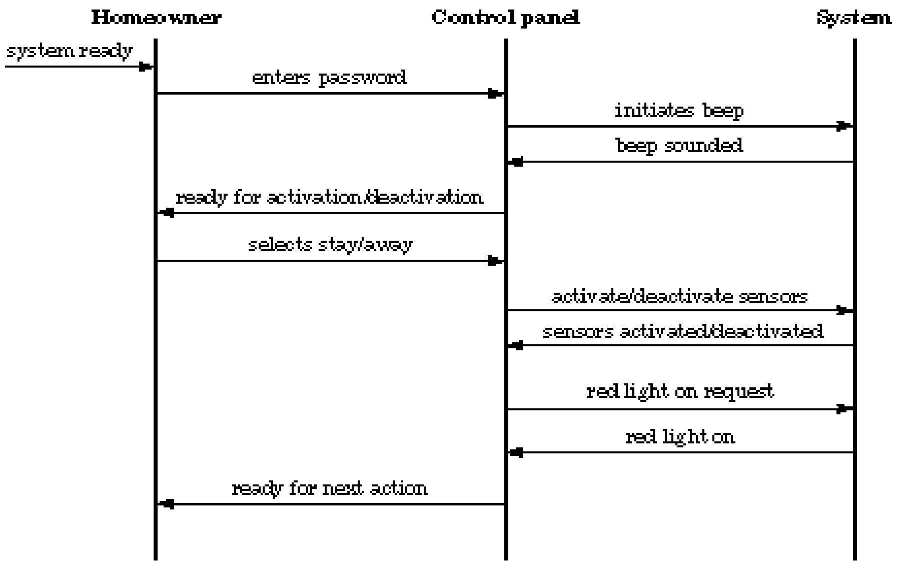
An event (*i.e.* a trigger) must occur to force an object to make a transition from one active state to another.

One component of an object-behaviour model is a simple representation of the active states for each object and the events (triggers) that cause changes between active states.



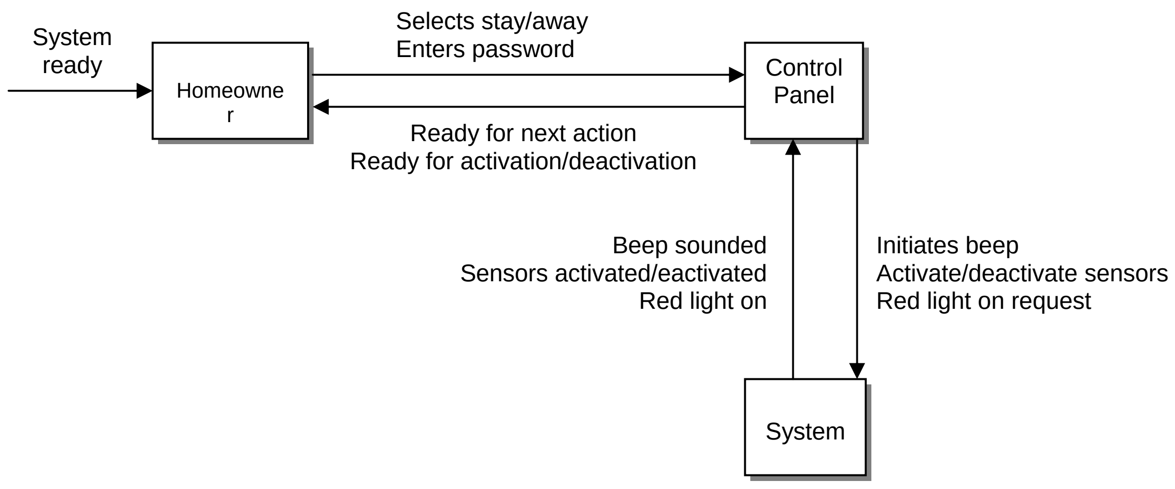
A partial active state transition diagram for the object *control panel*

The second type of behavioural representation for OOA considers a state representation for the overall product or system. This representation is an *event trace* model that indicates how events cause transitions from object to object and a state-transition diagram that depicts the processing behaviour of each object. An event trace is actually a shorthand version of the use case.



