THE ROYAL SCHOOL OF SIGNALS

TRAINING PAMPHLET NO: **359**

DISTANCE LEARNING PACKAGE
CISM COURSE 1998
MODULE 6 – SOFTWARE ENGINEERING

Prepared by:

Technology Wing
CISM Group

Issue date: 29 September 2000

ARMY Training & Recruiting
ARMY TRAINING AND RECRUITING AGENCY

DP Bureau

Training &
Recruiting
ARMY TRAINING AND RECRUITING AGENCY

CW/P-359.doc

Authority: ______________________________ **(Name in blocks)**

Signature: ______________________________

Date: ______________________________

Review Date	**Review Date**	**Review Date**
Signature	**Signature**	**Signature**

Authority: ______________________________ **(Name in blocks)**

Signature: ______________________________

Date: ______________________________

Review Date	**Review Date**	**Review Date**
Signature	**Signature**	**Signature**

Authority: ______________________________ **(Name in blocks)**

Signature: ______________________________

Date: ______________________________

Review Date	**Review Date**	**Review Date**
Signature	**Signature**	**Signature**

C O N T E N T S

SOFTWARE ENGINEERING

1. OVERVIEW

INTRODUCTION

1. Recent years have seen a great proliferation of computer systems. Virtually everybody in the developed world uses a computer in some form or another, be it an obvious desk-top PC, a video recorder or a telephone. This growth has been brought about, not only by the reductions in cost and increases in power of the hardware, but also by the flexibility in the use of such systems. The ability to change their function by simply running different software programs makes computers arguably the most versatile machines ever invented. Indeed, it is the software that is fast becoming the major cost of most computer systems. Each advance in speed, size and power of the hardware enables new tasks to be performed and more complex problems to be solved using software. Therefore, as it is with any investment, the software purchased must be cost effective (ie it does the job intended at economic cost), and consequently, needs to be developed in a cost effective way.

SOFTWARE ENGINEERING

2. Software engineering is concerned with the theories, methods, procedures and tools which are needed to develop the software for computers. As you will see, developing software is much like other engineering disciplines because the general process of specification, design, build and test is followed. However, unlike other engineering disciplines, software is not tangible. For example, you can see and touch a bridge, a car, a radio etc, but you can only see representations of the software, eg a paper listing of the source code, a floppy disk containing the executable code, a screen of data displayed by running the executable. Moreover, modern software systems are larger and more complex than ever before and require large teams of software engineers to build them. Unfortunately, the methods and practices used to develop small programs by one or just a few people have been found not to scale up. This is due mainly to the difficulty with communicating development problems, solutions and ideas amongst the development team.

3. Management of the software development process is therefore unique and difficult - you can manage more easily what you can see than what you can’t see. Consequently, various techniques are used to try to make aspects of the software more visible. Such techniques include design documents, code listings and print-outs of test results, and there are many proprietary and general methods and methodologies that have been developed to this end.

SOFTWARE AND SOFTWARE DEVELOPMENT

4. What actually is software and how can we build it? A good question and, as discussed above, it is difficult to give a precise answer! At the lowest level for the purposes of this course, software is the executable code or program of instructions ie the patterns of ones and zeroes that are interpreted by the micro-processor (eg Pentium). At the highest level it is the requirement to perform a task as specified by the user perhaps as a set of high level instructions. So software development can be described as the process by which instructions at the highest level (generally understood by an application specialist) are translated into instructions at the lowest level (understood by a particular type of machine).

SOFTWARE PRODUCTS

Unfortunately, the concept of a software product ie what you buy, install and run, is a lot more than just those low-level instructions. As mentioned already, modern software systems are large and complex (eg a word-processor) and users need guidance on how to perform any particular task using the software. Therefore, as part of the software product there will be a user guide or manual which explains how this software is to be installed and how it performs the various tasks. These items would be specified in the terms of the contract agreement and, in the case of many modern Commercial Off-the-Shelf (COTS) packages, the whole lot comes on a CD-ROM.

Student Exercise 1.1: Compare and contrast how a particular task can be performed in different ways in software systems with which you are familiar. For example, **cut and paste** in different word-processors (eg MS Word and Word Perfect) or different packages (eg MS Word and Excel) using pointer devices (eg mouse) and/or keyboard.

6. On the other hand, the developers will be required to enhance, modify and correct the system they have built. Therefore, their view of the software product would also include all the documentation used during the development ie designs, test plans and results, etc. Indeed, for some bespoke systems, these items may also be delivered to the customer if the maintenance task is to be transferred as part of the contract as well.

