Chapter 1: Why software engineering?

1. This type of situation could be indicative of a crisis in that the inadequacies of software contributed to the tragic loss of life. The fact that the pilot relied completely on the software to guide the plane turned a user interface error into a severe accident. In general, of course, aviation is better off because of software engineering because the high reliability of aviation software has made flying safer and more cost-effective. The situation that caused the accident is really a user interface problem. During development, more user feedback could have been designed into the software (so that the pilot might have been notified that he had typed in the code for Bogota, not Cali) and more usability testing could have been done (so that the misunderstanding might have surfaced before the software was installed in the airplane). The other issue is that of system boundary. Although the aeronautical charts used by pilots are outside the boundary of the computer system, the developers need to have knowledge of them and the codes and abbreviations they contain.

2. One example is the processing and synthesis of sensor data. If lots of real-time sensor data are flowing in to the software system, the hard part is getting the timing right so that an accurate picture of the overall system is obtained. For instance, if the software is controlling the shape of the wing of a plane, based on real-time sensor data about the air speed, pressure, temperature, etc., decisions must be based on up-to-date information from the sensors, even when flying through rapidly changing conditions (like a storm). The actual polling of the sensors is easy; the relationships are hard. The relationships are more complicated still when dealing with asynchronous changes to the wing shape.

3. Errors are misunderstandings that reside in the developer's thought processes. When that misunderstanding leads the developer to write something in a development artifact (specification, design, code, test data, etc.) that is not correct, that incorrect information is called a fault. When the fault causes the software to behave in an incorrect manner or produce incorrect results, that behavior is called a failure. An example of an error is when a developer mistakenly believes that the diameter of a circle is three times its radius. This error could lead to a fault in the requirements document if this developer writes a requirement that says that the system, when given a radius as input and a command to compute the diameter, should return three times the radius. Or, the error could lead to a fault in the design if the design for the module for converting radii into diameters includes the faulty formula. Or, if the formula relating radius and diameter is not specified in either the requirements or the design, a fault could still be introduced in the code if the developer encodes his/her erroneous understanding of the formula. If the fault is introduced in the requirements or the design and propagates to the code, then it may become a failure if that part of the code is exercised and consequently produces a wrong output. However, if the test data contained a fault that caused the code for computing the diameter not to be executed during test, then this would allow a failure to occur during operational use, when that code is exercised.

4. A count of faults is a misleading measure of product quality because there usually is not a one-to-one correspondence between faults and failures. If many faults are located in code that is never or rarely executed, then they are unlikely to result in many failures, which is usually a more relevant measure of quality. On the other hand, if just one fault is located in code that is exercised heavily in regular use, it could result in numerous failures, and thus low-quality software. Furthermore, neither a count of faults nor a count of failures gives an indication of the severity of the problems.

5. In one small division of NASA, a group of software developers who developed software to support satellite missions decided to build an object-oriented library of software components. It was estimated that such a library would drastically reduce development time in this division, and would even allow the physicists and flight dynamics analysts in the division to put together their own applications rather than relying on software developers to write the programs they needed for mission support. The library was built using an architecture based on many years of experience in this domain, using sound software engineering principles, and was tested extensively. The components in the library were of very high technical quality. However, the library was of limited value to its intended customers, the physicists and analysts, because there were no tools or documentation that facilitated its use. Users of the library had to have a deep understanding not only of the application domain, but also of object orientation and software development, as well as the structure of the library itself. Thus, the library was not the success it was hoped to be.

The Therac 25 case is a good example of the dangers of narrowing the definition of quality. The developers of the software no doubt had tested the software extensively and considered it to be high quality. However, what was relevant was the quality (in particular, the safety) of the system as a whole: software, hardware, and operator. Because the Therac 25 developers adopted such a narrow view of quality, lives were lost.

6. The advantages of using COTS software packages include lower development costs, and no responsibility for maintenance. The disadvantages include unknown reliability, risk of vendor dropping support or refusing to make needed changes or demanding unreasonable prices for future support, and the vendor's claims being misleading.

   Developers, customers, and users must anticipate the possibility of having to abandon the COTS product and having to replace it with locally written code or another COTS product. In other words, there must always be a contingency plan.

7. Ideally, the originators of the software that failed should be responsible for the consequences. However, it is often not possible to determine precisely what part of a system caused a failure, and it is often a combination of factors, or misunderstandings between the developers of the system itself, subcontractors, and/or COTS developers. One way to resolve the issue is to specify carefully such legal responsibilities ahead of time in the initial contracts. For example, the lead company might be willing to take on all liability for failure in exchange for additional cost savings from the COTS vendor or subcontractor. In this case, of course, the company will want extensive evidence of the quality of the software it is buying. Standard measures of quality (if they existed) would be helpful. At the very least, particular tests can be specified ahead of time that the software must pass before being accepted.

8. For example, the rule that advertisements for alcohol may be shown only after 9pm should be kept within the system boundary since the content and time of the advertisement are necessary pieces of information for the system to keep track of anyway and it would not be too difficult for it to check for conformance to this rule. On the other hand, it may become too complicated for the system to keep track of the content of advertisements in general. In that case, it could not include this constraint within its boundary.

   A similar argument could be made for the rule that if an advertisement for a class of product (such as an automobile) is scheduled for a particular commercial break, then no other advertisement for something in that class may be shown during that break. If advertisement content information can easily be tracked at the proper level of detail to enforce this rule, then it makes sense to keep it within the system boundary.

   Another rule states that if an actor is in a show, then an advertisement with that actor may not be broadcast within 45 minutes of the show. In order to include this constraint within the boundaries of the system, the system would have to keep track of all actors appearing in all programs and all advertisements. This information is considerable and not otherwise useful for the system to handle, so it would make sense for this constraint to be left outside the system boundary.

9. One reasonable way to assess the impact of a failure is how severely it affected people and how many people it affected. It is part of the job of the media to assess the impact in terms of the amount of damage (i.e. money or injury) and how much of society is affected (i.e. all taxpayers or the investors in a particular business). The Ariane-5 incident involved a large amount of money "owned" by many people (taxpayers). The failure of the Panix system, presumably, only affected its users and investors. Therefore, by this measure, the Ariane-5 incident had more impact and thus deserved wider coverage. However, of course, there are deeper issues. Perhaps a particular incident that had little impact itself may be indicative of problems in the same or another system that could cause much bigger problems later. It may be considered the job of the media to bring the public's attention to such potential problems. For example, the Panix failure could provide an opportunity to educate the public about the potential dangers of Internet "invasions."

Chapter 2: Modeling the process and life-cycle

1. Determining the process boundary is similar to determining a system boundary. One must decide exactly which activities, inputs, and outputs the process includes, and which are part of some other process. For example, in specifying a software development process, one must decide if it begins with writing the contract for the software, or if it assumes the contracting process has already taken place and the contract is finished.

   Also, both systems and process models can be decomposed into subsystems and subprocesses. Both can be decomposed in at least two different ways. Subsystems and subprocesses could be parts of the larger system or process which, when put together, constitute the whole. Or, the arrangement could be in layers, as in Figure 1.10.

   Both systems and processes can be modeled at different levels of detail and from different perspectives in order to understand them, and to discover their strengths and weaknesses.

2.

Waterfall:
- Benefits:
  - simple, familiar to most developers, easy to understand
  - easy to associate measures, milestones, and deliverables with the different stages
- Drawbacks:
  - does not reflect how software is really developed
  - not applicable for many types of development
  - does not reflect the back-and-forth, iterative nature of problem solving

V model:
- Benefits:
  - better spells out the role of different types of testing
  - involves the user in testing
- Drawbacks:
  - extensive testing may not always be cost-effective
  - some of the same drawbacks as waterfall

Prototyping:
- Benefits:
  - promotes understanding of problem before trying to implement solution
  - reduces risk and uncertainty
  - involves user in evaluating interface
- Drawbacks:
  - in systems where the problem is well understood or where the user interface is simple and straightforward, the extra time spent in prototyping is not warranted
  - prototyping can use up a lot of resources, especially if the prototype fails completely and must be scrapped

Operational specification:
- Benefits:
  - allows user and developer to resolve requirements uncertainty early on
- Drawbacks:
  - upfront investment may not be warranted if problem is well understood

Transformational:
- Benefits:
  - eliminates large steps of the process, thus reducing cost and opportunity for error
  - provides automatic documentation
- Drawbacks:
  - needs a very precise formal specification

Incremental or iterative development:
- Benefits:
  - reduces time to when customer receives some product
  - customer training can begin early
  - creates markets for new functionality

frequent releases allow problems to be fixed quickly
expertise can be applied to different releases

Drawbacks:

customer may not be satisfied with an incomplete product or with frequent changes to system
product may never be "complete"
problem may not be easily decomposable
changes may have to be made to completed parts in order to work with new parts

Spiral:

Benefits:

monitors and controls risks throughout process
easily incorporates prototyping

Drawbacks:

a lot of overhead

3. Waterfall: re-analyze requirements, redesign, recode, retest

V model: same as Waterfall

Prototyping: prototype change in design and code, iterating with customer

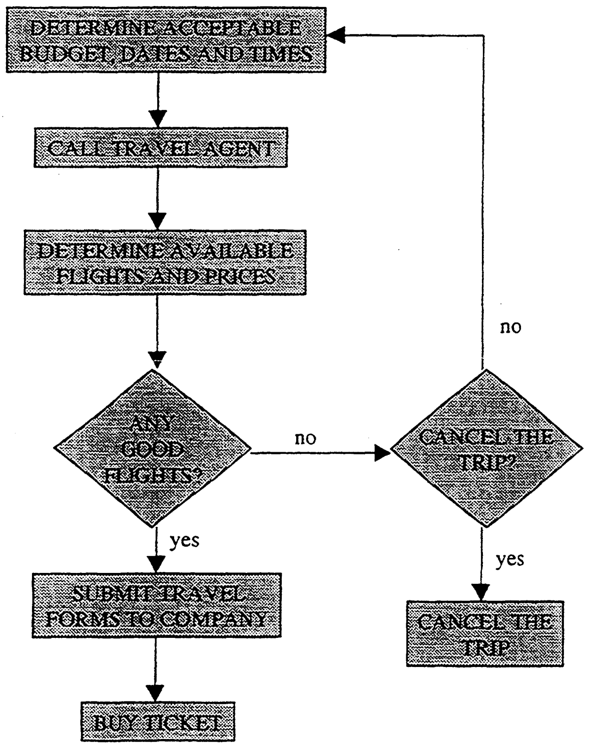
Operational specification: re-transform specification

Transformational: re-do relevant transformations

Incremental or iterative development: implement change in another increment or iteration

Spiral: implement another mini-spiral

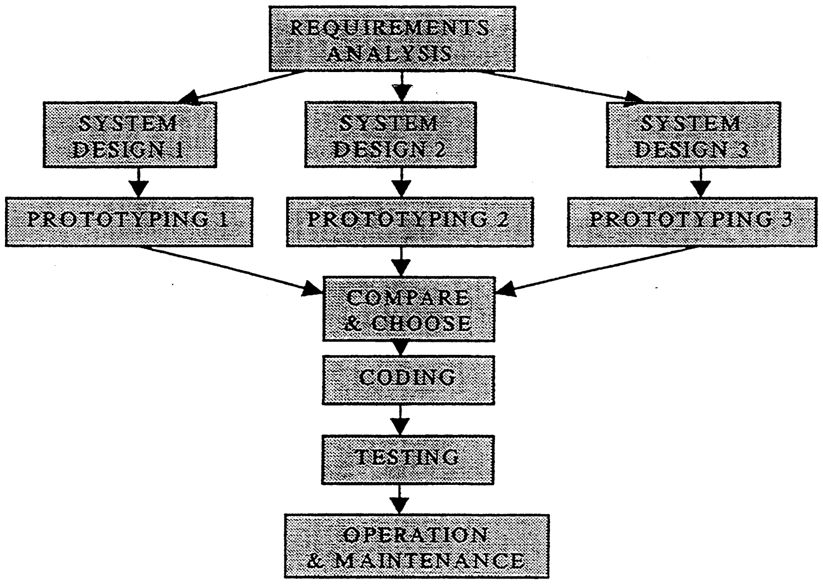
4.