A partial event trace

Once a complete event trace has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This can be represented using an event flow diagram.



A partial event flow diagram

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.

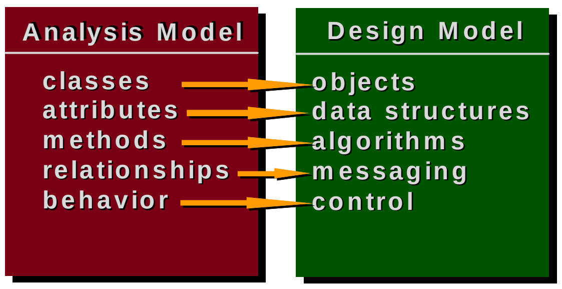
There are many OOD approaches, almost all of which are the direct adjuncts of OOA approaches (e.g. the Booch method, the Coad and Yourdon Method, the Jacobson method, the Rambaugh method). The following gives just an overview of the issues that are common to all approaches.

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.

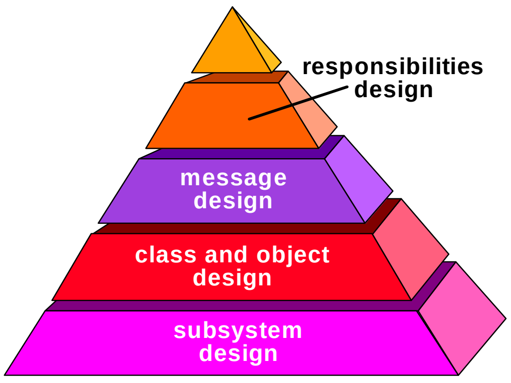
Thus, the main goal in OOD is to make the OOA models less abstract by filling in all the details, but without going as far as writing the OO code. This will require you to state exactly:

- 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).

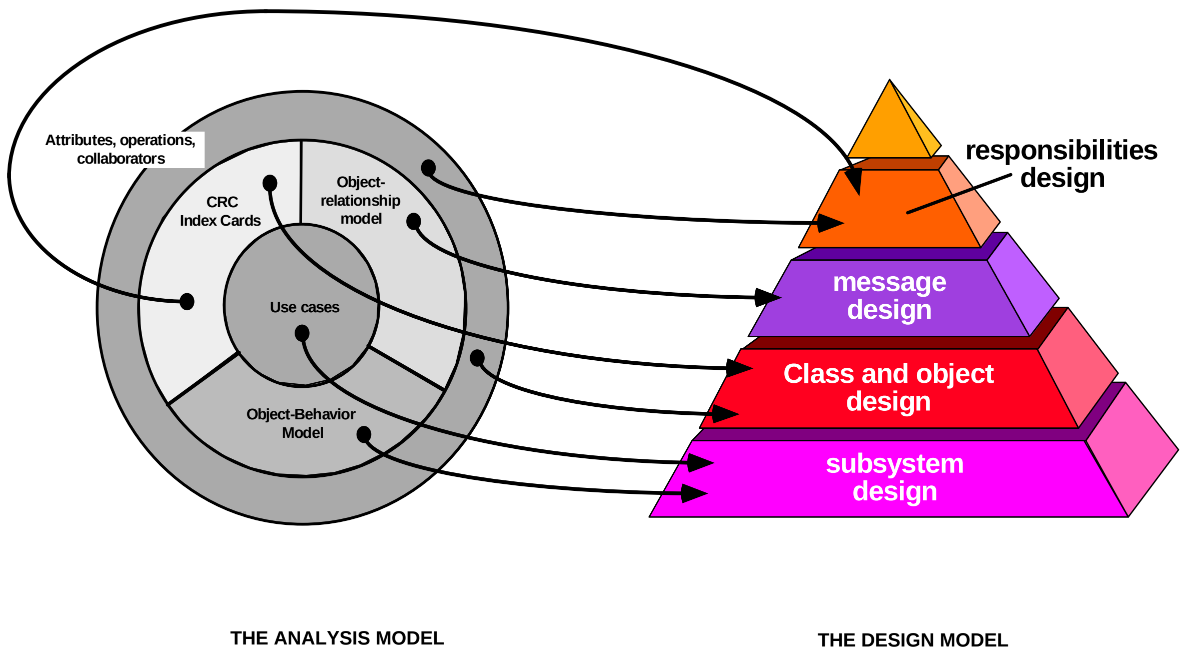
In essence, developing the design based on the analysis model, we are focusing on the algorithmic, representation, and interface issues; on *how* the system will be implemented, rather than on *what* will be implemented. When we write the OO code, we then realize that design using a specific language.



