Computer Science 415.340

Operating systems

INTRODUCTION

WHAT IS "OPERATING SYSTEMS" ?

The organisation of this course is different from that which you will commonly find in textbooks on the subject, so we think it appropriate to begin by telling you why. The basic reason is our dissatisfaction with the traditional course structure, which we did not find sufficiently flexible to cope in a natural way with developments in operating systems practice. Over the past fifteen or so years, the variety of operating systems in common use has expanded enormously, and the rate of expansion shows no sign of diminishing. From a universal pattern of interactive work through typewriter-like terminals to a more or less remote "mainframe" (a large and expensive computer, usually with operating staff), with support from a batch processing system of some sort, we have moved to a much more diverse way of working, with stand-alone microcomputers from desktop to palm size, widespread networking, graphics, and other less obvious changes, all of which call into question the traditional view.

Not only have things changed; they have changed very quickly, and show every sign of continuing to do so. We are not convinced that in this fluid world a description of one or two current operating systems is the best way to introduce the subject. Instead, we believe that it is much more important to seek out the principles behind the implementations, and to discuss what these are, and how they interact in a large and complex operating system. That is what we have tried to do.

Some years ago, therefore, we began to look for a different approach which we hoped would be able to accommodate continuing change in a natural way. Our aim was to find a model for an operating system which would be accurate, flexible, likely to persist beyond the lifespan of any individual operating system, and a suitable foundation for our stage 3 operating systems lecture course. Here's a summary of our deliberations.

THE "MANAGER" MODEL.

This is the traditional model of an operating system. The system is defined as

That which manages the available resources,

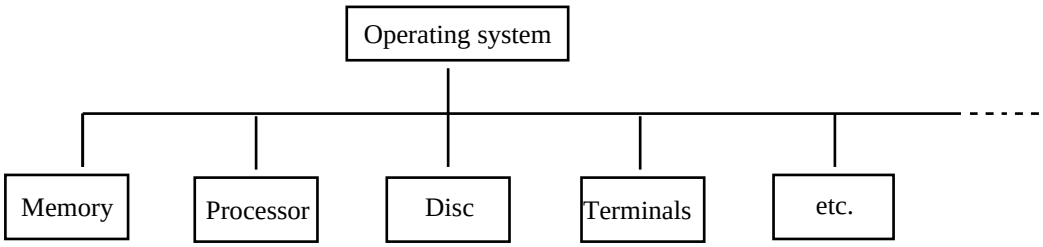
usually with emphasis on hardware management. While management is certainly part of the operating system's function, we found this definition too narrow. Real operating systems are concerned with user interfaces and control languages and file system (not just disc space) organisation and accounting and many other activities. While one can always fit such tasks into some sort of management, there is often an element of contrivance in the description which results.

The manager model does not cope very well with the diversity of operating system configurations. Once we have developed a system appropriate for a mainframe-based interactive service, we have to start again when developing a microcomputer system. Of course, the job can be done – but the example shows that some significant factor in operating systems is missing from the model. This is particularly obvious now, when many microprocessors are comparable in processing ability to the mainframes which they replaced, but we still require different sorts of operating system for the two cases.

We could conclude that the "manager" model is useful, but is still lacking something. With the benefit of hindsight, we might now suggest that the "something" is a goal – you can't manage something unless you have some criterion by which to evaluate the management, and apart from rather arbitrary ideas of efficiency no such criterion was evident. Considerations of efficiency alone don't resolve our problem with the time-sharing and microprocessor systems.

You will gather that we don't think much of this model, but it's important to make one point in its defence : the model is realistic from the point of view of the operating system itself. The operating system *is* that which controls what happens in the computer; ideally, nothing happens in the computer except by permission of the operating system. Our point is not that the model is wrong, but that it isn't particularly helpful.