SOFTWARE ATTRIBUTES

7. Like the other engineering disciplines, software engineering is not just concerned with producing products but producing them in a cost-effective way. Given unlimited resources, the majority of software problems can probably be solved. The challenge to software engineers is to produce quality software with a finite amount of resources and to a specified schedule. (Therein lies the rub! The means to provide software cost estimates will be addressed later in the course).

8. The attributes of the software are those which appear once it has been installed and is in use. It concerns the product's dynamic behaviour and not the services it provides. Sommerville (SOM96) considers that the essential attributes are maintainability, reliability, usability and efficiency which are outlined in Fig 1 below. Consider maintainability for example. Systems are frequently specified several months, if not years, before the system is finally commissioned. Consequently, the delivered system will not meet fully the real needs of the user since the job will have changed since the requirements were produced. Therefore, the system will have to be modified to include these changes (plus any corrections for features that don't work as intended). Well-engineered software should be capable of taking these changes on board with the minimum of disruption and cost.

Product Attribute	**Description**
Maintainability	The software should evolve to meet the changing needs of the customers.
Reliability	The software should run consistently as intended and be resilient to erroneous data. Unexpected failures should be handled to give graceful degradation and not cause human, physical or economic damage.
Usability	There should be adequate documentation to use and maintain the system and the user interface should be appropriate.
Efficiency	The software should not be wasteful of system resources such as memory and processor cycles.

Fig 1. Attributes of well-engineered software (SOM96. p6)

9. However, these attributes may conflict and so priorities need to be defined for each project. For example, a better interface may reduce the system efficiency. Furthermore, there is the law of diminishing returns where eventually the costs of improvement will outweigh the advantages to be gained.

Student Exercise 1.2: Describe how the priorities differ in the following projects:

- An avionics system for a small fighter aircraft.
- A company's personnel records system that includes monthly pay calculations.
- A tourist information system to be used by the general public.

MAINTENANCE

10. At this point it is worth pointing out that, like hardware, software does deteriorate during the operation and maintenance period. The pattern of failure rate for hardware is quite well known, ie the bathtub curve: there will be an initial period of high failure rate due to manufacturing defects, but this will drop to a steady state (as low as possible), before rising again as more key components start to wear out.

11. The failure pattern for software could be considered to follow an idealised curve (Fig 2). Initially, there will be a period of time when errors are discovered that weren't found during testing. As these errors are fixed so the system settles down to a steady state of fixing obscure faults and providing upgrades.

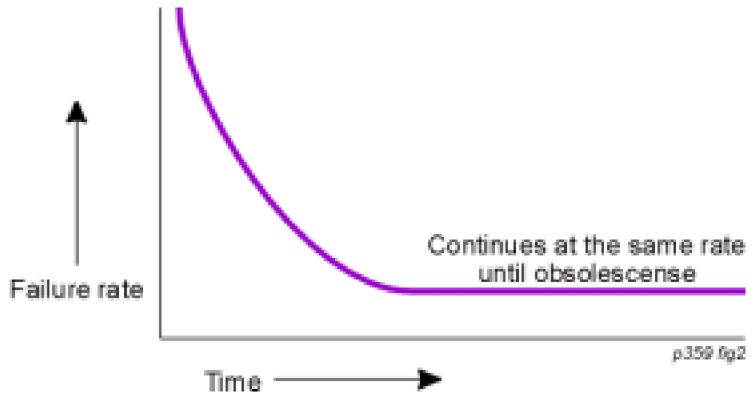


Fig 2. Idealised failure curve for software (PRE92, p 12)

12. However, experience shows that this is not the case and the actual curve is more like that in Fig 3. The reasons for this are twofold. First, each upgrade will produce its own set of faults which will slip through testing and be found during use, hence the spikes. Before the curve can return to its steady state, another change is implemented and, over time, this has the effect of producing an underlying increasing minimum failure rate. Second, as changes are incorporated, the design moves further away from the system architecture that was ideal for the original system but not the modified one. This means that changes become more difficult to incorporate and so the number of new defects at each change increases. This is discussed very well by Brooks (BRO95, p 120-123).