There are several level to an OO design. These are often represented in a pyramidal manner (just as in the case of the design of imperative programs).



And we can visualize the mapping of the OOA models to the OOD, as follows:



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.

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

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

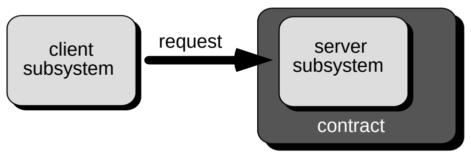
2. Human interaction subsystems: the subsystems that implement the user-interface (incorporating reusable GUI class libraries).

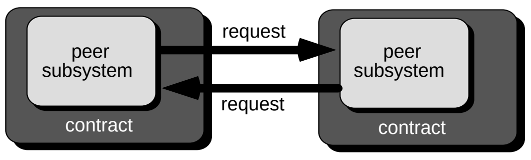
3. Task management subsystems: the subsystems that are responsible for controlling and coordinating concurrent tasks.

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

Note that, when two subsystems communicate with one another, they can establish a client/server link or a peer-to-peer link.

In a client/server link, each of the subsystems takes on one of the roles implied by the client and server. Service flows from server to client in only one direction. In a peer-to-peer link, services may flow in either direction.





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.

In addition to designing the organization of (sets of) classes and objects, we also have to design individual objects and classes. Typically, the design will incorporate two distinct aspects: a protocol description and an implementation description.

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

9. OO testing strategies, metrics [14]

Object-Oriented Testing (OOT)

Recall that the goal of testing is to find the greatest number of errors (i.e. the focus should be to try to break the system, not just demonstrate that it produces the right answer in certain circumstances).

However, we need to broaden our view of testing in OO to include not only conventional testing of code/programs but the review of OO analysis and design models. Why? Because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Thus, in OOT

- we begin by evaluating the correctness and consistency of the OOA and OOD models;
- we recognize that the testing strategy changes
  - o the concept of the 'unit' broadens due to encapsulation
  - o integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
  - o 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

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.

[14] Please refer also the Chapters 19, 22-23 of 'Software Engineering – A Practitioner's Approach', R. S. Pressman.

OOT Strategy

OO brings particular complexities to the testing process. In particular, encapsulation and inheritance make testing more complicated.

In the case of encapsulation, the data members are effectively hidden and the test strategy needs to exercise both the access methods and the hidden data-structures.

In the case of 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.

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:

    1. operations within the class are tested

    2. the state behavior of the class is examined (very often, the state of an object is persistent).

- integration testing requires three different strategies:

    1. thread-based testing—integrates the set of classes required to respond to one input or event (incremental integration may not be possible).

    2. use-based testing—integrates the set of classes required to respond to one use case

    3. cluster testing—integrates the set of classes required to demonstrate one collaboration

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.

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)

Partition Testing

- 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

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

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.

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).

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.

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).

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

Technical Metrics

Technical metrics are needed that highlight the special features of OO:

- Localization (data cohesion)
- Encapsulation
- Information Hiding
- Inheritance
- Abstraction

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 (consider those defined in the derived class vs those defined in base class)

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

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).

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.

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’.

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.

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.

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.