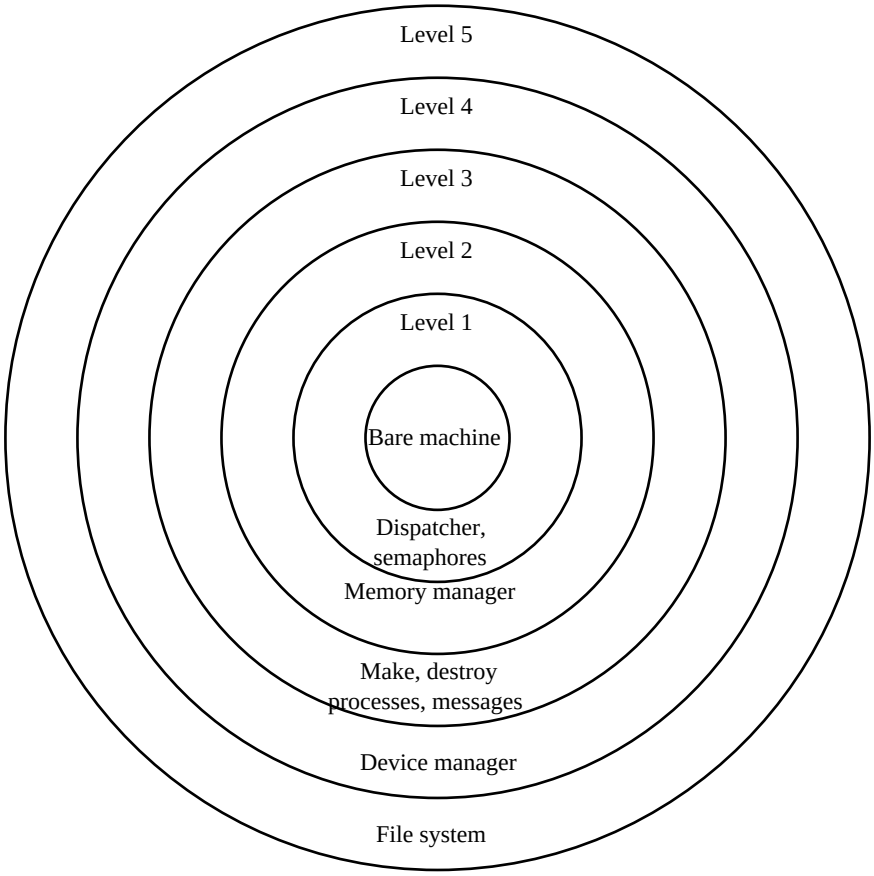
Here's a picture of the manager model. Its main characteristic is the subservience of everything to the operating system.



THE ONION MODEL.

This model was never really what you might call a mainstay of our course, but we mention it here because the ideas which it embodies are valuable, and they certainly appeared in the course. The system definition does not change significantly – the system is still a resource manager – but notions of internal structure are added to the resource manager principle. The system is seen as a layered construct resembling the layers of an onion, with the hardware at the centre, incorporating ideas of information hiding which are still with us. This is a venerable model; the figure (next page) was presented in the textbook of Madnick and Donovan[INT5], dated 1974. The interdependencies between the topics are explicitly noticed in this approach.

These models were used in the early treatments because they are the traditional models. They were once exactly right. They are still not wrong; the principles they embody are still valid, and appear particularly clearly in systems such as those based on client-server models. Our reservations are not that the older models have become invalid, but that they are limited, and do not provide the support we want for our course organisation.



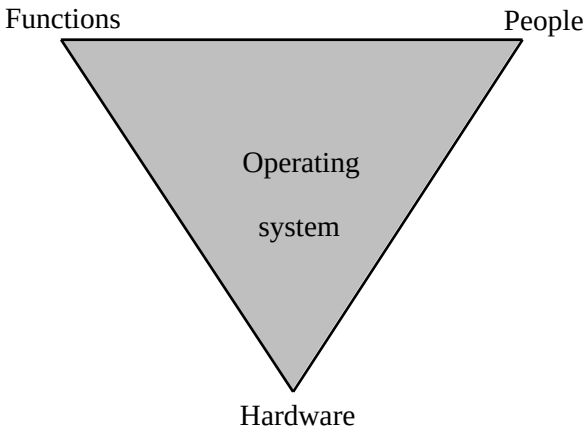
THE "DUSTBIN" MODEL.

Our next model, which lasted for some time, was developed from the observation above that the operating system was required to handle a large number of necessary, but more or less unrelated, activities. This leads to the dustbin definition of an operating system :

That which does all the things that no one else wants to do.

This is a nice snappy definition. It isn't particularly precise, but it expresses the ragbag nature of the subject, and explains why it's like that – the system must cover all the bits we must have in order to run the programmes effectively, and which are neither operations handled by hardware nor things which people should be expected to do (and can be relied upon to do) for themselves. The very significant advantage of the dustbin model is that it's true – that is exactly how the operating systems developed.