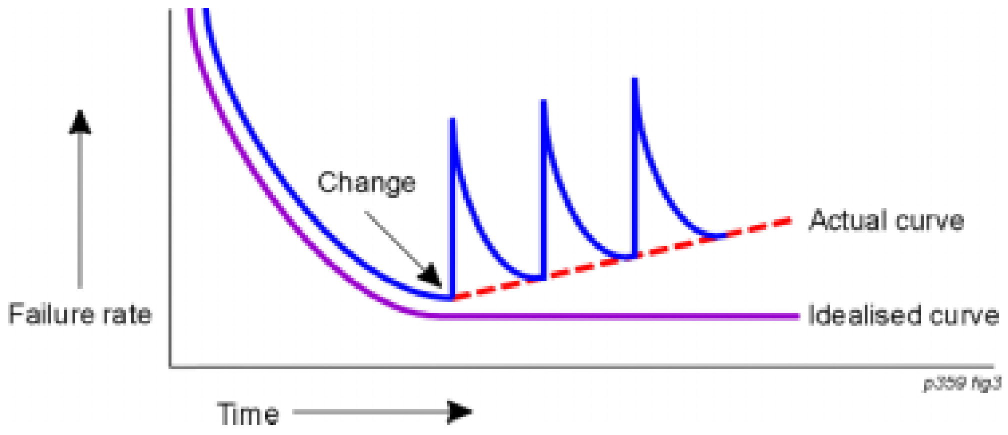


Fig 3. Actual failure curve for software (PRE92, p 12)

13. Current research is examining this problem and trying to identify the point at which it would be more cost-effective to redesign and build a new system than spend money on supporting the old one. This is particularly pertinent when it is realised that an error found during operation can easily cost one hundred times as much to fix than if it had been found at the requirements stage. This will be discussed further later on in the course.

Student Exercise 1.3: Identify some other differences between mechanical machines and software that may affect their system reliability and the maintenance activities.

2. SOFTWARE LIFE CYCLE

MODELS

14. We have already discussed the need to make software visible so that the development can be managed more easily. The first technique is to define a model of the development process ie identify the different phases and arrange them in a model according to how they interact with each other. In this way we can see what needs to be done for a given software product and what has been done at any stage, thereby providing us with a map.

15. It was stated previously that the software development process followed the same pattern as other engineering processes. In essence this is so. The traditional engineering project typically follows:

- Produce Requirements Specification
- Produce initial design
- Build and test prototype
- Refine design
- Build and test product
- Start mass production and/or maintenance

16. In a software project, the difference is that the prototype is usually the final product. This is because the visible parts of the prototype software (eg the user interface or the data output) are identical to those required in the final product. So once it appears to be working, ship it. However, unless the maintainability attribute discussed earlier is a priority, it will be difficult to modify the system with those enhancements found during use that would otherwise have been found with a prototype. Moreover, the customer will be supplied with a less than ideal system.

WATERFALL MODEL

17. The first explicit model developed was called the Waterfall model and was derived, not unexpectedly, from traditional engineering processes (Fig 4).

18. In this model, each phase, starting with the *requirements definition*, produces a deliverable which acts as the input to the next phase. Hence, it is described as a ‘deliverable oriented’ method. Progress is thus shown flowing down the Waterfall until the *operation and maintenance* phase is reached. Unfortunately, this assumes that each phase is complete and gets everything right so that no errors are passed onto the next phase. To cope with this, paths are added so that previous phases can be revised if errors are found.

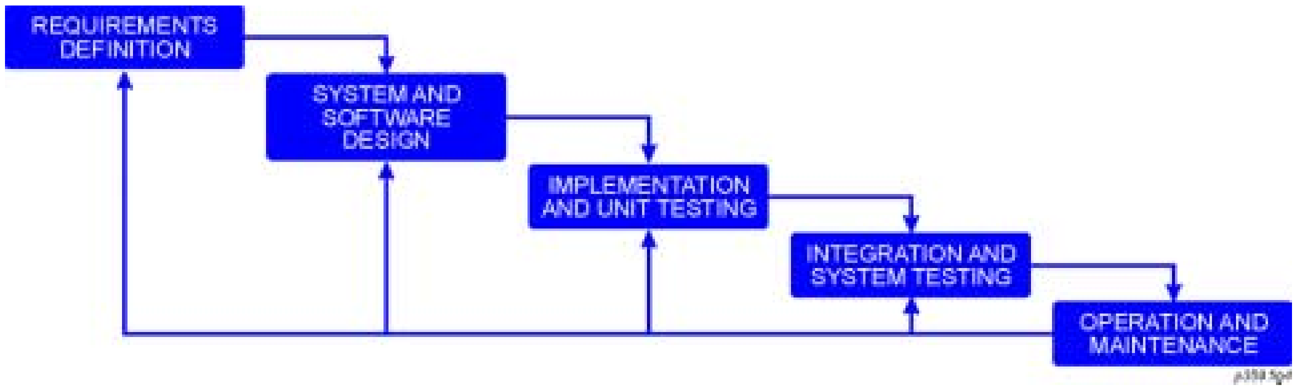

REQUIREMENTS DEFINITION
SYSTEM AND SOFTWARE DESIGN
IMPLEMENTATION AND UNIT TESTING
INTEGRATION AND SYSTEM TESTING
OPERATION AND MAINTENANCE
p359 5gd