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++).

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.

Specialization Index (SI)

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

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.

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.

Test Metrics

Encapsulation

Lack of Cohesion in Methods (LCOM)

(see above)

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.

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)

See above.

10. Social, ethical, and professional issues: code of ethics, copyright, & security

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.

IEEE Code of Ethics

(Taken from: http://www.ieee.org/about/whatis/code.html)

We, the members of the IEEE, in recognition of the importance of our technologies in affecting the quality of life throughout the world, and in accepting a personal obligation to our profession, its members and the communities we serve, do hereby commit ourselves to the highest ethical and professional conduct and agree:

1. 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;
2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;
3. to be honest and realistic in stating claims or estimates based on available data;
4. to reject bribery in all its forms;
5. to improve the understanding of technology, its appropriate application, and potential consequences;
6. 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;
7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;
8. to treat fairly all persons regardless of such factors as race, religion, gender, disability, age, or national origin;
9. to avoid injuring others, their property, reputation, or employment by false or malicious action;
10. to assist colleagues and co-workers in their professional development and to support them in following this code of ethics.

Approved by the IEEE Board of Directors
August 1990

Software Engineering Code of Ethics and Professional Practice

(IEEE Computer Society and Association for Computer Machinery code of ethics for software engineers: http://computer.org/certification/ethics.htm)

"The time is right to get serious about this. As software becomes increasingly dominant in the IT industry, and, indeed, in everything else, there is an obvious need for a professional-level recognition. Far too much is placed on particular credentials for specific products or applications without regard to the bigger picture. The result is poorly engineered software projects."

(Version 5.2) as recommended by the IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices and Jointly approved by the ACM and the IEEE-CS as the standard for teaching and practicing software engineering.

Short Version

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

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:

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.

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.

SOFTWARE ENGINEERING CODE OF ETHICS AND PROFESSIONAL PRACTICE

IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices

Full Version

PREAMBLE

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.

The Code contains eight Principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer├s humanity, in special care owed to people affected by the work

of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer.

It is not intended that the individual parts of the Code be used in isolation to justify errors of omission or commission. The list of Principles and Clauses is not exhaustive. The Clauses should not be read as separating the acceptable from the unacceptable in professional conduct in all practical situations. The Code is not a simple ethical algorithm that generates ethical decisions. In some situations standards may be in tension with each other or with standards from other sources. These situations require the software engineer to use ethical judgment to act in a manner which is most consistent with the spirit of the Code of Ethics and Professional Practice, given the circumstances.

Ethical tensions can best be addressed by thoughtful consideration of fundamental principles, rather than blind reliance on detailed regulations. These Principles should influence software engineers to consider broadly who is affected by their work; to examine if they and their colleagues are treating other human beings with due respect; to consider how the public, if reasonably well informed, would view their decisions; to analyze how the least empowered will be affected by their decisions; and to consider whether their acts would be judged worthy of the ideal professional working as a software engineer. In all these judgments concern for the health, safety and welfare of the public is primary; that is, the "Public Interest" is central to this Code.

The dynamic and demanding context of software engineering requires a code that is adaptable and relevant to new situations as they occur. However, even in this generality, the Code provides support for software engineers and managers of software engineers who need to take positive action in a specific case by documenting the ethical stance of the profession. The Code provides an ethical foundation to which individuals within teams and the team as a whole can appeal. The Code helps to define those actions that are ethically improper to request of a software engineer or teams of software engineers.

The Code is not simply for adjudicating the nature of questionable acts; it also has an important educational function. As this Code expresses the consensus of the profession on ethical issues, it is a means to educate both the public and aspiring professionals about the ethical obligations of all software engineers.

PRINCIPLES

Principle 1 PUBLIC

Software engineers shall act consistently with the public interest. In particular, software engineers shall, as appropriate:

1.01. Accept full responsibility for their own work.

1.02. Moderate the interests of the software engineer, the employer, the client and the users with the public good.

1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.

1.04. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.