DETERMINE ACCEPTABLE BUDGET, DATES AND TIMES
CALL TRAVEL AGENT
DETERMINE AVAILABLE FLIGHTS AND PRICES
ANY GOOD FLIGHTS?
no
CANCEL THE TRIP?
no
yes
SUBMIT TRAVEL FORMS TO COMPANY
BUY TICKET
yes
CANCEL THE TRIP

5.

Name	*Module*	
Synopsis	*This is the artifact that represents a class of software modules.*	
Complexity type	*Composite*	
Data type	*(module_c, user-defined)*	
Artifact-state list		
identified	*((state_of(module-spec(module)) = exists)* *(state_of(module.implementation) = nonexistent)* *(state_of(module.test) = untested))*	*Module is identified in the specification but has not been implemented yet.*
initial_implementation	*((state_of(module.implementation) = initial)* *(state_of(module.test) = untested))*	*Module has been initially implemented, but has had no testing.*
testing	*(((state_of(module.implementation) = initial) or* *(state_of(module.implementation) = revising))* *(state_of(module.test) = partial))*	*Module is being tested and bugs being fixed.*
complete	*((state_of(module.implementation) = final)* *(state_of(module.test) = complete))*	*Module is complete.*
Sub-artifact list		
	implementation	*The module's implementation.*
	test	*The process of testing the module.*
Relations list		
module-spec	*The part of the spec pertaining to the module.*	

6.



7. They would all be nice, but the only ones that are essential are that the model facilitates human understanding and communication and that it supports process management. Projects in which the problem and solution are not well-understood need to have as flexible a process as possible, which in most cases means that the process should not be defined at too detailed a level of granularity. In addition, as few resources as possible should be taken by purely process issues because they need to be devoted to understanding the problem and solution. Therefore, process improvement, guidance, and enactment should not be top priorities for the project. On the other hand, the model should facilitate following the process with as little effort as possible, so it should facilitate understanding and communication. Also, such projects can easily get out of control, so process management should be important.

8. Like manufacturing, there is a concern in software development with assuring the quality of the product, and many resources are spent in assessing and improving that quality. As well, both manufacturing and software development require careful planning and monitoring of the process. However, software development is creative in the sense that nearly every problem is new and requires a solution that is at least partly something that has never been done before. Therefore imagination must be used to envision how and if the solution will work.

9. The advantage of adopting a single process model for all software development in an organization is that it standardizes training, terminology, the collection of process metrics, planning, and estimation. This works well if all the organization's development projects are very similar in nature. If not, however, adopting a single standard process may unnecessarily constrain some projects from using the process that is best suited to the problem and solution.

10. Conformance to a particular process is often checked with the use of milestones. That is, the process is defined in such a way that there are tangible products at various points in the process whose existence indicates that particular process steps have been carried out. For example, when using the waterfall process, these intermediate products, or milestones, could be a requirements document, a design document, the code itself, test documents, etc. The timing of these products would indicate whether or not the process was being followed as planned. Another way to monitor use of a process is by measuring effort. Developers working on the project could be required to report the effort they spent on different process activities. By tracking when effort is spent on which activities, progress through the steps of the process could be monitored.

11. Both the incremental and iterative models provide a great deal of flexibility to accommodate requirements changes. They differ, however, in what types of requirements changes they handle best. The incremental model is good at handling requirements changes which add or delete whole areas of functionality because each increment adds to the system in terms of functions. So, if a new function is added to the requirements, its implementation can be planned as part of a future increment. If a function is deleted, it may not even have been implemented yet, and can be deleted from the plan for a future increment. On the other hand, the iterative model best handles requirements changes which modify functionality, rather than adding or deleting it. In iterative development, every function is implemented at the beginning, but refined through successive iterations. Since most functions are modified in each iteration anyway, it is usually not difficult to incorporate modifications due to requirements changes.

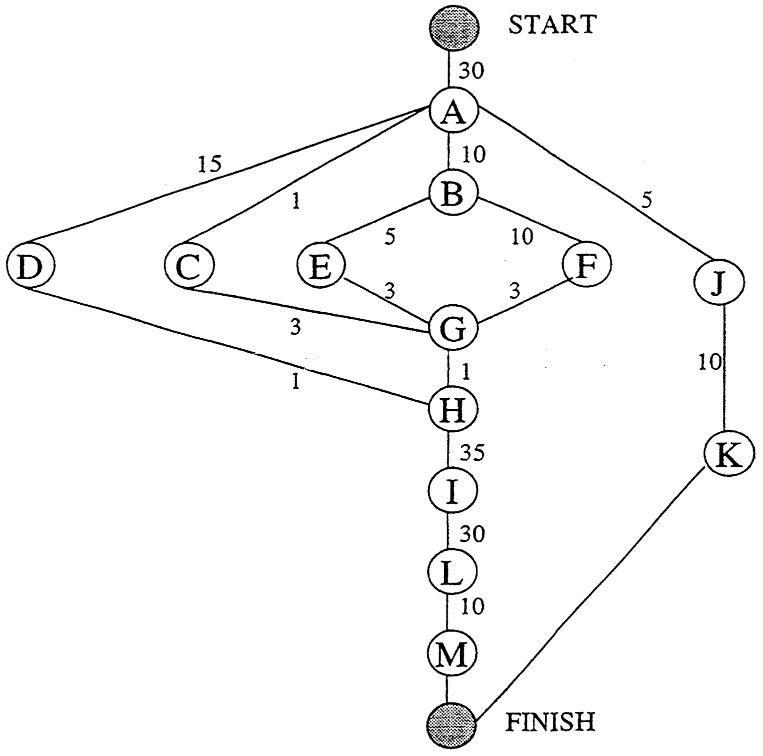
12. The main issue is who is responsible for the failure of the software, and for any reparations necessitated by that failure. Ethically, you should have done everything in your power to ensure that the software was defect-free. That would include performing code reviews because you, as a software developer, know that code reviews are a valuable tool in locating software faults. Legally speaking, however, you were not required to perform code reviews. Moreover, you may have been prevented from performing those code reviews, either by a lack of resources provided by Amalgamated, or even by the terms of the contract, which may prohibit all process activities not specifically prescribed. In that case, Amalgamated would at least share responsibility for the software failure.

Chapter 3: Planning and managing the project

1.

Activities:	**Duration (in minutes):**
A. Choose recipe	30
B. Assemble cake ingredients	10
C. Prepare cake pan	2
D. Preheat oven	15
E. Mix dry ingredients	5
F. Mix wet ingredients	10
G. Fold wet ingredients into dry ingredients	3
H. Pour batter into pan	1
I. Bake cake	35
J. Assemble icing ingredients	5
K. Mix icing	10
L. Cool cake	30
M. Apply icing	10

Activity Graph:



Critical Path: A-B-F-G-H-I-L-M 129 minutes

2.

Activity leading to:	Precursors
A	
B	A
C	A
D	A,B
E	A
F	A,C
G	A,E
H	A,C,E,F,G
I	A,B,D
J	A,B,D,E,G,I
K	A..J
L	A..K

Activity leading from .. to:	Earliest start time	Latest start time	Slack
A..B	1	1	0
B..D	4	4	0
B..I	4	5	1
D..I	9	9	0
I..J	11	11	0
A..C	1	5	4
C..F	6	10	4
F..H	9	13	4
A..E	1	4	3
E..G	5	8	3
G..H	8	14	6
G..J	8	11	3
H..K	10	14	4
J..K	13	16	3
J..L	13	13	0
K..L	15	18	3
L	21		

Critical path: A..B..D..I..J..L 20 days

3. **Critical path**: A..B..C..E..D..I..K..L 24 days

4. Test planning can take place in parallel with requirements, design, and coding activities. Low-level design and coding on different parts of the system can take place in parallel. Activity graphs might

hide interdependencies because you cannot represent the fact that the results of a particular activity may require re-doing part of a previous activity. Also, a complex interdependency between two activities may not require that one be completed before the other. There are other types of interdependencies, e.g. information dependencies.

5. Adding personnel requires extra time for training as well as a superlinear increase in communication channels and thus communication costs. These added costs may outweigh the time savings from having extra people working on the project.

6. The Hardand estimate would be $5.25 * (20000)^{.91} = 43{,}062$ person-months. If the size estimate is 10% too low, then it will take 4,332 more person-months to complete. If we express k as a fraction, rather than a percentage, then the relationship between the original estimated size, S_e, and the actual size, S_t, is

$$S_t = S_e / (1-k)$$

and the relationship between the original estimate, E_e, and the actual needed time, E_t, is

$$\begin{aligned} E_t &= 5.25\, S_t^{.91} \\ &= 5.25\, (S_e / (1-k))^{.91} \\ &= (1/(1-k))^{.91} * 5.25\, S_e^{.91} \\ &= (1/(1-k))^{.91} * E_e \end{aligned}$$

If k is 10% (or 0.1), then the above formula yields 47,395 person-months, which is 4,332 more than the original estimate.

7. A utility program usually takes longer to develop than an applications program because the developer has to take into account all of the different environments in which it might be used. For instance, a sort routine must be callable from many different other programs, and it has to return its output in some "generic" way that can be used by an assortment of other components. On the other hand, an applications program is usually very specific. It has a much narrower scope, and the set of possible calling programs is much smaller. Its output is usually tailored for the very small number of other programs that need it. A systems program, being even more general than a utility program, has a larger set of possible callers, and it must be even more generic. So it can take a longer time to specify the set of calling and receiving programs, and it is likely to have a lot of cases to handle, especially in its error-handling.

8. Factors to consider include:

   - how long it would take to build in-house;
   - how soon the software is needed
   - how many programmers would be needed;
   - whether or not the technical and managerial expertise is available in-house;
   - the quality of the purchased software;
   - maintenance costs, both in-house and through the vendor

   It is generally better to build when the expertise is already available in-house, when the organization is willing and able to take on maintenance of the system, or when there is uncertainty about whether or not the purchased software would meet quality requirements. It is generally better to buy when the software is needed sooner than it can be built in-house, or when it can be bought much more cheaply than it can be built.

9. Yes and no. Development time, strictly defined, may be shortened by adding more people, but actual project time would probably increase due to increased training, communication, and management effort. It also depends a great deal on when in the project lifecycle people are added. A project that starts out with more people will probably finish earlier than if it had started out with fewer people. However, adding those people nearer to the end of the project is less likely to be beneficial.

10. The Bailey & Basili model reflects the first factor through the cost factor "customer initiated program design changes." COCOMO Stage 3 addresses the latter through the factor "personnel continuity."

11. Teams of three students each implemented an automatic car wash control system in 90 days as part of a software engineering course. Below are some of the risks, the corresponding exposures, and possible mitigating actions.

Risk	Exposure	Mitigating actions
unable to agree on design	(.30)(10) = 3 days	pick one chief designer who makes all decisions
one member not working	(.10)(30) = 3 days	choose team members; appeal to professor
many bugs found in testing	(.40)(5) = 2 days	inspections

12. Unit size per unit time is an unsatisfactory measure of productivity for several reasons. Unfortunately, it is pretty much the only measure available. Some of the reasons it is inadequate:

- programmer output varies widely
- size per time varies according to application language, application domain, difficulty, and newness of problem
- amount of reuse also affects productivity
- management style and structure
- programmer style can vary according to the targets given to them
- values for size are not available until long after estimates are needed, and size estimates can be very inaccurate

Consequently, it would be very helpful to have a measure for productivity that overcame all of these shortcomings. Furthermore, we need to understand some of the surrounding issues, such as the effect of productivity targets, management style, and programmer expertise on productivity, no matter how it is measured.