Fig 4. The Waterfall Software Life Cycle Model

19. Each phase is described by Sommerville (SOM96, p 10) as follows:

a. Requirement analysis and definition.

The system's services, constraints and goals are established by consultation with system users. They are then defined in a manner which is understood by both users and developers.

b. System and software design.

The systems design process partitions the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves representing the software system functions in a form that may be transformed into one or more executable programs.

c. Implementation and unit testing.

During this stage, the software design is realised as a set of programs or program units. Unit testing involves verifying that each unit meets it specification.

d. Integration and system testing.

The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

e. Operation and maintenance.

Normally this is the longest and costliest life cycle phase, about 70% of the total. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered. This may involve repeating some or all of the previous stages.

20. In practice, these stages overlap and feed information to each other. During design, problems with the requirements are identified; during coding, design and perhaps requirement problems are found and so on. The software process is not a simple linear model but involves a sequence of iterations of the development activities.

21. Unfortunately, a model which includes frequent iterations makes it difficult to identify definite management checkpoints for planning and reporting. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored or programmed around. This premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

22. A key problem with the Waterfall model is its inflexible partitioning of the project into these distinct stages. Delivered systems are sometimes unusable as they do not meet the customer's real requirements. Nevertheless, the Waterfall model reflects engineering practice. Consequently, it is likely that software process models based on this approach will remain the norm for large hardware-software systems development.