1.05. Cooperate in efforts to address matters of grave public concern caused by software, its installation, maintenance, support or documentation.

1.06. Be fair and avoid deception in all statements, particularly public ones, concerning software or related documents, methods and tools.

1.07. Consider issues of physical disabilities, allocation of resources, economic disadvantage and other factors that can diminish access to the benefits of software.

1.08. Be encouraged to volunteer professional skills to good causes and contribute to public education concerning the discipline.

Principle 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. In particular, software engineers shall, as appropriate:

2.01. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.

2.02. Not knowingly use software that is obtained or retained either illegally or unethically.

2.03. Use the property of a client or employer only in ways properly authorized, and with the client's or employer's knowledge and consent.

2.04. Ensure that any document upon which they rely has been approved, when required, by someone authorized to approve it.

2.05. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

2.06. Identify, document, collect evidence and report to the client or the employer promptly if, in their opinion, a project is likely to fail, to prove too expensive, to violate intellectual property law, or otherwise to be problematic.

2.07. Identify, document, and report significant issues of social concern, of which they are aware, in software or related documents, to the employer or the client.

2.08. Accept no outside work detrimental to the work they perform for their primary employer.

2.09. Promote no interest adverse to their employer or client, unless a higher ethical concern is being compromised; in that case, inform the employer or another appropriate authority of the ethical concern.

Principle 3 PRODUCT

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. In particular, software engineers shall, as appropriate:

3.01. Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.

3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.

3.03. Identify, define and address ethical, economic, cultural, legal and environmental issues related to work projects.

3.04. Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.

3.05. Ensure an appropriate method is used for any project on which they work or propose to work.

3.06. Work to follow professional standards, when available, that are most appropriate for the task at hand, departing from these only when ethically or technically justified.

3.07. Strive to fully understand the specifications for software on which they work.

3.08. Ensure that specifications for software on which they work have been well documented, satisfy the users ├ requirements and have the appropriate approvals.

3.09. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates.

3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.

3.11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.

3.12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.

3.13. Be careful to use only accurate data derived by ethical and lawful means, and use it only in ways properly authorized.

3.14. Maintain the integrity of data, being sensitive to outdated or flawed occurrences.

3.15 Treat all forms of software maintenance with the same professionalism as new development.

Principle 4 JUDGMENT

Software engineers shall maintain integrity and independence in their professional judgment. In particular, software engineers shall, as appropriate:

4.01. Temper all technical judgments by the need to support and maintain human values.

4.02 Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.

4.03. Maintain professional objectivity with respect to any software or related documents they are asked to evaluate.

4.04. Not engage in deceptive financial practices such as bribery, double billing, or other improper financial practices.

4.05. Disclose to all concerned parties those conflicts of interest that cannot reasonably be avoided or escaped.

4.06. Refuse to participate, as members or advisors, in a private, governmental or professional body concerned with software related issues, in which they, their employers or their clients have undisclosed potential conflicts of interest.

Principle 5 MANAGEMENT

Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. In particular, those managing or leading software engineers shall, as appropriate:

5.01 Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.

5.02. Ensure that software engineers are informed of standards before being held to them.

5.03. Ensure that software engineers know the employer's policies and procedures for protecting passwords, files and information that is confidential to the employer or confidential to others.

5.04. Assign work only after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.

5.05. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work, and provide an uncertainty assessment of these estimates.

5.06. Attract potential software engineers only by full and accurate description of the conditions of employment.

5.07. Offer fair and just remuneration.

5.08. Not unjustly prevent someone from taking a position for which that person is suitably qualified.

5.09. Ensure that there is a fair agreement concerning ownership of any software, processes, research, writing, or other intellectual property to which a software engineer has contributed.

5.10. Provide for due process in hearing charges of violation of an employer's policy or of this Code.

5.11. Not ask a software engineer to do anything inconsistent with this Code.

5.12. Not punish anyone for expressing ethical concerns about a project.

Principle 6 PROFESSION

Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. In particular, software engineers shall, as appropriate:

6.01. Help develop an organizational environment favorable to acting ethically.

6.02. Promote public knowledge of software engineering.

6.03. Extend software engineering knowledge by appropriate participation in professional organizations, meetings and publications.