Chapter 4: Capturing the Requirements

1. Developers, customers, and users all bear some responsibility for the situation, as they all had input into the requirements of the system. However, the developers (or the development organization) should probably bear most of the responsibility because they, as software professionals, should have the expertise to ensure that the requirements are complete.
2. No system can be built to be safe and reliable in all environments and under all conditions. Therefore, we need to explicate any assumptions we make about the environment, about input values, and about the ordering of inputs. Such non-functional requirements must explicitly specify all of the situations and conditions under which the system must not fail. The context and the environment in which the system will be running become pre-conditions for stating and satisfying its safety and reliability requirements. For example, "the system must not fail catastrophically when the user supplies a non-valid numerical input." Ideally, the requirement should also specify what the system should do in such a situation (e.g. give a meaningful error message to the user and continue processing). The problem with specifying the requirement to "never fail" is that it does not explicitly state all of the situations in which the system must not fail, therefore it is not testable or demonstrable. Confidence that a system meets this requirement can be built up, however, by testing under as many conditions and in as many different situations as possible, and by soliciting input from users and other experts about the different situations that could be tested.
3.

   a) The client daemon must be invisible to the user

   Design constraint, referring to distributed solution

   Functional requirement related to user interface

   b) The system should provide automatic verification of corrupted links or outdated data

   Functional requirement

   c) An internal naming convention should ensure that records are unique

   Functional requirement related to data constraints

   d) Communication between the database and servers should be encrypted

   Quality requirement referring to the security of the system

   e) Relationships may exist between title groups [a type of record in the database]

   Functional requirement referring to data relations

   f) Files should be organizable into groups of file dependencies

   Functional requirement related to data constraints

   g) The system must interface with an Oracle database

   Design constraint referring to the interface with other systems

   h) The system must handle 50,000 users concurrently

   Quality requirement referring to the performance of the system

   Items d and f may be premature design decisions. They could be rewritten as:

   d) Internal data stores and communications should be secure against illegal accesses

   f) Dependencies among information should not be cyclic

   In addition, items a and c refer to design constructs (e.g., clients, internal naming conventions). These requirements could be better expressed by eliminating reference to these constructs:

   a) User should believe that he is interacting with a centralized system

   c) All records must be unique

4. Decision table:

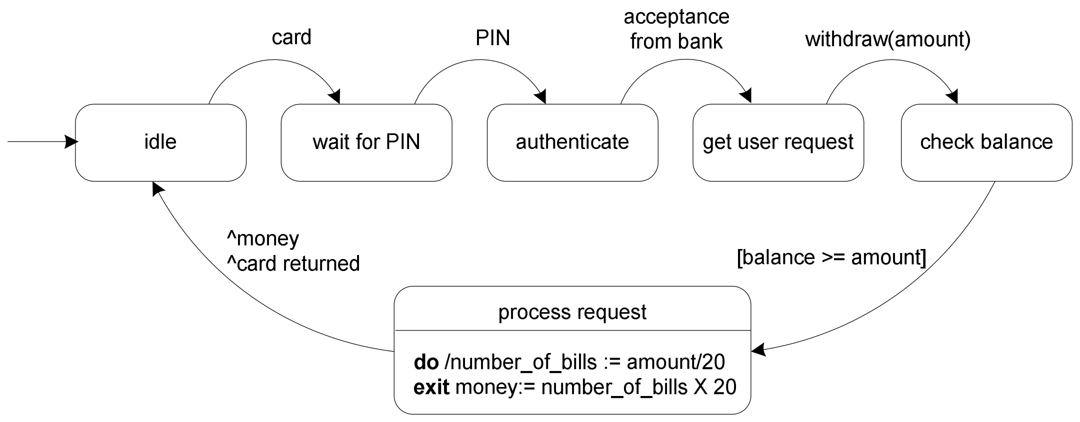
	R1	R3	R4	R5	R6	R8	R9	R10
black's turn	T	T	-	-	F	F	-	-
black has a legal move available	T	F	-	-	-	-	-	-
red has a legal move available	-	-	-	-	T	F	-	-
black marker reaches red side of the board	-	-	T	-	-	-	-	-
red marker reaches red side of the board	-	-	-	-	-	-	T	-
no red markers on the board	F	F	F	T	F	F	F	F
no black markers on the board	F	F	F	F	F	F	F	T
black's turn over		X						
red's turn over						X		
red makes a move					X			
black makes a move	X							
black marker is king'ed			X					
red marker is king'ed							X	
game over; black wins				X				
game over; red wins								X

5. The specification is contradictory if there are two columns which specify the same condition (same set of truth values) but different actions. The specification is ambiguous or incomplete if there are some conditions for which there is no column (i.e. no specified action).

6.

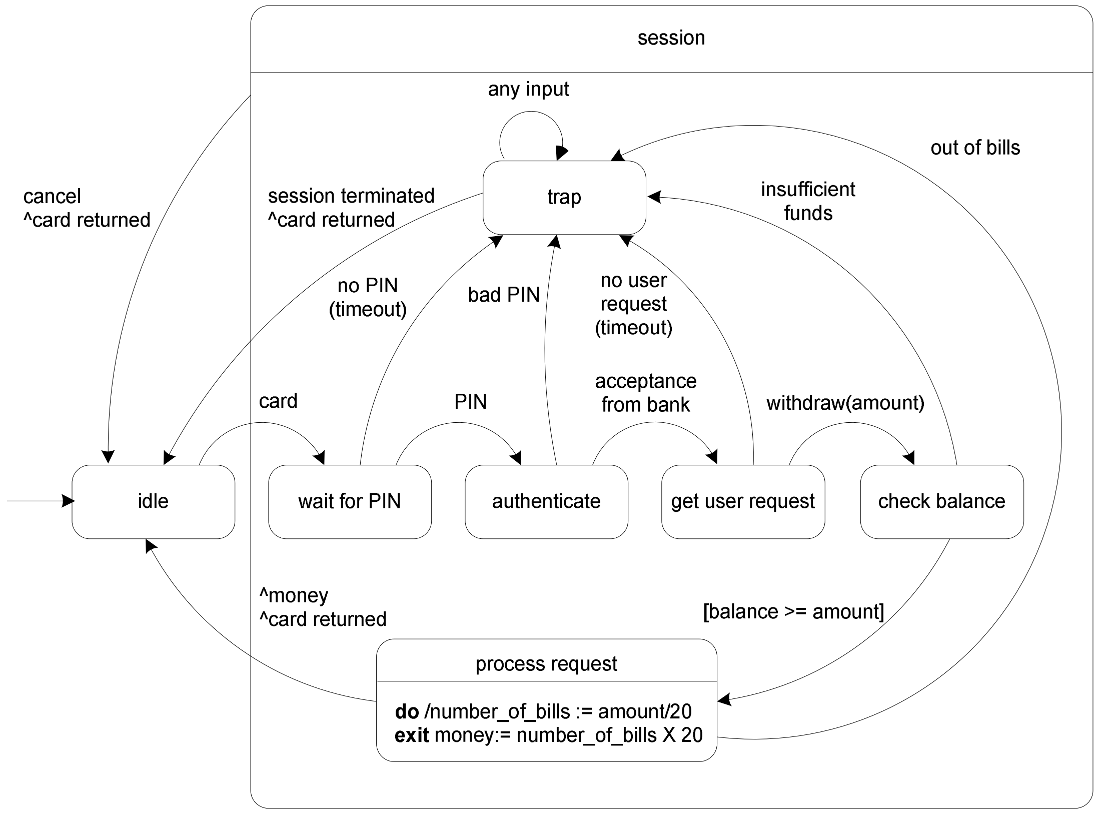
	$b^2 - 4ac < 0$	$b^2 - 4ac = 0$	$b^2 - 4ac > 0$
quadratic_real_roots (a,b,c) =	×	$\frac{-b}{2a}$	$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

7. A state-machine specification to illustrate the requirements of an automatic banking machine (ABM).



Note: This state machine shows only the operation of withdrawing cash. Other operations can be modeled in a similar way.

8. A state-machine for the ABM including a trap state.



9. Safety properties:

- The amount of money returned by the ABM is never more than the amount requested by the client in the withdrawal.

  $\Box\,(\text{money} \Rightarrow (\text{money} \leq \text{amount}))$

- The amount of money returned by the ABM is never more than the client’s balance.

  $\Box\,(\text{money} \Rightarrow (\text{money} \leq \text{balance}))$

Liveness properties:

- When an authenticated client specifies the amount of money to withdraw, and the client has sufficient funds in his account, and the ABM has sufficient funds to dispense, the ABM dispenses the requested amount of money

  $\Box\,((\text{process req} \wedge \text{amount} \leq \text{balance} \wedge \text{amount} \leq \text{ABM funds}) \Rightarrow O(\text{money} = \text{amount}))$

- When the client terminates the session pressing the cancel button, the ABM returns his card

  $\Box\,((\text{session} \wedge \text{cancel}) \Rightarrow O\ \text{card returned})$

10.

Proof 1 : The amount of money returned by the ABM is never more than the amount requested by the client in the withdrawal.

<u>PROOF</u>	<u>RATIONAL</u>
a. money	*given in formula*
b. ABM dispenses $20 bills	*assumption*
c. number_of_bills = amount/20	*action in state process request, a*
d. ABM has sufficient cash	*assumption*
e. money = number_of_bills x 20	*action in state process request, c*
f. money = (amount/20) x 20	*substitution b in d*
g. money <= amount	*number_of_bills is integer part of amount/20*

Assumptions about the environment

ABM dispenses $20 bills

ABM has sufficient cash

Proof 2: The amount of money returned by the ABM is never more than the client's balance:

PROOF	RATIONAL
a. money	*given in formula*
b. ABM dispenses \$20 bills	*assumption*
c. number_of_bills = amount/20	*action in state process request, a*
d. money <= number_of_bills x 20	*action in state process request*
e. money <= (amount/20) x 20	*substitution of b in c*
f. amount <= balance	*guard on entering state process request*
g. money <= balance	*transitivity of <=*

Assumptions about the environment

ABM dispenses \$20 bills

Proof 3:

PROOF	RATIONAL
a. in state process request	*given in formula*
b. number_of_bills = amount/20	*action in state process request*
c. amount = (amount/20)x20	*assumption*
d. ABM has sufficient cash	*given in formula*
e. money = number_of_bills x 20	*action in state process request, c*
f. money = amount	*substitution of a, b into d*
g. O (money=amount)	*no guard on transition leaving state process request*

Assumptions about the environment

Amount request is divisible by 20

Proof 4:

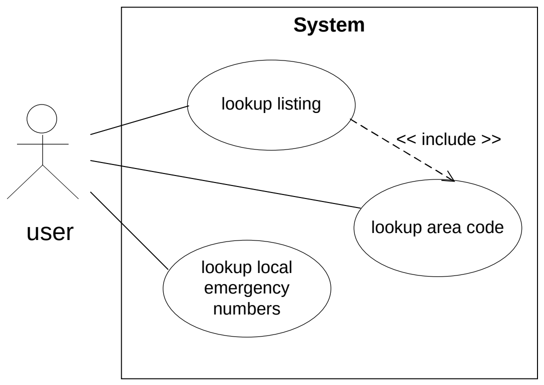
PROOF	RATIONAL
a. session	*given in formula*
b. cancel $\Rightarrow$ O(card returned)	*guard and action on transition leaving*
c. amount = (amount/20)x20	*state session*

11. Some factors to consider in this decision are

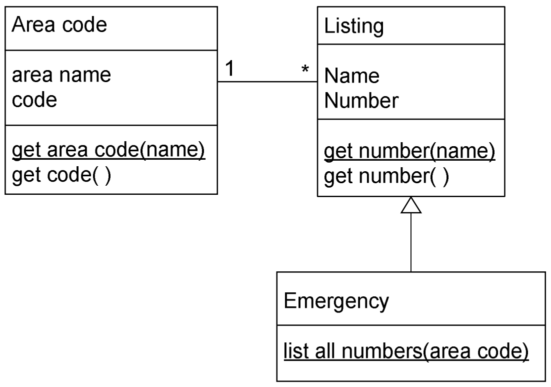
- Who will be building the prototype? If they have a good understanding of the requirements, then they could write the requirements after the prototype has been evaluated. Otherwise, they will need some guidance or documentation, and a draft of the requirements should be written by someone else so that the prototype builders can use them.

- How tight are the schedule and budget? If the requirements are written before the prototype is developed, they will likely have to be substantially modified after the prototype is evaluated.
- How much is known about the requirements before the prototype is evaluated? If little is understood about the requirements, then it does not make sense to try to write them down. They will be clarified through the evaluation of the prototype(s) with the customer.