<u>'V' MODEL</u>

23. One such variation is the 'V' or STARTS model (see Fig 5).

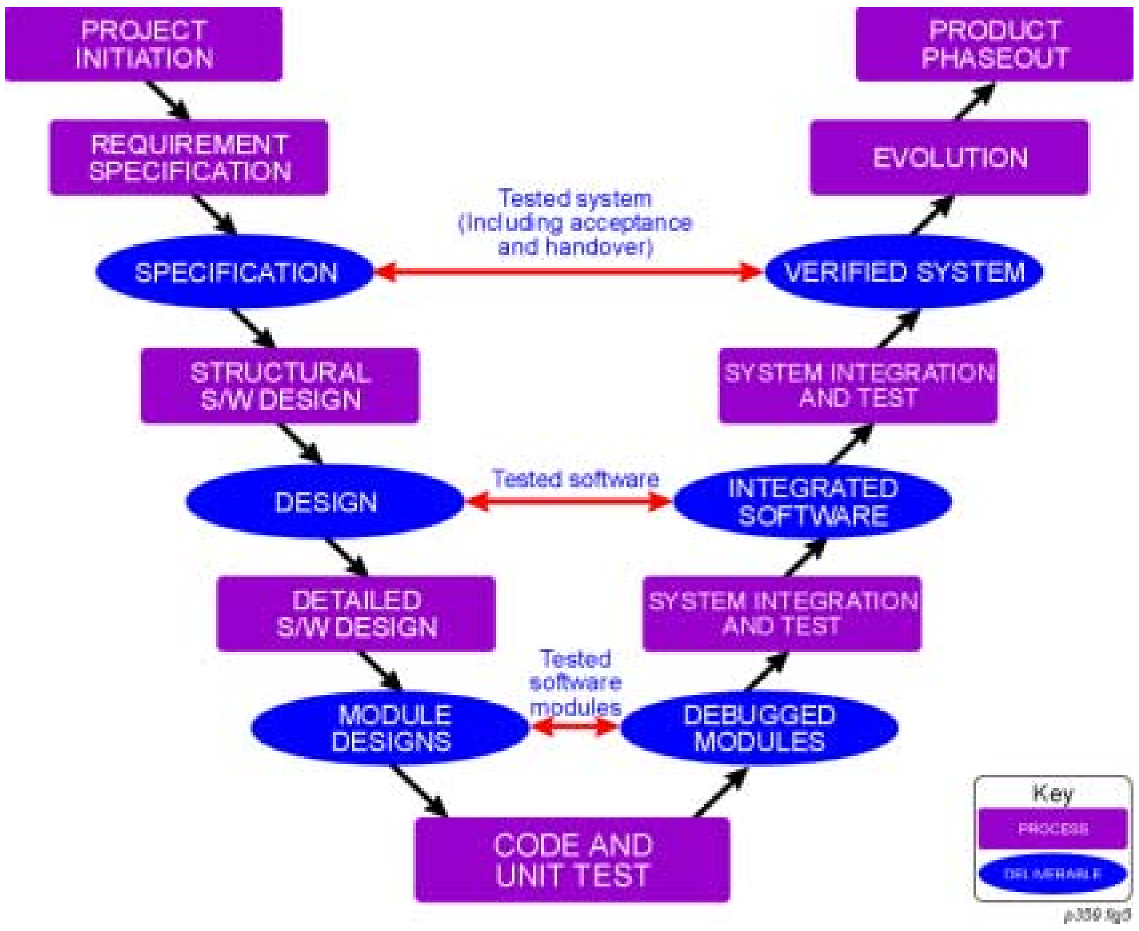


Fig 5. 'V' or STARTS Life Cycle Model

24. One of the weaknesses in the standard Waterfall model is that testing is shown so far down the line ie after the code has been developed. On the face of it this seems sensible. However, testing can be a very complex phase in itself and requires a lot of planning and preparation. Therefore it needs to be considered much earlier in the cycle to ensure efficient and continuous development. From the 'V' model it can be seen that an acceptance test plan can be initiated when the specification has been approved. This is because the acceptance tests will be verifying that the system developed is what was specified by the customer. Likewise, test plans for the integrated software and the software modules can be started when these specifications become available. Consequently, all test planning can take place in parallel with the system development and not left until the last minute.

EVOLUTIONARY DEVELOPMENT

25. One approach to overcoming the weaknesses identified in the Waterfall model is to accept that insufficient detail will be available at any stage to produce a complete system first time. Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refining this through many versions until an adequate system has been developed (see Fig 6). Rather than have separate specification, development and validation activities, these are carried out concurrently with rapid feedback across these activities.

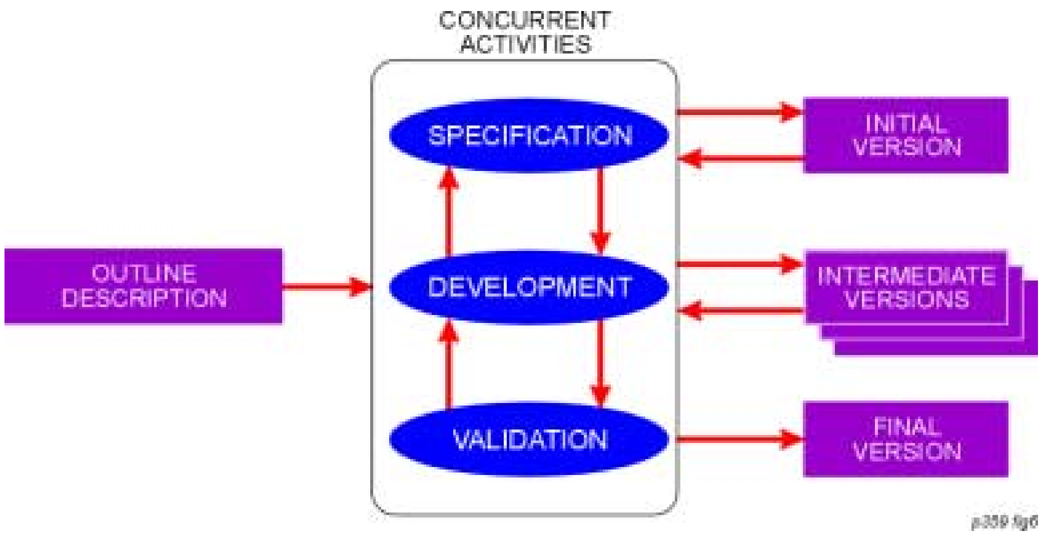


Fig 6. Evolutionary Development

26. Sommerville (SOM96) suggests 2 types of evolutionary development: exploratory programming and throw-away prototyping.

a. Exploratory programming is where the objective of the process is to work with the customer to explore their requirements and deliver a final system. The development starts with the parts of the system which are understood. The system evolves by adding new features as they are proposed by the customer. Sometimes this is also called evolutionary prototyping.

b. Throw-away prototyping is where the objective of the evolutionary development process is to understand the customer's requirements and hence develop a better definition for the system. The prototype concentrates on experimenting with those parts of the customer requirements which are poorly understood. Any resultant software is discarded since it will probably not fit in with the architecture of the final system.