It also implies a new dimension to the system; it includes the idea that the computer system is there to run programmes written by people, who don't really come into the manager model except perhaps as rather inconvenient entities to be managed somehow. In adding this dimension, we recognise that some of the functions of the complete system will be discharged by programmes supplied by people, but that it's unreasonable (not to mention unsaleable) to expect the people to look after all the tiny details of the hardware. This approach leads us to a rather different view of the operating system :



The corners of the triangle represent the active entities in the computer system – each is a component which can in some sense do things. We shall call these the *primary entities* of the system, which sounds impressively academic. The real work, though, is done by cooperation of all three primary entities, and we represent these activities by the area within the triangle. This is the domain of the operating system. In accordance with our definition, it includes all the activities which none of the primary entities is keen, or able, to do, but which have to be handled somehow if we are to get any work done.

As a simple example, consider the action of someone running a programme from a keyboard. Something, somewhere, has to deal with the signals which come from the keyboard, find the programme, get it into memory, point the processor at it, and so on. None of these can reasonably be expected of any of the primary entities acting alone; so they are all operating system tasks.

The new description gives us a better idea of what is in the system, but still offers no motivation. In effect, it's a map of the terrain, but we are still left on our own to find our way around it. And that's pretty well what you'd expect given the working definition. A definition by exception – one which says what the thing defined isn't – leaves much to be desired as a starting point for an orderly and coherent treatment of a subject.

THE "SERVICE" MODEL.

Well, then, clearly our next step must be to say what the thing defined *is* – to find some positive definition of what we want the operating system to do. (Notice, by the way, that that statement is meant rather precisely : computing is not a natural science, and operating systems, despite appearances, don't grow wild. If we want a thing called an operating system, it really is up to us to define it.) We think that the key to this definition is to move the emphasis away from the hardware to the other corners of the triangle. That we at present use certain sorts of electronic device to get the work done is immaterial; the important thing is that we can define certain useful jobs which we wish to have performed for us somehow. That puts the emphasis strongly on the jobs, rather than the machinery – and, therefore, on the people, who are ultimately the source of the jobs, and what they want to do.

That's *what* we want to do; our task in this course is to study *how* it can be done. Why, then, are operating systems "really" there ? Not – except incidentally – to drive the machine efficiently, nor to allocate resources, nor to cope with multiple processes. They are there precisely to do "everything anyone might ever want"; the new snappy definition of an operating system is, in fact :

That which provides computer services to people.

And that must work – for the only reason we would wish to change or extend a computer system is in some way to provide some sort of service for someone. This is the goal which we wanted, and we can use it to impose order on our development. In terms of our triangular diagram, we shall begin from the *People* corner and work into the triangle, being guided in direction by whatever we need to provide the required services.

You are doubtless now reviewing possible extensions to a system for exceptions to our rule; but we would argue that, provided that we cast our net sufficiently broadly, the definition will work. The closest approximation to an exception which we have found is in operating systems for real-time control use, where the people's interests can be remote, and we are happy to accept that our phrase is a definition of operating systems used by people, rather than by machines. Apart from that, though, we think that it holds, and it's certainly a good enough foundation to support a first course on operating systems.

HOW IT WORKS OUT.

Now we're ready to explore the consequences of our definition, and the resulting structure of our operating systems course. The most significant feature follows quite directly from the definition; now we have defined a goal, we can treat the whole subject as an exercise in system design, and proceed by some form of top-down analysis. Our problems of diversity are solved, as details of hardware and environment will be incorporated naturally into the system design as we need to know them, so the relationships between the time-sharing and microcomputer cases of our example should be clarified. Likewise, as new technological developments come along, they will fit in naturally as additional answers to questions which we ask somewhere in the analysis.