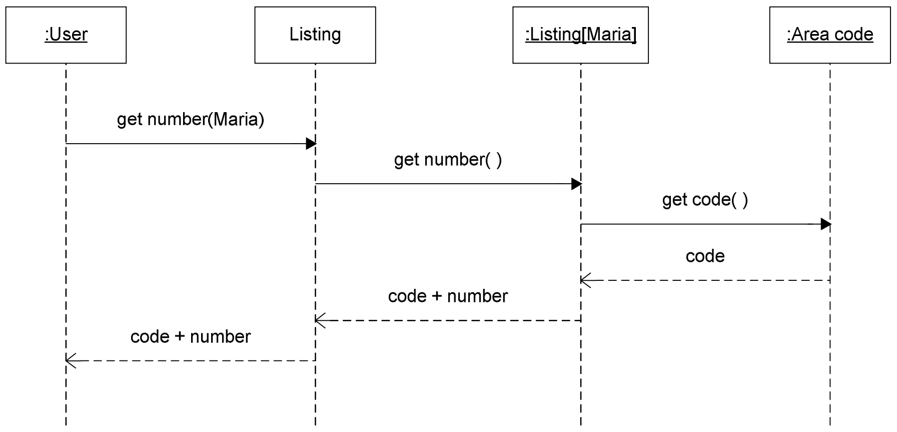
12. UML use-case diagram for an on-line telephone directory.



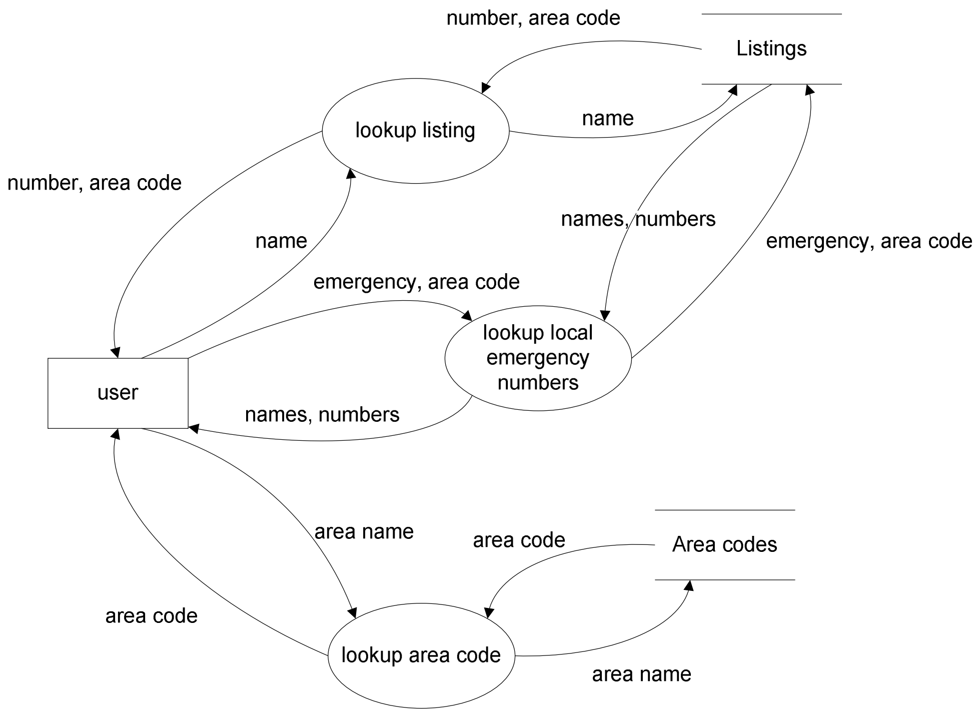
UML class diagram. Note that this is one way the objects might be organized.



UML sequence diagram showing the realization of the 'lookup listing' use case.



13. Data-flow diagram to illustrate the functions and data flow for the on-line telephone directory system.



14. Although the two are related, separating functional flow and data flow often simplifies the specification of the system by dealing with just one source of complexity at a time. Also, the two types of flow often correspond to different development tasks (e.g. writing computational subroutines vs. designing the interfaces between modules).

15. The major difficulty with specifying real-time systems is specifying feasible timing constraints for computations and communications. The order of inputs to a real system is often nondeterministic, and so timed responses for every input combination must be considered in the specification.

16. Most users think of requirements in terms of features, which describe new functionality. Thus, functional specifications are often easier to read and write. The primary advantage of object-oriented specifications is that they are supposed to be easier to change. OO specifications organize data and operations on data into separate classes, such that most requirements and design changes can be realized by local additions or changes to the model, new methods, or new sequences of method calls.

17.

```
┌─ Schedule ─────────────────────────────────────────────
│ presentation: Presenter ⇸ Date
├────────────────────────────────────────────────────────
│ ∀ d: Date | # { p: Presenter | presentation(p) = d • p } ≤ 4
└────────────────────────────────────────────────────────
```

```
┌─ AddPresentation ──────────────────────────────────
│ Δ Schedule
│ p?: Presenter
│ d?: Date
├────────────────────────────────────────────────────
│ ∀ d: Date | # { p  | presentation(p) = d • p } < 4
│ presentation′ = presentation ⊕ { p? ↦ d? }
└────────────────────────────────────────────────────
```

```
┌─ RemovePresentation ──────────────────────
│ Δ Schedule
│ p?: Presenter
├───────────────────────────────────────────
│ presentation' = { p? } ⩤ presentation
└───────────────────────────────────────────
```

```
┌─ SwapDates ─────────────────────────────────────────────────────────────
│ Δ Schedule
│ p1?, p2?: Presenter
├─────────────────────────────────────────────────────────────────────────
│
│ presentation' = presentation
│    ⊕ { p1? ↦ presentation(p2?) }  ⊕ { p2? ↦ presentation(p1?) }
└─────────────────────────────────────────────────────────────────────────
```

```
┌─ ListPresentations ───────────────────────────
│ Ξ Schedule
│ d?: Date
│ l!: ℙ Presenter
├───────────────────────────────────────────────
│ l! = { p: Presenter | presentation(p) = d? • p }
└───────────────────────────────────────────────
```

```
┌─ ListDate ────────────────────────────
│ Ξ Schedule
│ p?: Presenter
│ d!: Date
├───────────────────────────────────────
│ d! = presentation(p?)
└───────────────────────────────────────
```

```
┌─ SendReminders ─────────────────────────────────────────────────
│ Ξ Schedule
│ today?: Date
│ peopleToRemind!: ℙ Presenter
├─────────────────────────────────────────────────────────────────
│  peopleToRemind! = { p: Presenter | presentation(p) = today? • p }
└─────────────────────────────────────────────────────────────────
```

18.

```
isOnLoan(New, i) ≡ false;
isOnLoan(borrow(lib, i), i2) ≡ if (i=i2) then true;
                               else isOnLoan(lib, i2);

isOnLoan(buy(lib, i), i2) ≡ if (i=i2) then false;
                            else isOnLoan(lib, i2);
isOnLoan(reserve(lib, i), i2) ≡ isOnLoan(lib, i2);
```

```
unreserve (New, i) ≡ New;
unreserve(buy(lib, i), i2) ≡ if (i=i2) then buy(lib, i);
                             else buy(unreserve(lib, i2), i);
unreserve(borrow(lib, i), i2) ≡ borrow(unreserve(lib, i2), i);
unreserve(reserve(lib, i), i2) ≡ if (i=i2) then unreserve(lib,i2);
                                 else reserve(unreserve(lib, i2), i);

isOnReserve(New, i) ≡ false;
isOnReserve(reserve(lib, i), i2) ≡ if (i=i2) then true;
                                   else isOnReserve(lib, i2);
isOnReserve(buy(lib, i), i2) ≡ if (i=i2) then false;
                               else isOnReserve(lib, i2);

isOnReserve(borrow(lib, i), i2) ≡ isOnReserve(lib, i2);
```

Note: The unreserve operation is supposed to cancel all previous reservations. Thus, even if unreserve finds a matching reservation, it continues to search the rest of the sequence of library operations, looking for more matching reservations.

19. In most cases, an application domain-specific checklist is preferable. However, some things that would be included in any checklist are:
- For each function described, are all conditions under which the function can be invoked specified?
- For each function described, are all conditions under which the function can terminate specified?
- For each user input, are appropriate responses specified for invalid input?
- For each output, is the format specified in all situations?
- Are there any ambiguous statements (e.g. the use of "can" or "should")?
- Are there any "to be specified" features?
- Are all statements understandable?
- Is each requirement specific enough to be testable?

20. Yes, they could be merged if the customer is familiar with software development and is to be highly involved in the entire software development process. In this case, merging the documents reduces effort and the amount of documentation to keep track of. Otherwise, the benefit of having two documents is that it provides documentation for communicating with the customer as well as documentation from which to develop a system design. Separating the two helps keep them clear and understandable to their two separate audiences.

21. Specifications that use formal methods could be compared with specifications developed using other methods meant to ensure thoroughness, such as the use of requirements reviews.

22. Some factors to consider:
- If a customer consistently ends up with systems with less functionality than specified, how will that affect his or her view of the software provider? of the software industry?
- If a customer has unrealistic ideas about what software can deliver, is it up to the software provider to educate the customer? Can the customer be expected to know what is realistic? Is the customer expecting to be educated in this way?
- Say that you decide to be honest and tell the customer that you cannot implement the requirement. What if the customer cancels the contract after finding another software vendor that claims the system can be implemented?

Chapter 5: Designing the Architecture

1. The NIST/ECMA model is closest to the layered architectural style because it is organized into parts where each part (payer) communicates principally with the parts on either side of it.

2.

   Pipe and filter: compilers, with the filters being the lexical analysis, parsing, semantic analysis and code generation processes.

   Client-server: The internet, with websites being hosted on web servers and browsers (clients) being used to access the sites. E-mail systems, with the mail clients communicating with the POP/IMAP mail servers. Multiplayer on-line games with the games being hosted on a central server and the players (clients) connecting to it.

   Peer-to-Peer: VoIP and instant messaging applications, where the peers communicate directly, without the need for a communication channel managed by a central server.

   Publish-subscribe: Graphical user interface-based applications, where different parts of the system provide functionality through notification of user interface events such as clicking a mouse button or placing the mouse cursor over a determined area of widget.

   Repositories: Business intelligence applications, where huge amounts of business and operation-related data are stored in a data warehouse. This data is then made available to be accessed by different business units for processing and analysis.

   Layered: Modern web browsers, where the HTML rendering engine, network protocol handling and operating system interface are the different layers in the architecture.

3. The repository design has high coupling because of the shared data, and has high cohesion since each module is responsible for one function of the system. The data abstraction design has lower coupling among components than the repository due to the use of interfaces to retrieve the necessary data. It has lower cohesion than the repository design since the primary functions are spread among the modules. The implicit invocation design has lower coupling than the previous designs due to the use of ADTs, and high cohesion as it separates the primary functions into separate modules. The pipe and filter design has the low coupling because the dependence between the filters is the pipe between them and has high cohesion because each independent filter relates to a primary function of the system.

4. Prototyping processing intensive applications does not result in saving any significant amount of development time. There is no way to create a prototype for such systems without first fully implementing the necessary operations and processes. Take a mathematics equation solving application as an example. Without the equation parsing and solving algorithms, the prototype would be used only to portray the user interface and would not result in any significant change in development time.

5. Systems with high user interaction benefit from prototyping by providing feedback on usability issues. Systems with complex protocols or transactions also benefit from prototyping by testing the implementation of such protocols and transactions.

6. Application generators are, conceptually, an advanced form of reuse. Modularity facilitates reuse by breaking functionality into pieces that can be manipulated and tailored individually in new contexts. This is even truer for application generation because the application-specific information used by a generator must be organized in small pieces so that it can be applied to one piece of functionality at a time.

7. In a shared data architecture, the components of the system manipulate the data directly and, as a result, are highly coupled to the data structure. In order to reuse the components of said system, it would require either having the same data model as the component or changing the component to operate with a new data model. Having the same data model is not always a viable option, as even though the systems may be similar the data may be completely different. Modifying the component to work with a new data model would probably take as much effort as writing a new component with the desired functionality.

8. The characteristics should include:
   a) cohesion
   b) coupling
   c) error, exception, and fault handling
   d) reusability
   e) modifiability
   f) clarity
   g) modularity

   For an operating system, probably the most important criteria would be error, exception and fault handling. Modifiability would also be important as operating systems tend to be used and maintained over long periods of time. Error, exception and fault handling would also be important for a word processor because of the user interactions. Reusability might also be important if a product line of word processors is envisioned. The same is true for a satellite tracking system, as such a system for a different satellite could reuse much of the same code. Cohesion, coupling, clarity and modifiablity are always important because they contribute to the ease of implementation.