27. Exploratory programming is essential when it is difficult (or impossible) to establish a detailed system specification. Some might argue that all systems fall into this category. However, exploratory programming is mostly used for the development of Artificial Intelligence (AI) systems which attempt to emulate some human capabilities. As we don't understand how humans carry out tasks, setting out a detailed specification for software to imitate humans is impossible. More recently, the development of rapid application development systems such as Delphi, Visual Basic and Powerbuilder have led to using this method for prototyping and developing MS Windows based applications where the user interface is a key factor.

28. Therefore, the evolutionary approach to software development is usually more effective than the Waterfall approach in producing systems which meet the immediate needs of customers. However, from an engineering and management perspective, it has 3 basic problems:

a. The process is not visible.

Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents which reflect every version of the system.

b. Systems are usually poorly structured.

Continual change tends to corrupt the software structure as we saw earlier in Maintenance. Software evolution is therefore likely to be difficult and costly.

c. Special skills are often required.

It is not clear how the range of skills which is normal in a software engineering team can be used effectively for this mode of development. Most available systems developed in this way have been implemented by small teams of highly skilled and motivated individuals.

29. To resolve these problems, the objective of evolutionary development is sometimes the development of a system prototype. This is used to understand and validate the system specification. Here, the evolutionary development process is part of the wider process (such as the Waterfall process).

30. Because of these problems, large-scale systems are not usually developed in this way. Evolutionary development is more appropriate for the following:

a. The development of relatively small systems. The problems of changing the existing system are avoided by re-implementing the system in its entirety whenever significant changes are required. If prototyping tools are used, this need not be too expensive.

b. The development of systems with a short lifetime. Here, the system is developed to support some activity which is bounded in time. For example, a system may be developed specifically for the launch of a new product.

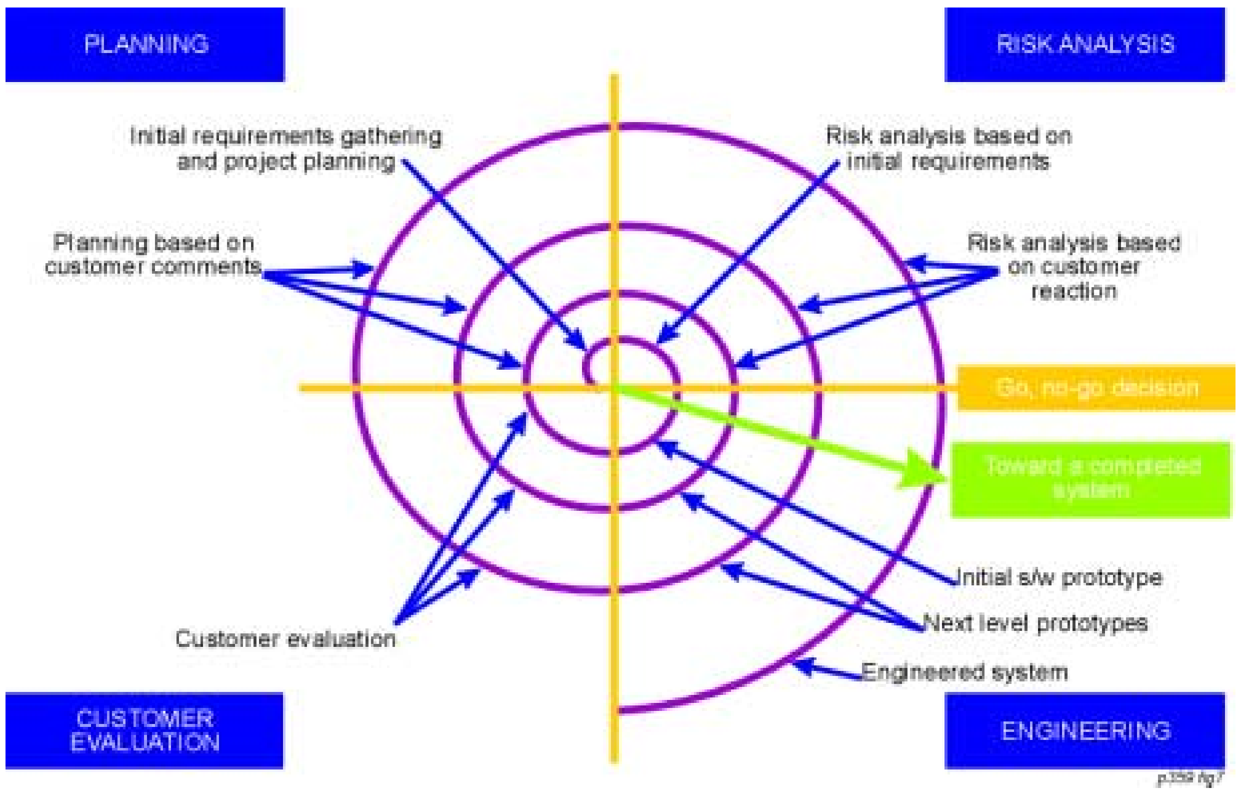
c. The development of system or parts of larger systems where it is impossible to express detailed specifications in advance. Examples of this type of system are AI and user interfaces (as mentioned above).