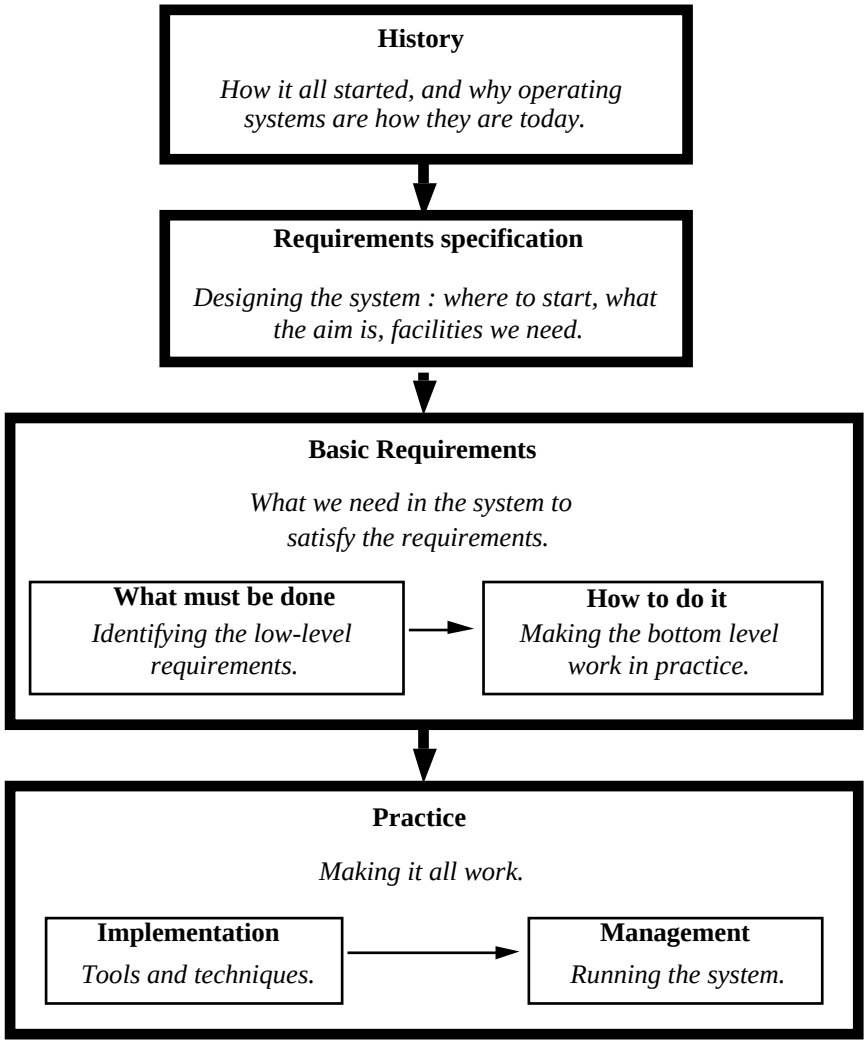
The details of the development are nonetheless not trivial to determine. It took some years of thought, and about four revisions of the order of presentation of the topics (not all of which reached the students) to develop the structure we now see as desirable, and which is embodied in these notes. The result is described in the diagram on the next page. It's a beautiful diagram – so beautiful that it's hardly important that it doesn't always work. It works reasonably well for a book, where you can easily refer to other parts when you so desire, but it's less satisfactory for a lecture course. The reason for this disorganisation is that the top-down serial treatment implied by the diagram is good logic but bad practice; all the development happened in parallel, and the people who worked out how to make the systems work knew about all the other developments as they went along. Real life is messy, though, and we propose to stick with our model until we find a better one. (It does get better as we keep on worrying about it and understand the structure more clearly ourselves. We're learning too, which is as it should be – that's what a university is about.) It should be clear how the parts of the course fit together. As the course proceeds, you might find it helpful to review these ideas from time to time to see where the various topics fit into the pattern.

There is one element in that diagram which does not follow directly from our model – the section on operating system history. We have included that for two reasons.

- First, it introduces most of the topics which turn up in the ensuing analysis, and shows how the current attitude to operating systems differs from that which guided the design (such as it was) of earlier systems.

- Second, it gives us some confidence that the students who take the course have some idea of what we're talking about. A significant problem from our point of view with modern microcomputers is that they hide much of the operating system from view. When our students moved from older and less elaborate

microcomputers to Macintosh machines, we noticed a very pronounced slump in the general level of understanding of what operating systems are about. Now Unix has reappeared in stage 2, we hope for an improvement, but it still seems to be a good idea to include this material in the stage 3 course. One could argue that the material will become clear anyway as we work through the analysis; in practice, though, most people don't work that way. Top-down analysis is all very well in theory, but most people feel happier with it if they can see something of the outcome before the analysis starts. To use an analogy, if you are designing a house, it's easier to begin with some experience of working houses than to stick to pure abstract analysis of some definition such as "A machine for living in" (Le Corbusier, we think).