9. Answers to this question may vary.

10. Questions (c), (d) and (e) deal with architectural design decisions. The types of report will affect how the reporting module should be designed as well as how it interacts with other modules. Both questions (d) and (e) deal with support for concurrency which should be addressed at the architectural level such as "will the application need to support more than one user at a time?", "will there be more than one instance of the application running at a time?" Question (a) deals with platform specific details such as computer architectures and should not be dealt with during the architectural design. Question (b) deals with the look and feel of the application and user interaction, and has no influence on architectural decisions. Similarly, question (f) deals with details of the depreciation algorithm implementation and do not contribute to the architectural design.

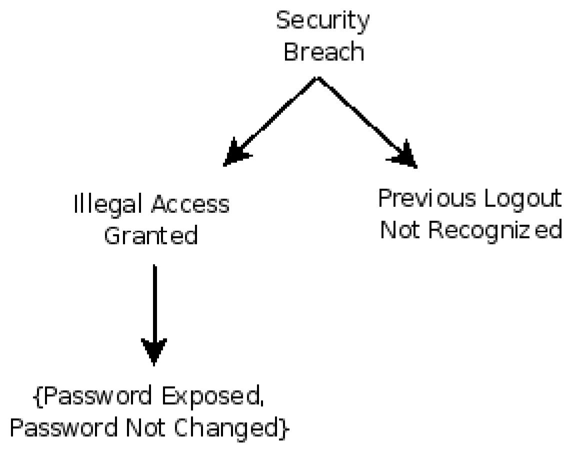
11.

   a) Use a repository software architecture. The central data store (bank server) contains the information about all the accounts in the bank. Each automated banking machine is a data accessor, able to retrieve account information for a given account and either withdraw or deposit cash into the account.
   b) Use a publish-subscribe software architecture. Each user is able to subscribe to the desired news bulletins, and will receive a notification from the news feeder whenever a new story is posted.
   c) Use a pipe-and-filter software architecture. Each different operation refers to a different filter (rotation, color tinting, cropping, etc) that can be combined ad-hoc to process the picture as desired.
   d) Use a client-server software architecture. Each sensor is a client that connects to the forecasting application (central server), uploading data as needed.

12. One strategy to improve performance is to do all of the authentication and processing in the automated banking machines, and only sending confirmed transactions to the bank server. This would free the bank server from having to deal with authentication, processing and confirmation of requests. On the other hand, improving security would require the bank server to validate all of the processes from the automated banking machines, to prevent compromised automated banking machines from requesting erroneous transactions. Improving performance have a negative effect on security and vice-versa.

13. One way to detect faults would be to identify exceptions in the sensors and their data. The sensors send data periodically, if a sensor has not sent data over a certain period of time then it is faulty. Defining expected ranges for the data can also help in identifying faults, if a sensor is sending .

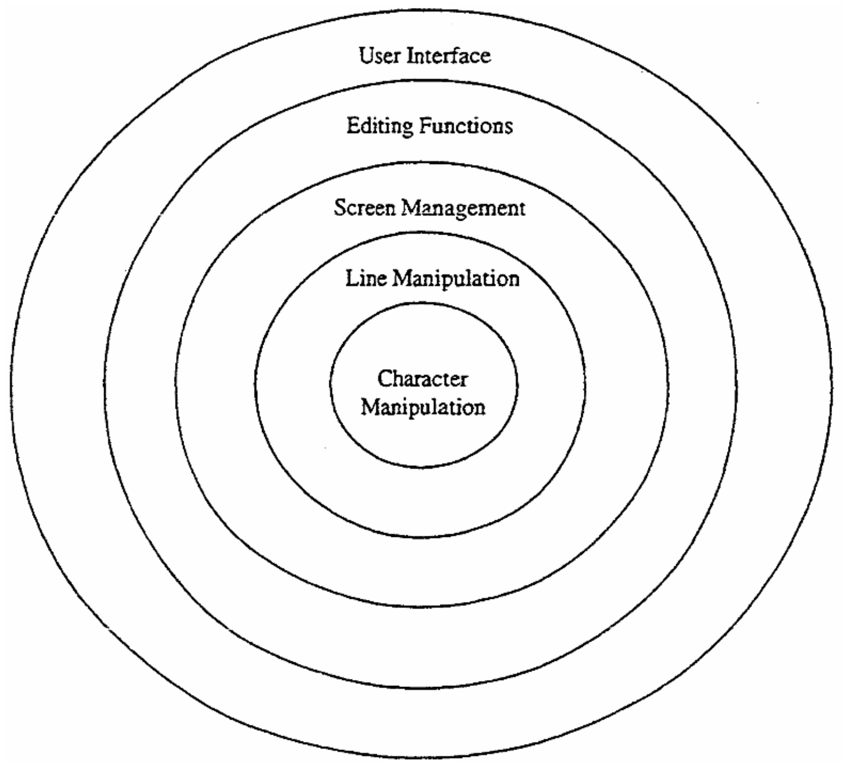
14. The cut set tree is shown below.


Security Breach
Illegal Access Granted
Previous Logout Not Recognized
{Password Exposed, Password Not Changed}

15. It affects the initial cost-benefit analysis by changing the value added by the bin-based indexing. The new value can be calculated by solving the equation
(150 requests/second - 60 requests/second) x 2000/year = $180,000/year.
This modification does not alter the result of the cost-benefit analysis, as design 2 is still the one with the higher ROI and lower payback period.

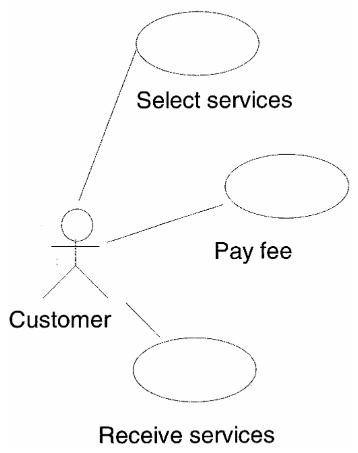
16. The common graduation requirements for three different disciplines at the (fictional) CGNU are a required minimum average, a minimum number of courses in the area of the discipline, with a subset in specific areas and minimum number of elective courses outside the area of the discipline. The differing requirements are participation in the co-op program for a minimum number of terms for discipline A, enrollment in at least one of the offered lab specializations for discipline B and a final year dissertation for discipline C. These requirements can be combined into a product line, with the common requirements providing the core functionality common to all systems in the family. This product line allows each instantiated application to specialize in a set of discipline specific requirements.

17. The architectural design of the text editor is shown on the next page. The design attempts to reduce coupling by having components in each layer communicate only with each other and with components in adjacent layers. However, there is likely to be a lot of dependency between these components. Other issues to consider when evaluating a more detailed design are cohesion, modularity, complexity, functionality, error and exception handling, and fault handling.


User Interface
Editing Functions
Screen Management
Line Manipulation
Character Manipulation

Chapter 6: Designing the Modules

1. Answers to this question may vary, since some details for the specific scenarios will be filled in by the reader. The number of distinct use cases is also subjective, as long as the use cases in combination describe all of the car wash functionality specified. One solution is as follows:



The "select services" scenario begins when the customer drives his car up to the control panel. The system displays on the control panel is listing of the different levels of service available and the price for each. The customer initiates a transaction by pushing a button on the control panel corresponding to a particular service. The system then prompts the customer to indicate (by pushing a button on the control panel) the type of car he or she is driving (e.g. compact, midsize, minivan, truck), or to cancel. if the customer selects "cancel", the system resets and the customer can select a different service, if desired.

The "pay fee" scenario begins when the customer has finished selecting the services. The system then computes the fee and displays the amount on the control panel. If the customer does nothing, then after a short period of time the system resets itself. Otherwise, the customer inserts money into the money reader, which is responsible for reporting to the system the value of the money just entered. [Note that because the car wash is to be an automated system, we assume the existence of an external system for accepting customer payment.] The system compares the value of the money just entered to the fee remaining to be paid. If the amount just entered is greater, then the system instructs the money reader to return the proper amount of change; otherwise, the system calculates the remaining fee and instructs the money reader to display this amount on the control panel. When the fee has been completely paid the system is ready to dispense services.

The "receive services" scenario begins when the customer has completely paid the fee. If the car wash is currently busy, the control panel indicates that the customer should wait. If the car wash is empty, or when the car wash becomes empty, the control panel instructs the customer to drive forward. When the sensors indicate that the customer has driven forward to the appropriate position, a “stop” light is illuminated inside the car wash. The system then begins to dispense services. Depending on the type of vehicle indicated by the customer, particular hoses and nozzles may or may not be activated. Once services have been dispensed, the “stop” light is no longer illuminated. The customer drives out of the car wash.

2. Lorenz and Kidd’s specialization index is a measure of how much of the functionality of a class is specific to that class rather than inherited by the class from elsewhere in the inheritance hierarchy. It may help to think of the metric being defined as (NOO/total class methods) * the level of the class in the inheritance hierarchy. The first term then measures what percentage of the methods available to the class are specifically defined in the class itself. A low value for this term means that very few of the class’ inherited methods were overridden with behavior specific to the class itself, that is, that most of the methods used by this class were defined elsewhere and inherited as-is. Conversely, a high value indicates that most of the methods in the class were overridden with specific behavior. The second term weights the index by increasing the value the deeper the class is in the inheritance tree, i.e. classes with more superclasses are assumed to be more specialized. This makes sense both semantically (each new level of an inheritance tree should represent a further refinement of some organizing concept) and quantitatively (the value of the index should be increased to allow for the fact that some methods may have been overridden in the parent classes). A major change in the index should indicate that the level of inheritance should be questioned. If the index becomes lower and lower, there is less and less specific functionality in this subclass (and perhaps less and less reason to keep the extra layer of the inheritance hierarchy). If the index becomes higher and higher, there is less and less functionality from the superclasses still present in the class (and perhaps less and less reason to keep this class as part of the hierarchy rather than an independent class).

3. Any system can be expressed using an object-oriented approach, but that does not mean this approach is always the right one. OO has many strengths: It allows a consistent vocabulary to be used across different phases of the development process, facilitating traceability. Many claim that it helps designers understand the problem better by allowing the design of classes that mirror real problem components. By providing a number of different diagrams, representing both dynamic and static aspects of a system, OO facilitates the analysis of various aspects of the system being designed. And, OO facilitates information hiding and encapsulation, which are widely understood to be beneficial design strategies. However, OO does have its weaknesses; maintenance of an OO system can be a difficult task because information about a particular aspect of the system can be spread across several different types of diagrams. Information hiding can help developers gain a broad understanding of the design but can make other tasks harder, such as impact analysis. Features such as inheritance and polymorphism can be helpful in the hands of an experienced designer but can create hard-to-understand systems if used poorly. One type of system for

which OO might not be the best choice is a rule-based system. When a system is mostly concerned with algorithmic processing or rule look-up (i.e. there are few discernable entities and the real complexity comes from computation), OO does not provide many applicable benefits.

4. One way to refine the design to avoid stamp coupling is to use interfaces. By designing interfaces to access the necessary data contained in the complex data structure, it is possible to reduce the stamp coupling to a data coupling since now the component depends on atomic data instead of complex data types.

5. Clearly the designers of the Ariane-5 system made a serious error when they made this decision, and making another decision would have prevented the disaster. However, equally clearly, the designers did not intend to cause such a disaster. The error was due to incompetence, not malice. The designers share responsibility for the disaster with everyone involved in the development of the software, although the decision they made seems to be at the root of the chain of events that led to the disaster.

6.
   Coincidental: a component that prints the current time or gives a directory listing, depending on the user input.
   Logical: a component that prints a document or writes it to a file.
   Temporal: a component that logs users into the system, check their mail and shows them the calendar for the day.
   Procedural: a component that reads a user database query, checks the availability of the database, then searches for the requested information.
   Communicational: a component that collects disk access housekeeping data while accessing user-requested data.
   Sequential: a component that requests a user password, reads the password, checks the password and initiates the user's session.
   Functional: a component that validates a user password and does nothing else

7.
   Content: component 1 reads in a stream of characters and updates a variable, which is internal to component 2, that keeps track of the number of lines that have been read. Component 2 uses this variable to calculate when a new page needs to be started.
   Common: components 1 and 2 are as above, but the line counter variable is in a shared data space.
   Control: component 1 reads in a stream of characters, keeping track of the number of lines read, and invokes component 2 which performs a calculation based on the data in that structure.
   Stamp: component 1 performs a validity check on a complicated data structure, then passes that structure to component 2 which performs a calculation based on the data in that structure.

Data: component 1 performs a validity check on a complicated data structure, then passes individual elements of the data structure that are needed by component 2 to complete its calculation.