Student Exercise 1.4: Explain why the commercial and financial sectors are employing so many programmers with skills in Visual Basic and Delphi.

BOEHM'S SPIRAL MODEL

31. Despite the problems with the Waterfall model it is still the one most adopted by government agencies and large software procurers due to its superior visibility over the evolutionary models. What is needed is a model that subsumes the visibility and the iteration features of the 2 paradigms which is then acceptable to management, procurers and developers. Such an approach was proposed by Boehm (BOE88) which also has the advantage of explicitly recognising risks within the project.

32. As discussed by Pressman (PRE92, pp 29-30), the model takes the form of a spiral (Fig 7) with each iteration passing through 4 quadrants which represent 4 major activities. Each iteration is considered a phase but the number of phases is project specific.



PLANNING
RISK ANALYSIS
Initial requirements gathering and project planning
Risk analysis based on initial requirements
Planning based on customer comments
Risk analysis based on customer reaction
Go, no-go decision
Toward a completed system
Initial s/w prototype
Next level prototypes
Engineered system
Customer evaluation
CUSTOMER EVALUATION
ENGINEERING
p359 fig7

Fig 7. Boehm's Spiral Model (PRE92, p 29)

33. The 4 activities are:

a. Planning.

Determination of objectives, alternatives and constraints.

b. Risk analysis.

Analysis of alternatives and identification and resolution of risks.

c. Engineering.

Development of the next level product.

d. Customer evaluation.

Assessment of the results of engineering.

34. An intriguing aspect of the spiral model becomes apparent when we consider the radial dimension. With each iteration around the spiral, beginning at the centre, progressively more complete versions of the software are built. During the first circuit, the objectives, alternatives and constraints are defined, and the risks are identified and analysed. If risk analysis indicates that there is uncertainty in requirements, prototyping may be used in the engineering quadrant to assist both the developer and the customer. The customer evaluates the engineering work and makes suggestions for modifications. Based on these suggestions, the next phase of planning and risk analysis occurs and the culmination of the risk analysis results in a 'go, no-go' decision. If the risks are too great, the project can be terminated.

35. In most cases, flow around a spiral path continues, with each path moving the developers outward towards a more complete model of the system and, ultimately, to the operational system itself. Every circuit around the spiral requires engineering (lower right quadrant) that can be accomplished using either the classic Waterfall or prototyping approaches. It should be noted that the number of development activities occurring in the engineering quadrant increases as activities move further from the centre.

36. The spiral model for software engineering is currently the most realistic approach to the development for large scale systems and software. It uses an evolutionary approach enabling the developer and customer to understand and react to risks at each evolutionary level. It uses prototyping as a risk reduction mechanism, but more importantly, enables the developer to apply the prototyping approach at any stage. Yet, it maintains the systematic stepwise approach suggested by the classic Waterfall within a more realistic iterative framework. Consideration of the technical risks is required in each phase and, if properly applied, should reduce risks before they become problematic.

37. Unfortunately, the spiral model is not the panacea; it too has weaknesses. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not discovered, problems will still undoubtedly occur. Furthermore, it is relatively new and it is difficult to convince large customers that such an evolutionary approach is controllable. Therefore, it will be a number of years before a true evaluation of it can be made with any certainty.

Student Exercise 1.5: Brooks advocated in 1975 (BRO75, p 116) "plan to throw one away; you will anyhow", yet in his follow-up book in 1995 (BRO95, p 265) he admits that he was wrong. What do you think changed his mind? (Hint: Consider the dates.)