6.04. Support, as members of a profession, other software engineers striving to follow this Code.

6.05. Not promote their own interest at the expense of the profession, client or employer.

6.06. Obey all laws governing their work, unless, in exceptional circumstances, such compliance is inconsistent with the public interest.

6.07. Be accurate in stating the characteristics of software on which they work, avoiding not only false claims but also claims that might reasonably be supposed to be speculative, vacuous, deceptive, misleading, or doubtful.

6.08. Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work.

6.09. Ensure that clients, employers, and supervisors know of the software engineer's commitment to this Code of ethics, and the subsequent ramifications of such commitment.

6.10. Avoid associations with businesses and organizations which are in conflict with this code.

6.11. Recognize that violations of this Code are inconsistent with being a professional software engineer.

6.12. Express concerns to the people involved when significant violations of this Code are detected unless this is impossible, counter-productive, or dangerous.

6.13. Report significant violations of this Code to appropriate authorities when it is clear that consultation with people involved in these significant violations is impossible, counter-productive or dangerous.

Principle 7 COLLEAGUES

Software engineers shall be fair to and supportive of their colleagues. In particular, software engineers shall, as appropriate:

7.01. Encourage colleagues to adhere to this Code.

7.02. Assist colleagues in professional development.

7.03. Credit fully the work of others and refrain from taking undue credit.

7.04. Review the work of others in an objective, candid, and properly- documented way.

7.05. Give a fair hearing to the opinions, concerns, or complaints of a colleague.

7.06. Assist colleagues in being fully aware of current standard work practices including policies and procedures for protecting passwords, files and other confidential information, and security measures in general.

7.07. Not unfairly intervene in the career of any colleague; however, concern for the employer, the client or public interest may compel software engineers, in good faith, to question the competence of a colleague.

7.08. In situations outside of their own areas of competence, call upon the opinions of other professionals who have competence in that area.

Principle 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. In particular, software engineers shall continually endeavor to:

8.01. Further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.

8.02. Improve their ability to create safe, reliable, and useful quality software at reasonable cost and within a reasonable time.

8.03. Improve their ability to produce accurate, informative, and well-written documentation.

8.04. Improve their understanding of the software and related documents on which they work and of the environment in which they will be used.

8.05. Improve their knowledge of relevant standards and the law governing the software and related documents on which they work.

8.06 Improve their knowledge of this Code, its interpretation, and its application to their work.

8.07 Not give unfair treatment to anyone because of any irrelevant prejudices.

8.08. Not influence others to undertake any action that involves a breach of this Code.

8.09. Recognize that personal violations of this Code are inconsistent with being a professional software engineer.

This Code was developed by the IEEE-CS/ACM joint task force on Software Engineering Ethics and Professional Practices (SEEPP):

Executive Committee: Donald Gotterbarn (Chair),

Keith Miller and Simon Rogerson;

Members: *Steve Barber, Peter Barnes, Ilene Burnstein, Michael Davis, Amr El-Kadi, N. Ben Fairweather, Milton Fulghum, N. Jayaram, Tom Jewett, Mark Kanko, Ernie Kallman, Duncan Langford, Joyce Currie Little, Ed Mechler, Manuel J. Norman, Douglas Phillips, Peter Ron Prinzivalli, Patrick Sullivan, John Weckert, Vivian Weil, S. Weisband and Laurie Honour Werth.*

Case studies from the Online Ethics Center for Engineering and Science, Case Western Reserve University
(taken from http://www.onlineethics.org).