8. Answers to this question may vary.

9. Probably not completely. For example, consider a system that provides a number of independent services in response to a user request. The components providing the services could be decoupled, but they would each be connected to the component which reads the user request and then calls the appropriate service.

10. Any system design could be made cohesive by making it monolithic. That is, one way to ensure that each component is self-contained is to have only on component that does everything in the system. This would be a functionally cohesive design. The trade-off would be that such monolithic designs are generally much harder to maintain and test.

11. Some examples of quality attributes and the effect of good design:
reliability: system designs that exhibit high modularity and low coupling will generally have higher reliability because they will be easier to implement and test and because the use of modularity and low coupling avoids many types of errors that are difficult to find and fix.
traceability: system designs that exhibit high cohesion enhance traceability because the relationships between functions in the requirements, design, and code are clearer when each component corresponds to just one functions.
maintainability: modularity, low coupling, and high cohesion all contribute to maintainability by facilitating understanding of the code and ease of modification.

12. In many situations, a recursive component is a very good idea because it is the most straightforward way to implement some types of algorithms. It preserves good design principles if the recursion is straightforward and direct. That it, the component calls only itself directly and does not involve other components indirectly in the recursion. Otherwise, it greatly increases and complicates the coupling between components.

13. Answers to this question may vary.

14.

```
addword(french: WORD, english: WORD)
//adds English and French words and their associations to the
//dictionary.
requires: 'french' is a French word and 'english' is an English word
          not associated in the dictionary.
ensures: adds 'french' and 'english' if not in dictionary,
          associates 'french' and 'english'.

getPronunciation(french: WORD)
//retrieves the pronunciation of a French word from the dictionary.
requires: 'french' is a French word in the dictionary.
returns: the pronunciation of the word specified in 'french'.
```

```
translate(english: WORD)
//retrieves the French translation of an English word form the
dictionary.
requires: 'english' is an English word associated with a French word
         in the dictionary.
returns: a French word that is associated with the word specified in
         'english'.
```

15. This module could be redesign in different ways to improve its generality. One way would be to allow input of types other than INTEGER, such as DOUBLE, or FLOAT. This would allow the module to be used in sorting different types of number representations. A second way would be to include a second parameter to indicate if the sorting should be done in decreasing or non-decreasing order.

16. This module might fail if one of the arrays is null, or if at least one of the arrays does not contain integers, or if the arrays are of different sizes. This module could recover from these failures by raising one or more exceptions dealing with the failure. The responsibility of passing correct values, as well as how to deal with improper inputs, would rest with the calling module.

17. This is a bad use of inheritance because Stack is not a specialization of a List. There is no overlapping functionality between the two data structures and the methods provided by the List do not apply to the way a Stack works.

18. Specification (a) is not substituable by any other specification. Having val not in the list is not required in (b) nor (d) as it is in (a), (c) does not require anything and is therefore looser than (a). Specification (b) is substituable by (a) and (d) since both specify the same behavior without requiring that val is not in the list, even though specification (d) is more strict than (b). It is not substituable by (a) nor (d)

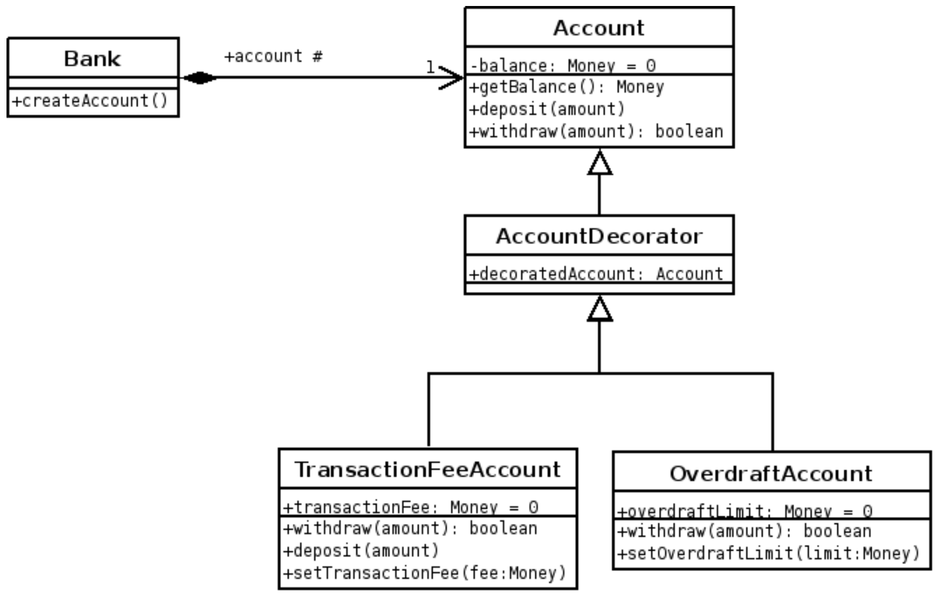
19. The specification a' is better than a since it uses active fault detection. This means that the responsibility of ensuring passing correct input parameters, or ensuring that the system recovers from an invalid state, belongs to the calling module and not the module being called.

20.
```
local
   count: INTEGER
   capacity: INTEGER
   dictionary: function that maps Key to Element
   insert(elem: Element; key: String)
   // Insert elem into dictionary
         ensure: has(key) and retrieve(key) == elem,
         if count > capacity then raise CAPACITYEXCEEDEDEXCEPTION,
         if has(key) then raise DUPLICATEKEYEXCEPTION
         if not key.valid() then raise INVALIDKEYEXCEPTION
   retrieve(key: String): Element
   // Returns the element indexed by key
   ensure: result = dictionary(key),
         if not has(key) then raise ELEMENTNOTINDEXEDEXCEPTION
end


local
   gauge: INTEGER
   capacity: INTEGER
   fill()
      //Fill reservoir with water
      ensure: in_valve.closed and out_valve.closed and is_full,
            if in_valve.closed then raise INVALVECLOSEDEXCEPTION,
            if out_valve.open then raise OUTVALVEOPENEXCEPTION
   is_full(): BOOLEAN
      // Tests whether reservoir is full
      ensure: result == (0.95*capacity <= gauge)
end
```

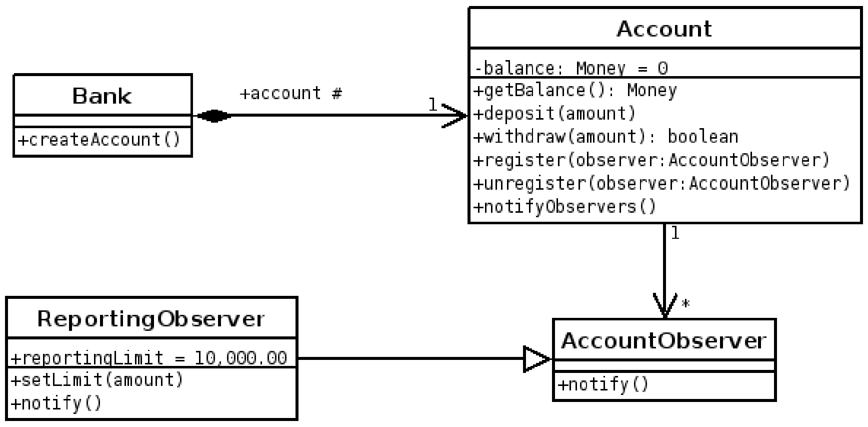
21. The second one is better. Returning the smallest representable integer does not ensure that the input parameter is valid, and therefore can result in unexpected behavior. For example, passing an array containing the smallest representable integer and passing an empty array would both result in the same output. By requiring that the input is not empty, the module ensures that the output correlates to the input.

22. The figure below shows both the overdraft protection and transaction fee modifications to the banking system.



Bank
+createAccount()
+account #
1
Account
-balance: Money = 0
+getBalance(): Money
+deposit(amount)
+withdraw(amount): boolean
AccountDecorator
+decoratedAccount: Account
TransactionFeeAccount
+transactionFee: Money = 0
+withdraw(amount): boolean
+deposit(amount)
+setTransactionFee(fee:Money)
OverdraftAccount
+overdraftLimit: Money = 0
+withdraw(amount): boolean
+setOverdraftLimit(limit:Money)

23. The figure below shows the application of the Observer pattern to the banking system.



Chapter 7: Writing the programs

1. A computed case statement interferes with the top-down flow of a program by jumping from the case statement to various other parts. It might enhance maintainability, however, by making the decision structure very explicit and clear. The result of each value of the variable will be obvious in a computed case statement. New cases are easy to add and can be assumed not to conflict with any of the other cases. Reusability may be hampered because the conditions for branching are constrained to be the value of one variable. The entire case statement would have to be rewritten to accommodate more complicated decisions.

2. All parties involved have some area of responsibility in ensuring that the component does not fail. The original developer as well as those who modified the component are all responsible for ensuring its reliability. All developers are also responsible for documenting their work clearly enough to avoid misunderstandings on the part of those who might modify it later. In theory, the developer who made the original error which resulted in a fault in the code that then caused the software to fail is responsible, but this is usually not possible to ascertain. Anyone evaluating a component for possible reuse should be sufficiently convinced of its reliability to be willing to take responsibility for any failures.

3. Recursive definition: A list is null, a single element, or an element followed by a list.

   Items are added to a list by using the third part of the recursive definition. The new element followed by the original list is, by definition, a list.

   Items are also deleted using the third part of the definition. If the first element of a list is deleted, the result, by definition, is still a list. If the list is null, it remains unchanged (or deleting causes an error).

4. Following are two Pascal code fragments, both of which add an item to a list. The first is recursive (using links), the second is not. The first should be easier to understand.

```
/***************/
type
       list = ^node
       node = record
              data: integer;
              next: list;
       end;

var    L : list;

procedure add_data (var L: list; datum: integer)
begin
       new_node := new (node);
       new_node^.data := datum;
       new_node^.next := L;
       L:= new_node
end;

/***************/
type
       list = record
              data: array[1..MAX] of integer;
              first: 1..MAX;
              last:  1..MAX;
       end;

procedure add_data (var L: list; datum: integer);
var i: integer;
begin
       if L.first = 1 then begin
       /* move all elements down 1 spot */
              if L.last=MAX then
              /* so we don't run over the end of the array */
                     L.last := L.last-1;
              for i := last downto 1 do
                     L.data[i+1] := L.data[i];
              L.data[1] := datum;
              L.last := L.last + 1
```

```
                end
                else begin
                        L.first := L.first - 1;
                        L.data[L.first] := datum
                end
        end;
```

5. One possibility is to store, for each possible year that the user could specify, an array element containing the day of the week on which the year starts, and whether or not it is a leap year. The program to print the calendar would index into the array and use the two pieces of information to print out the entire year. This would require that the rules about the numbers of days in each month would have to be coded into the computational part of the program.

   Another strategy is to store the rules about leap years and the number of days in each month in a single data structure. This structure could have an array of 12 numbers, each corresponding to the number of days in a month. It could also contain the constants needed in the algorithm to determine whether or not the year entered is a leap year (e.g. 4, 100, etc.). In this case, the computational part of the program would be much more complex, but the data structure much more compact.

6. Comments:

```
/* First, calculate the discrimant, b² - 4ac */
:
/* Case 1: The discriminant is less than 0, so there are no real roots - output
a message */
:
/* Case 2: The discriminant is equal to 0, so there is just one real root -
calculate and output the root */
:
/* Case 3: The discriminant is greater than 0, so there are two real roots -
calculate and output the roots */
:
```

   External documentation:

   The algorithm for calculating the roots of a quadratic equation has two steps. The first step is to calculate the discriminant, which is defined by a formula involving the coefficients of the equation. The second step depends on the value of the discriminant calculated in the first step. If the discriminant is less than zero, this indicates that the equation has no real roots, and a message to that effect is output. If the discriminant is equal to zero, then there is one real root. This root is calculated using the quadratic formula and output. If the discriminant is greater than zero, then there are two real roots. In this case, both roots are calculated using the quadratic formula and output.

7. This OS implements the LRU (least recently used) page replacement policy. Each process running is allocated a particular number of page frames, depending on the current load on the system. As a process requires different pages from secondary memory, they are loaded into the frames allocated to that process. Each time a page in memory is read or written, it is timestamped (i.e. it is marked with the time that it is accessed). Each time a page in memory is written to, that page is also marked "dirty," signifying that it needs to be copied back to secondary memory at some point. If a process requires a page that is not in main memory, and all the process's frames are allocated, then a page must be replaced. The page that has the oldest timestamp is replaced. If it is dirty, then it is written to secondary memory before being replaced.