Finally, it was nice to find – after we'd pretty well settled the course – this opinion from a guru[INT1] :

... a course on operating systems should trace the evolution of operating systems from early batch monitors and timesharing systems to the systems in current use.

LEVELS.

As well as the notion of level which defines the top in top-down, there is the sort of level we speak of in terms like "low-level language". Low, in this sense, means something like "close to the hardware", though the precise meaning depends on the context. For example, a sequence of ideas of increasing level in the context of the input interface might be interrupt, mouse movement or key depression, event or character, word or icon, system instruction. In the diagram above, these ideas turn up in no very obvious order (interrupts in the management, key depressions and events in the implementation, instructions in the requirements specification, etc.) The two sorts of level are, in fact, quite distinct; operating systems is not a linear subject.

The second sort of level is important in deciding what you're talking about. You don't normally expect material on interrupts in a discussion of the system control language, nor vice versa. We have tried to put links in the text where we think they're appropriate, but you'll probably have to do some hunting if you want to find all the connected pieces. The same idea is important in gauging the meaning of an examination question; there is (usually) enough indication in a question to identify the intended level, and it's worth making sure – if you get it wrong, you might answer the wrong question !

WHAT'S IN THIS COLLECTION.

The notes which follow have evolved from a set originally written to accompany the operating systems course in days when we presented the course in the traditional way, using a textbook which covered the topics of the course very unevenly, so that we had to provide additional material. (Other available textbooks had other defects, and the one we had was quite good on the bits it covered.) Since then, the course has changed to include new topics and to eliminate old ones as the course content evolved to reflect the development of computing practice, and it changed again when we switched to a new textbook. Most of all, though, it changed when we started presenting the course in the new order. Indeed, it changed several times, as we juggled with the order of topics searching for one we considered acceptable.

That grew to a fairly substantial body of notes, intended primarily as a way of presenting topics in the top-down order, but not as a textbook in its own right. We included comments and additional material wherever it seemed convenient, either to expand on the conventional textbook material or to emphasise the structure, but we expected that students would refer to a textbook for further detail and factual information. The collection of material reflected the historical development of the course : some parts were there because we wanted to make a point which wasn't in the textbook of the time, some were there as new developments, some represented topics we didn't have time to cover in any detail in the lectures, and some were our own private barrows which we like to push. The only consistency was in the insistence throughout on the top-down treatment; if the collection had a message, that was it.

Until 1995. Quite early in 1995, a few weeks before the 340 course lectures began, we learnt that our textbook had gone out of print. Panic struck. Should we choose a new book ? We'd chosen the old one (by Lane and Mooney[INT3]) because it fitted the course significantly better than any other we'd found, and we had been fairly discouraged by the rest. Our objection wasn't that they were specially bad books, but they were all written to much the same pattern, which wasn't what we wanted. We got in touch with one of the authors of our old text; he gave us to understand that it would rise again quite soon, which gave us a reason to avoid selecting a new text; it wouldn't help anybody to try to switch to a different book for just one year. Apart from that, even if we ordered a

new book immediately (which we didn't want to do – we'd certainly prefer to select one rather carefully), it would take some time to arrive.

So, quite suddenly, the notes grew into something much more like a textbook. It wasn't a new idea, but the absence of Lane and Mooney spurred us on to produce a rather quick rush job that would get us through 1995. It did, not too badly, but – being a quick rush job – fell short of perfection.

Then a few other factors complicated the picture. First we found that Lane and Mooney was rather unlikely to appear again. We also, encouraged by the experience, started discussions with a publisher on the possibility of converting the (now fairly extensive) notes into a textbook. And, in 1996, Alan Creak was away on sabbatical leave so wasn't involved in the course. Because of that, nothing much happened to the notes in 1996, but the experience of 1995 had suggested that they were not really sufficient yet to support the course on their own, so a new textbook was indicated. The book by Silberschatz and Galvin[INT4] was chosen, and we've stuck with it ever since.

And that brings us just about up to date. The 1998 notes are significantly advanced beyond the 1995 version; the textbook we'd like (L&M) is out of print; and the textbook we recommend (S&G) is of good quality, but not precisely what we want.

We don't think that is necessarily a bad thing. Operating systems has never been a clean and tidy subject in which there is one unambiguous and universally accepted true doctrine; there have always been differing views of what systems should be and how they should be designed and constructed. That being so, it is not unreasonable that you should meet a selection of views in the course. These notes present the top-down view which we have briefly described, but we include references to the recommended text where appropriate – and we've left in the old references to Lane and Mooney for good measure.