Infants Under Pressure
Sam Wilson, an experienced engineer was employed by MedTech, a company that made medical equipment. An important line of products were respirators, used in hospitals. A colleague of Sam asked him to check out one of these respirators, one designed for infant use. He soon determined that a relief valve intended to protect against overpressure being applied to the infant's lungs was incorrectly placed, so that, under certain circumstances, the infant could experience dangerously high pressure. Correcting the error would not be difficult, since all that was needed was to reposition the relief valve. In similar circumstances in the past, Sam had seen such problems handled with dispatch. He called the matter to the attention of the appropriate manager and assumed that it would be taken care of.

A month or so later (Sam was not directly involved with this particular device) he learned that nothing had been done. Hundreds of these devices were already in use, and Sam was concerned about the increasing likelihood of a tragic event. He went back to the manager and urged him to take appropriate action. When the manager fended him off, Sam said that if prompt measures were not taken to correct the problem he would have to report it to the cognizant regulatory agency. The response of MedTech was to fire Sam. Apparently the then current president of MedTech did not have the same attitude toward product quality that had been prevalent in the past.

At about the same time, the respirator problem was identified by a physician who had encountered one in hospital practice. Sam brought suit against MedTech for wrongful discharge, claiming that his actions in calling attention to the problem were mandated by the code of ethics that binds professional engineers. Sam is a licensed PE.

Various management changes have since occurred and the legal process is slowly moving along.

Using Other People's Software
Jim Warren was a senior software systems expert, hired by NewSoft, a start-up company, to help in the development of a new product. He soon learned that the product was based on proprietary software for which NewSoft did not have a license. Jim assumed that this was some sort of mistake and spoke to the company president about the matter. He was assured that the situation would be rectified. But time passed and nothing happened except that Jim found other instances of the same practice. Repeated efforts to get NewSoft to legalize its operations failed and Jim, after threatening to notify the victimized companies, was discharged.

Law enforcement officials were brought into the picture and lawyers on all sides began negotiating. At this date it is not clear whether criminal charges will be filed. There appears to be a strong possibility of some sort of out-of-court settlement among the companies involved. It is not clear how this will ultimately affect Jim Warren.

Not Lighting Up
Will Morgan, a licensed electrical engineer, worked for a state university on construction and renovation projects. His immediate manager was an architect, and next in the chain of command was an administrator, John Tight, a man with no technical background. Tight, without talking to the engineers, often produced estimates on project costs that he passed on to higher university officials. In those cases, not infrequent, where it became evident that actual costs were going to exceed his estimates, he would pressure the engineers to cut corners.

One such occasion involved the renovation of a warehouse to convert some storage space into office space. Among the specifications detailed by Morgan was the installation of emergency exit lights. These were mandated by the building code. As part of his effort to bring the actual costs closer to his unrealistic estimate, Tight insisted that the specification for emergency lights be deleted.

Will strongly objected on obvious grounds. When he refused to yield, Tight brought charges against him, claiming that he was a disruptive influence. Although his immediate superior, the architect, did not support these charges, he did not fight for Morgan, who was ultimately dismissed by the university. Morgan is now suing for wrongful discharge.

Making Good Wafers Look Bad
Don Fisher, an electrical engineer, worked for Dicers, a company that purchased wafers for microprocessor chips from another company and then diced, packaged, and sold them. Don was assigned the task of testing these wafers. After a while, he was instructed by his manager to alter the testing process in such a manner that the quality of the purchased wafers was made to seem lower than it really was, which had the effect of the lowering the price paid. Don objected to this practice and refused to go along. Eventually, he was discharged .

Air Bags
SafeComp is a company that, among other things, designs and makes sensing devices for automobile air bags. Bob Baines was hired to work in the quality control department. About six weeks after starting work, he was asked to sign off on a design that he felt very uncertain about. He checked with people involved in the design and found the situation, at best, ambiguous.

Bob told his manager that he would not feel right about signing off, and, since he was relatively inexperienced with SafeComp's procedures, asked that he not be required to do this. His manager kept applying pressure. Eventually, Bob decided that he wished neither to violate his principles by doing something that he thought was wrong, nor to become involved in a battle in which his career would certainly be major casualty. He quietly resigned. (For a little more information on this case, see [3]).