8. Some improvements to consider:

   - standard header comments for each component
   - comments containing pseudocode
   - meaningful variable names
   - top-down control structure
   - greater modularity
   - greater generality
   - localizing input and output procedures
   - effective use of formatting and white space

9. The advantages are the economies of scale in terms of training, documentation, and tool licenses. As well, improvement efforts will be more efficient as improvements that are discovered in one application

area would more likely be applicable to other areas. The disadvantage is that it might constrain some projects to use a particular language or tool when it is not the best choice for that application, thus resulting in inefficiencies for that project.

10. Many standards can be coded into the tool itself, such as standards for header comments and variable names. Design and documentation standards might be facilitated by modifying them so that they are at least partially satisfied by the specifications written for the code generator. In the case of reuse from a repository, the standards must be enforced for the higher-level components that call or include the reused components. Enforcing them in the reused components may negate any cost savings from reuse.

11. In fact, it can't easily, which is a major problem for testing and documentation, especially when OO is used for real-time systems. It is sometimes possible for the software designer to map out the **intended** control flow, but in an OO program, one can never tell in advance the order in which objects will be invoked. So there is more uncertainty, and in a sense less control, with an OO program or design than there is with a procedural (hence the name) design. But you can capture most of the relevant information by using a state transition diagram/chart or petri net to describe each object's states and possible actions. Then the charts or nets can be used to help trace the likely control flow.

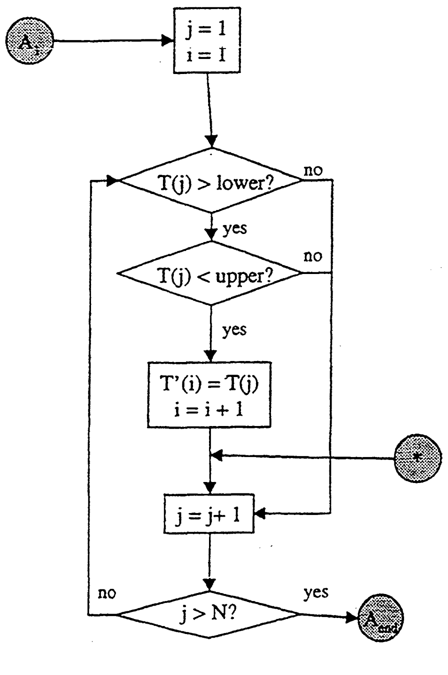
Chapter 8: Testing the programs

1. In the HP scheme, each fault has just one origin, one mode, and one or two types. For example, code errors have just one type, while design errors have two. The origins and modes are all orthogonal, and the type categories in each box in the diagram are orthogonal to the other categories in the same box. So a fault could have two types, from two different boxes, but would not fit into two type categories in the same box.

2. Assertions:

   A_1: (T is an array) & (size(T) = N) & (lower < upper)

   A_{end}: (T' is an array) & (size(T') $\leq$ N) & ($\forall$ i in 1..size(T'), T'(i) $\geq$ lower & T'(i) $\leq$ upper & $\exists$ j in 1..N, T(j) = T'(i))



3. First, the input assertion is:

   A_1: (T is an array) & (T is of size N)

   The assertion that is invariant for the inner loop (the one that goes through the array one element at a time) is the following. It is true at the beginning of every iteration of the loop (i.e. just before i is incremented):

   A_2: $i > 0$ & $i < N \rightarrow$ [(T'(i) $\leq$ T'(i+1)]

   The assertion that is invariant for the outer loop (the one that keeps starting a new pass through the array) is the following. It is true at the beginning of every iteration of the loop (i.e. just before i is set to 0):

   A_3: [*not*(*more*) = *true*)] $\rightarrow$ [(T' sorted]

   Finally, the assertion that is true at the end of the program is the following:

   A_{end}: (T' is an array) & ($\forall$ i if $i < N$ then (T'(i) $\leq$ T(i+1))
   & ($\forall$ i if $i \leq N$ then $\exists$ j (T'(i) = T(j)) & (T' is of size N)

One possible path from the input condition to the output condition is:

$A_1 \rightarrow A_3 \rightarrow A_2 \rightarrow A_3 \rightarrow A_{end}$

The proof that these assertions hold on this path is outlined below:

- At the beginning of the first time through the outer loop, *more* has just been sent to *true*, so the antecedent of A_3 is false, thus A_3 holds.
- At the beginning of the first time through the inner loop, *i* has just been set to 0, so the antecedent of A_2 is false, thus A_2 holds.
- At the beginning of the second time through the outer loop, if the inner loop has only executed once, then $N = 1$ and so by definition *T* must be sorted, thus A_3 holds.
- At the end of the program, since $N = 1$, A_{end} must hold.

4. In general, a program with *N* 2-branch decisions would require 2^N test cases. A program with *N M*-branch decisions would require M^N test cases. These are in the worst case, when the decisions are sequential. The number of paths can be reduced by nesting the decisions. For example, the following program with 4 decisions has $2^4 = 16$ paths:

```
if (a < 0) and (b < 0) then ...
if (a < 0) and (b ≥ 0) then ...
if (a ≥ 0) and (b < 0) then ...
if (a ≥ 0) and (b ≥ 0) then ...
```

The following equivalent program has 3 decisions, but they are nested so that there are only 4 possible paths:

```
if (a < 0) then
        if (b < 0) then ...
        else ...
else
        if (b < 0) then ...
        else ...
```

5. Statement testing requires that the set of test cases exercises all the statements in the program. If each statement is a node in the graph, and each test case can be represented as a path through the graph, then statement testing would be equivalent to finding a set of paths that, together, cover all nodes. Branch testing must cover all branches of every decision, so in terms of the graph we must be concerned with edges not nodes. That is, in order to cover all branches of a particular decision, not only the node corresponding to that decision, but all the edges leading out of it must be covered. So branch testing is equivalent to finding a set of paths that cover all edges. Path testing must cover all branches and all statements and all possible combinations of branches and statements, so all paths in the graph must be covered.

6. One approach is to use a depth-first search from each node in the graph. The general algorithm would be:

```
for each node n in the graph do
  find_paths({n}, n)
```

where:

```
procedure find_paths (path, n)
begin
  for each node n2 such that (n, n2) is an edge in the graph do
        path = path + n2
        find_paths (path, n2)
  end
  if no such n2 exists then
        output path
```

This algorithm does not terminate if there are cycles in the graph, so a more complex algorithm would have to be used if that is a possibility. Otherwise, this algorithm works well even if there are many paths to be calculated because only one path is calculated at a time. However, it would become less efficient with graphs with very long paths because the depth of recursion would equal the length of the path being calculated. Thus, for graphs that tend to have fewer but longer paths, a breadth-first search design might be preferred.

7. Bottom-up:	1) Test E, G, H, J, K, L, M, and N 2) Test (F, L) and (I, M, N) 3) Test (B, F, L, G), (C, H), and (D, I, J, K, M, N) 4) Test (A..N)
Top-down:	1) Test A 2) Test (A, B, C, D, E) 3) Test (A, B, C, D, E, F, G, H, I, J, K) 4) Test (A..N)
Modified top-down:	1) Test A 2) Test B, C, D, and E 3) Test (A, B, C, D, E) 4) Test F, G, H, I, J, and K 5) Test (A, B, C, D, E, F, G, H, I, J, K) 6) Test L, M, and N 7) `Test (A..N)
Big-bang:	1) Test A, B, C, D, E, F, G, H, I, J, K, L, M, and N 2) Test (A..N)
Sandwich:	1) Test A, L, M, and N 2) Test (A, B), (A, C), (A, D), (A, E), (F, L), and (I, M, N) 3) Test (B, F, G), (C, H), (D, I, J, K) 4) Test (A..N)
Modified Sandwich:	1) Test A, B, C, D, E, F, I, L, M, and N 2) Test G, H, J, K, (A, B), (A, C), (A, D), (A, E), (F, L), and (I, M, N) 3) Test (B, F, G), (C, H), (D, I, J, K) 4) Test (A..N)

8. The graph, and the research it is based on, indicate that, if many faults are found early (e.g. at compile time), there are likely to be many more faults still undetected. When that happens, one can assume that a lot of effort will be expended later in the lifecycle on finding and fixing those faults, and so it would be better to start the effort over again and write new code with fewer faults from the beginning.

9. It may be that large numbers of faults found early on indicate sloppy programming practices, which would result in even more faults found later. Or, maybe if there are many faults found early on, then a lot of changes will be made to the code to fix those faults, thus resulting in poorly structured code with lots more faults.

10. If N is the total number of indigenous faults in the program, then

$$N = \frac{\textit{(total number of seeded faults) * (number of indigenous faults found)}}{\textit{number of seeded faults found}} = \frac{25 * 5}{13} = \textit{about } 10$$

So the number of indigenous faults remaining is $10 - 5 = 5$.

11. We will need to seed S faults, where:

$$\frac{S}{(S - 0 + 1)} = \frac{95}{100}$$

$$S = 19$$

The Richards formula requires that:

$$\frac{\binom{S}{s-1}}{\binom{S+1}{s}} = 0.95$$

Simplifying:

$$\frac{\binom{S}{s-1}}{\binom{S+1}{s}} = 0.95$$

$$\frac{\frac{S!}{(S-s+1)!\,(s-1)!}}{\frac{(S+1)!}{(S+1-s)!\,s!}} = 0.95$$

$$\frac{S!\,s!}{(s-1)!\,(S+1)!} = 0.95$$

$$\frac{s}{S+1} = 0.95$$

So the Richards formula gives the required ratio between total seeded and found seeded faults, not an absolute number for the number of faults which should be seeded.

12. In the testing of any system, there is a tradeoff between the thoroughness of the test and the resources available for testing. A more thorough test requires more resources. The differences in testing these three types of systems lie in how the tradeoff is resolved. In general, large amounts of resources are committed to testing a safety-critical system in order to ensure a very thorough test. The testing would also include checking for many types of error conditions, as well as many different combinations of input data, many of them unlikely to occur in practice. Fewer resources would be expended on the testing of a business-critical system, in order to apply those resources to other parts of the development process or to profit. A system not likely to affect lives, health or business would be tested with yet fewer resources. In addition, testing of these systems might very well concentrate more on user interface issues and other aspects that enhance marketability.

13. Consider a system that takes in sensor data from a large number of environmental sensors and performs calculations that combine data from different sensors. Different sensors have different patterns of data transmittal, and there are often unpredictable delays in receiving data. However, the calculations require that the data used were all recorded at the same moment, regardless of when they were received by the central system. Furthermore, certain calculations "go bad" in the sense that, if they cannot be calculated correctly within a certain amount of time, they are no longer useful and processing must proceed to the next calculation. In the design of the system, each sensor is treated as an object, as well as each module that performs a set of related computations. Some computational objects also pass the results of their computations to other modules as input to other time-sensitive calculations, further complicating the timing complexity. Testing such a system would require testing a large variety of different timing sequences, including data which arrive too late to be useful, data which arrive in an unpredictable order, and propagation of calculated data.

14. Some issues to consider:

- How was responsibility for failures handled in the contract between the developing organization and the independent test team, if there was a contract?

- What measures did the development organization take to ensure that testing was carried out properly?
- Who was responsible for the test plan?