We'd like you to read them all, but accept that this is likely to be impracticable. We recommend, therefore, that you read the notes, and look up as much of the references in S&G as you can. Consider the different treatments given in the two cases, and you should find that you learn something from each of them. Some of the topics are not too well represented in S&G, and for those you might like to look at L&M.

MISSING BITS.

No collection of notes on operating systems could ever be complete, but we are aware of a few particular omissions. For example, there should certainly be more about distributed systems, system accounting, and system operations.

There is another omission which is inevitable. In the university context, we are bound to present "operating systems" as an academic subject – but if it stays that way it is useless. An operating system is first and foremost a practical artefact constructed to get computing work done. Every time you use a computer for anything more ambitious than machine code or as a doorstop, you use an operating system. The bit we can't do is to make that real to you. We hope that contemplating the questions we have asked might help, but the most effective way to learn about operating systems is to keep asking yourself as you use a computer, "What is the operating system doing now ?". What happens when I press a key on the keyboard ? Why does that character appear just there on the screen ? How does the cursor follow the mouse movement ? Where did that electronic mail come from, and how did it get here ? What information does the system need to do that job, and where does it come from ? It doesn't really matter if you don't end up with the "right" answers to these questions, but do try to end up with a *possible*

answer, and one which you could in principle implement. You will learn more that way than in any other.

ABOUT REFERENCES.

We give references to our sources where we can; follow them if you would like more information. Our references come in two sorts : those headed "COMPARE", which point to the two textbooks we mentioned earlier, and those headed "REFERENCES", which point to other sources, usually from the technical literature. Be aware that the textbook references are certainly not exhaustive; they identify some relevant parts of the textbook, but it's up to you to use the textbook effectively.

ABOUT QUESTIONS.

As well as informative material and references, the notes include occasional questions – but no answers. This is quite deliberate. The questions might have zero, one, or many answers; they are not intended primarily as exercises to be completed, but as stimuli which we hope will set you off along paths which are interesting but which we haven't space to fit into the text.

REFERENCES.

INT1 : M.V. Wilkes : "Software and the programmer", *Comm.ACM* **35#5**, 23 (May, 1991).

INT2 : K. Skytte : "Engineering a small system", *IEEE Spectrum* **31#3**, 63 (March, 1994).

INT3 : M.G. Lane, J.D. Mooney : *A practical approach to operating systems* (Boyd and Fraser, 1988).

INT4 : A. Silberschatz, P.B. Galvin : *Operating system concepts* (Addison-Wesley, fourth edition, 1994).

INT5 : S.E. Madnick, J.J. Donovan : *Operating systems* (McGraw-Hill, 1974).

QUESTION.

The checklist on the next page comes from a fairly recent publication[INT2]. How does it fit our approach to operating systems ? Have we left anything out ? (We know that the fine detail doesn't match – but you could usefully consider how you would rewrite the fine detail for an operating system.) (You might like to review this question from time to time through the course.)

Systems-engineering checklist

Here is a checklist of the steps essential to applying systems-engineering principles to the design of smaller commercial products:

Requirements phase
- Consult potential customers to ascertain their actual needs.
- Have a multidisciplinary project team take that statement of needs and use it to develop a detailed specification describing the product's functional requirements.

System-design phase
- Develop a system architecture that supports specified product requirements.
- Develop system-design specifications that document the system's architecture, system-level performance specifications, and the functional requirements of each subsystem and component.
- If necessary, use simulations and other analytical techniques to verify that top-level design concepts support all specified product requirements.
- Define the system's life-cycle cost model.

Detail design phase
- Design the hardware and software as described in the system-design specifications.
- Schedule several detail-design reviews to make quite sure that the hardware and software meet the specifications.
- Build and test each component to verify that the design objectives have been met.
- Develop plans for integrating the components and subsystems into the entire system, and for testing the system.
- Compare the actual costs of designing the hardware and software with the cost estimates to verify that cost objectives are being met.

System integration phase
- Integrate the components and subsystems into a prototype system and verify its functionality.

Design verification and optimization phase
- Verify that all performance specifications are met over all specified operating conditions.
- Optimize the system's design by minimizing any differences found between expected and measured performance.

System validation phase
- Evaluate the final product's configuration to ensure that it complies with the original functional-requirements specification.