Flight is also Risky
Ralph Sims had worked for the US Government for many years as an engineer, rising to a fairly high managerial position. On retirement, he accepted an executive position with SuperCom, a company producing electronic equipment for the military.

Shortly after coming on board, Ralph was informed by a subordinate that, for a long time, a key test on an important product was not being made in the manner specified by the contract. This had been going on for several years and the subordinate felt very uncomfortable about it. Ralph, who had considerable expertise in the technology involved, looked into the matter carefully. It turned out that, in his previous career, he had acquired some knowledge about the specified test.

He found that, a shorter, and hence less costly, test had indeed been substituted for the required one. But, after some study, he concluded that SuperCom's test was actually as effective as the specified test. Nevertheless, by this unauthorized substitution, SuperCom was violating the contract and exposing itself both to criminal and to civil prosecution. He took his findings to upper management and urged them to apply to the contracting agency for a contract change authorizing the simpler test. Ralph felt confident that such a change would be accepted.

But his arguments were not accepted and SuperCom continued on their previous course. Ralph did not see why he should get into an unpleasant battle with the SuperCom's leaders over this, since there were no safety issues and even the quality of the product was not actually at stake. Nevertheless, he did not wish to be involved in a dishonest and probably illegal operation. Therefore, he chose the course of quietly resigning, without "turning in" the company.

About three years later, a SuperCom employee reported the deception to the government, and a criminal investigation was launched. When he resigned, Ralph had signed a non-disclosure agreement as a condition for receiving some severance pay. Nevertheless, when called upon by the prosecutor's office to give information about the situation, he cooperated fully.

To his dismay, when the indictments came down, he was one of the people charged with complicity in the fraud. This necessitated his hiring an attorney and undergoing both the expenses and anguish of being a defendant in a criminal case. Fortunately for him, after many months, a new prosecutor was assigned to the case. Shortly afterward, the charges against Ralph were dropped. But, meanwhile, the affair had cost him tens of thousands of dollars in out-of-pocket expenses, not to mention lost time and anxiety.

Some Remarks
Although the IEEE Ethics Hotline was listed in the IEEE Institute and occasionally mentioned in that publication, my impression is that only a small percentage of IEEE members were aware of its existence. Therefore, I suspect that the cases that came to us represent only a small fraction of what is out there. I feel that providing engineers facing ethics related problems with advice from experienced people as well as a sympathetic ear is clearly very useful. But, it should also be evident from the samples provided above, that engineering societies could do a lot more to help. Aside from the obvious value of providing some financial aid, there is also the possibility of low key intervention at the early stages. For example, such intervention in the ICU case and in the air bag case might have had very beneficial effects for all concerned, including not only the engineers and the general public, but also the employers. The mere presence in such cases of a large organization expressing an interest in the engineer's situation and contentions changes the entire picture. It makes it far more difficult for an employer to casually brush off an engineer expressing serious professional concerns. We have seen evidence of this in the past and, in a few recent cases, not mentioned here. An interesting aspect of some of these cases, fully consistent with other such cases previously on record, is the blatant irrationality displayed by some managers. What combination of ignorance, arrogance, stupidity, and greed produced the self-destructive behavior of management in the respirator case?

Acknowledgment:
Listing names of people who are part of an important enterprise is analogous to making up a list of people to invite to a wedding. A very short list omits significant contributors, and a longer list makes omissions more painful. Nevertheless, here is a short list of key figures in the drive to develop and energize ethics support in the IEEE, with apologies to others not mentioned:
Walt Elden, Ray Larsen, Joe Wujek, Mal Benjamin, Joe Herkert.

References

Stephen H. Unger, "What Happened to Ethics Support?", Letter, IEEE Institute, December, 1999, p. 15.

Stephen H. Unger, in Spring 1999 issue of IEEE Technology and Society magazine.

Stephen H. Unger, "Reality check: ethics and air bags", IEEE Institute, August, 1998, p. 2.