PROCESS VISIBILITY

38. We have already discussed the intangible nature of software systems and that software process managers need documents, reports and reviews to keep track of what is happening. Consequently,

most organisations involved in large system development use a ‘deliverable oriented’ process. Each activity must end with the production of some documents and it is these documents that make the software process visible.

39.	Unfortunately, regular document production has some drawbacks:

a.	Management needs regular deliverables to assess project progress. However, the timing of management requirements may not correspond with the time needed to complete an activity. Therefore, extra documents may have to be produced adding to the cost of the process.

b.	The need to approve documents constrains process iteration. The costs of going back and adapting a completed deliverable are high. If problems are discovered during the process, inelegant solutions are sometimes adopted to avoid the need to change ‘accepted’ project deliverables.

c.	The time required to review and approve a document is significant. There is rarely a smooth transition from one phase of the process to the next. In reality, formal procedures are sometimes ignored and development may even continue before the previous phase has been completed.

40.	Since the waterfall process model is still the most widely adopted ‘deliverable’ model, Fig 8 shows one possible way of splitting it into stages and the deliverables which might be produced at each stage.

Student Exercise 1.6: What correlation is there between the list in Fig 8 and the V model in Fig 5?

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

Fig 8. Documents from the Waterfall Model

SUMMARY

41. This chapter has introduced the novelty of software and considered how it is different to other products (eg tangibility, reliability and maintainability). It also defined software engineering and identified that the traditional engineering process can be used as a foundation for developing software. However, software's intangibility brings with it a new set of problems, particularly for management when trying to track progress. Therefore, models have been devised to help make the progress more visible (eg Waterfall and V) yet they have other problems. The most important of these is that they have difficulty reflecting the real process of software development, especially in large projects. The evolutionary approach overcame the reality problem but didn't help management. The best of both paradigms were brought together into Boehm's spiral model. The remainder of the course will focus on the construction of software and how various techniques can be employed to make that construction as effective as possible.

REFERENCES

BOE88	Boehm, B, *A Spiral Model for Software Development and Enhancement,* Computer, Vol 21, No 5, May 88, pp 61-72.
BRO75	Brooks, F, *The Mythical Man-Month - Essays on Software Engineering*, Addison-Wesley, 1975.
BRO95	Brooks, F, *The Mythical Man-Month - Essays on Software Engineering*, Anniversary Ed, Addison-Wesley, 1995.
PRE92	Pressman, R, *Software Engineering - A Practitioner's Approach,* 3rd Ed, McGraw-Hill, 1992.
SOM96	Sommerville, I, *Software Engineering*, 5th Ed, Addison-Wesley, 1996.

SUGGESTED SOLUTIONS TO STUDENT EXERCISES

1.1

Your answer will depend on the tasks you chose to analyse. The key points to note are that even to do the same simple task, different programs have slightly different sets of actions and even the same program has a selection of ways. In addition, not all the ways are very intuitive, for example, in a Windows based program are the icons for cut and paste self-explanatory? The user needs to know these facilities and how to use them.

1.2

The avionics system will need to be small and fast therefore efficiency will be a high priority. Furthermore, it will need to be reliable - aborted missions due to systems failure are very expensive. Usability is also quite important but pilots are already trained as expert users.

The company's record system needs to cope with changes in tax and pension schemes so maintainability will be a high priority. It will also need to have high reliability. It need not be very efficient and it will have a regular team of users.

The tourist information system will need a high priority for usability (differences in languages, understanding and physical abilities etc) and fairly high for reliability and maintainability. It need not be very efficient.

1.3

Here are just a couple of points, you may have more:

- Mechanical machines have parts that wear through use, so the concept of preventative maintenance is employed. Fault prevention in software is performed during development.

- Software will not deteriorate if it isn't changed. However, this is only applicable if it meets the requirements fully and they don't change.

1.4

Many of the applications written in these languages for these sectors are interfaces to databases, or where the time to get a new product to market, ahead of the competition, is critical.

1.5

Brooks' first book was written when the Waterfall model had been used widely and many of its weaknesses were becoming apparent. Therefore, he was suggesting that the first version to be built should be treated as the prototype and thrown away, and the knowledge learnt from it should be used to build the operational system. Since then there have been improved models (eg Boehm's spiral) and rapid application development tools. However, you can argue that he was still correct - don't deliver the prototype - although we still tend to.

1.6

The V model not only reflects the various stages but also includes the test plans at the various stages
.