15. The strategy and test plan could include:
    - For each of the three components, a unit test plan which outlines test cases for:
        - each function implemented by the component
        - valid and invalid input data or user commands
        - various user scenarios
    - Plans for any inspections or reviews that will take place
    - An integration plan which outlines:
        - the order in which components will be integrated
        - how the interfaces between components will be tested
        - the role, if any, of an independent testing team
    - A system test plan which includes
        - usability testing
        - the customer's role in this phase of testing
    - Deadlines or targets for all testing activities
    - Criteria for determining when testing is complete

Chapter 9: Testing the system

1. The major function of the first pass is building a symbol table, while the object of the second pass is to produce machine code. These are the two major functions that should be tested separately (although these functions could be broken down further into subfunctions, which could in turn be tested separately). The assembler should be clearly divided into components that implement these two functions. The part of the assembler that builds the symbol table could be tested by running it on an assembly language and capturing the output, which is a symbol table. The symbol table could then be checked for correctness. Once the first part of the assembler is well tested (on several different assembly programs), the second part could be tested by running it on symbol tables which are known to be correct, and then checking the resulting machine code for correctness. Finally, the entire system would be tested by running it on an assembly language program and then checking the machine code that results. The build plan could be designed similarly, with the symbol table building functions coded, integrated, and tested first, then the code generation subsystem, then finally the entire system.
2. This type of test would be considered a function test because it is performed in order to ensure that all the functions of the compiler work properly. The requirements being tested are functional requirements, not non-functional requirements, so it is not a performance test. It is not an acceptance test because it is not performed by the customer against customer expectations (other than in the sense that the customer probably expects it to be certified). It is not an installation test because it is not concerned with differences between the development and use environments.
3. One example is the availability of developers. If many of the developers assigned to implement the system are only available for a restricted calendar period, then the number of builds may be restricted so that the developers can code intensively, uninterrupted, for the time that they are available.
4. Causes:
    1. The first four characters of the command are "LINE."
    2. The command contains exactly two parameters separated by a comma and enclosed in parentheses.
    3. B = 0
    4. B <> 0

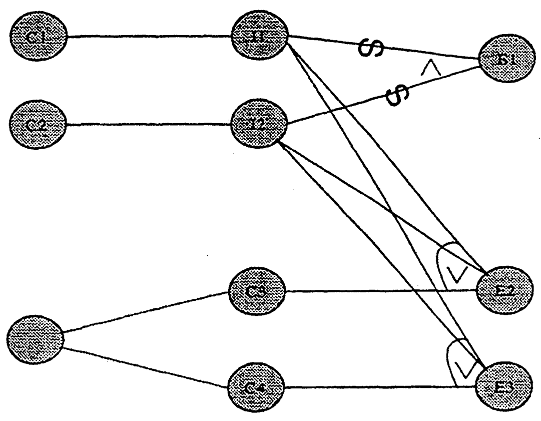
    Effects:

    1. System outputs "Improperly formed command."
    2. System outputs "D and E undefined."
    3. System outputs D and E, which are real numbers

    Intermediate effects:

    1. The command is syntactically valid.
    2. The operands are syntactically valid.

    Cause-and-effect graph:



5. Performance testing is concerned with testing non-functional requirements that specify things like speed, accuracy, and security. It is easy to specify these types of attributes ambiguously because the desire is always for the highest speed, the most accuracy, and the highest security possible. However, in order to be testable, very specific acceptance levels must be specified. For example, the requirement that the system must have "an acceptable turnaround time" is not testable because it does not give precise limits for the turnaround time. A testable requirement would be that the system "performs function X on up to 500K bytes of data in no more than 5 seconds.

6. Performance tests for a word processing system might include usability tests, i.e. tests that ensure compliance with non-functional requirements concerned with user-friendliness. Also, there might be requirements concerning the acceptable turnaround time for users, which would be tested using timing tests. Timing might also be a subject of performance tests for a payroll system, as well as security, to ensure that no one without the proper authorization can access payroll records. Security also would be a concern in performance testing an automated bank teller system, as well as recovery, timing and stress tests to ensure that the system will not get bogged down during peak times. A water quality monitoring system might be tested for the accuracy of its calculations, as well as environmental tests for speed and reliability under adverse conditions, such as natural disasters. It might also be tested for compatibility with the hardware devices (e.g. sensors) that it interacts with. Environmental tests would also be applied to a power plant control system, as well as timing and accuracy tests.

7. In the simplest case, such a system could have two configurations, one for the single-user version and one version for multiple users. There could be more than two configurations if there are multiple platforms or hardware that the software must operate with. In the simple case, the set of tests could be organized into two groups: one set that tests core functionality that is applicable to both versions, and one set that tests only the functions having to do with multiple users, e.g. timing, etc.

8. Some issues to consider:

   - what hardware devices are present on the airplane that were not available or were simulated in the development environment?
   - what regression tests should be performed?
   - what does the user consider to be the most important aspects to test?
   - what extreme conditions (e.g. weather) could be present at the installation site that were not duplicated in the development site?

9. Regression testing after the software was modified would have been the best way to detect this fault. An installation test at the Guam airport would probably also have uncovered the error.

10. If the "device" in question is a human being (e.g. for medical equipment), then it is not feasible to use an actual human for testing the software and some sort of simulator must be used. Also, if the device is inaccessible to the development environment (e.g. aboard a spacecraft in orbit) and is too large and expensive to duplicate for software testing, then a simulator must be used. Another example is when the software needs to be tested when the device is responding to some catastrophic situation with severe consequences (e.g. a device detecting a nuclear power plant meltdown). A system simulator might also be needed in this last situation so that the meltdown could be simulated (and not actually happen), as well as the system's response (shutting down the plant).

11. The form is very complete. It includes when and how the problem manifested itself, all of the events surrounding the discrepancy that might have an effect on why it occurred and how it might be fixed, as well as administrative information that will help in keeping the discrepancy process organized. Other administrative information that might be helpful would be the name of the system or subsystem being tested, the version and the release. Information about how the fault was found and fixed (e.g. components that needed to be modified, effort to isolate and fix the fault, etc.) would be included in a fault report form.

12. **Stress tests** would not be particularly useful, as this type of system would not be subject to severe unpredictable variations in activity. **Volume tests** might be applicable if there is likely to be very large amounts of payroll data. **Configuration tests** would not be applicable unless the system has several configurations. **Compatibility tests** would be required with any other systems, e.g. for inputting the payroll data and depositing the pay in employees' bank accounts. **Regression tests** are required if the system is replacing an existing system. **Security tests** are definitely needed to ensure the availability, integrity and confidentiality of payroll data. **Timing tests** need to be performed if there is a concern about acceptable response time for user requests. **Environmental tests** are probably not necessary for

this type of system. **Quality tests** might be appropriate to evaluate the system's reliability, maintainability and availability. **Recovery tests** would be appropriate to see if the system recovers properly from failures. **Maintenance tests** might be required if diagnostic tools and procedures are provided along with the software. **Documentation tests** would be needed to ensure that we have written the required documents. **Human factors tests** would also be important to evaluate display screens, messages, report formats, etc.

13. Installation test would be important in this situation because the installation sites may be different in significant ways from the development site. It would be important to check that each configuration has been calibrated correctly for the size of each factory, as well as to evaluate any other environmental differences.

14.
```
Step 1:   Determine values of A and B which indicate a low water
          level.
Step 2:   Type "LEVEL(A,B)" where A and B are the values
          calculated in Step 1.
          The system displays "LEVEL = SAFE."
Step 3:   Determine values of A and B which indicate a safe water
          level.
Step 4:   Type "LEVEL(A,B)" where A and B are the values
          calculated in Step 3.
          The system displays "LEVEL = SAFE."
Step 5:   Determine values of A and B which indicate a high water
          level.
Step 6:   Type "LEVEL(A,B)" where A and B are the values
          calculated in Step 5.
          The system displays "LEVEL = HIGH."
Step 7:   Type "LEVL(A,B)" where A and B are any values.
          The system displays "INVALID SYNTAX."
Step 8:   Type "LEVEL(A,$)" where A is any value.
          The system displays "INVALID SYNTAX."
```

15. The two sets of definitions are fundamentally different. Shooman's measures give a sense of the average behavior of a system, while the probability measures more completely describe system behavior. The probabilities can be converted into "mean time" measures by choosing the time at which the probability is 0.5, but the "mean time" measures cannot be converted into probabilities in general.

16. All of these people share some responsibility for the failure. In addition, those involved in writing the requirements (on which the test plan should have been based) share some responsibility, as well as the designers and coders who all could have considered that case.

17. Several factors must be considered in making this decision:

- what would be the consequences of a failure in the system?
- what would be the consequences of not deploying the system?
- what is the cost of building the system in the first place?
- are there other ways to address the problem, or parts of the problem?

18. The managers are, in part, responsible because they needed to review the procedures used by the V&V team to ensure that they were thorough. The V&V team itself is largely responsible because they missed something important that later caused a failure. The designers, coders, and testers might also share the responsibility depending on the ultimate source of the error.

19. The sum of $F(t)$ and $R(t)$ is always 1, so if reliability is increasing, the values of $R(t)$ will increase and the values of $F(t)$ will decrease for all values of t. In other words, the graphs of the two functions will become farther apart.

20. Some CM issues:

- What parts of the system are common to the two different versions?
- Which tests are equally applicable to both versions?
- Of the tests that are applicable to both, which are the most important ones to be used for regression testing?

- What dependencies exist between common components and components that are specific to each of the two versions?

A good configuration management strategy might have helped Wind River to more easily port their operating system because it would have made clear which parts of the system would need to be modified and how the new modified system should be tested.

21. In analyzing testing principles and methods, it is important to be able to tell when a test case or testing activity reveals a fault in the software. In order to do this, it is necessary to distinguish the behavior exhibited in the test environment from the desired behavior. In complex systems, erroneous behavior may appear similar to the desired behavior, so an oracle is required to distinguish the two.

22. A possible plan is outlined below. It starts with the basic pricing functions based on the time and day of the advertisement. It adds functionality by considering, in turn, other factors that affect price and strategy:

Spin	*Functions*	*Test start*	*Test end*
0	Basic Pricing (based on day and time)	1 January	31 January
1	Ratings	28 February	15 March
2	Restrictions	25 April	5 June
3	Opposition schedules	10 June	20 June

Chapter 10: Delivering the system

1. If the prototype and the final working system differ in ways that affect how the user performs tasks, then the user may have trouble blocking out the features of the prototype, which they might be used to or they might prefer to the features in the final system. This type of task interference would apply in addition to the task interference from the user's previous procedures for performing tasks.

2. Many applications that are designed for home PC use are not meant to have separate users and operators, as the user will also be responsible for installing the software, configuring its settings and parameters, and backing up and restoring files. Therefore both user and operator functions must be included in the training.

3. In general, in any software designed for the general public (e.g. word processors, spreadsheet packages, home financial applications), the underlying system should be transparent to the user. This is generally considered good design because the visibility of system details, which are not directly related to user functions, makes the interface less user-friendly. For example, a textual document might be stored on a computer in pages where the page size is determined by the operating system, not by any attributes of the document itself. But when the user processes that document using a word processor, it should be broken up into pages that correspond to the printed appearance of the document, and the user should have no sense of where the boundaries are between memory pages.

5. Some questions to consider:
   - What is the background of the system's users? Are they likely to understand failure reports?
   - Does not reporting failures to the user make the system appear more reliable than it really is?
   - Are there cases in which the user or operator might want to take additional action in response to a failure, besides what the system automatically does?
   - Could there be patterns of failures over time in which each individual failure is not particularly alarming, but the trend may indicate a more serious problem?
   - Would it be appropriate to record failures in a location that a user or operator could check periodically, rather than intrusively notify the user each time a failure occurs?

6. Most of the failure messages are clear and straightforward and would be understandable to most users. In some cases, friendlier language could be used (e.g. "printer" or "disk," etc., instead of "device"). Also, additional information would be helpful in several cases (e.g. line numbers for program errors).

Chapter 11 : Maintaining the system

1.

   a. An air traffic control system is an E-system because it is subject to numerous environmental changes, e.g. changes in air traffic patterns and routes, changing understanding of weather patterns, changes in aviation law, etc. Any aspect of such a system could change.

   b. An operating system for a microcomputer is also an E-system because it could change based on changes to the hardware it was written to support, or to changing user desires. Again, any aspect of the system could change.

   c. A floating-point acceleration system is most likely a P-system because the implementation is a less-than-perfect solution to the problem, although the problem is well defined. In this case, the specification, the requirements, and the system itself could all change, but not the problem or the world in which the system operates.

   d. A database management system is an E-system because there are numerous aspects of the real world that could motivate changes in the system. The underlying hardware, other software, customer needs and wants, etc., could all change in ways that necessitate changing some aspect of the database management system.

   e. A system to find the prime factors of a number is an S-system because the problem and solution can be precisely stated, the specification can precisely and completely describe the solution, and the implementation can completely fulfill the specification. No aspect of this system is likely to change.

   f. A system to find the first prime number larger than a given number is likely to be a P-system because, although the solution to the problem can be completely specified, it cannot be completely implemented, practically speaking, for arbitrarily large input numbers. Thus, the system would implement a practical abstraction, or approximation, to the solution. The specification, requirements, and system implementation could all change in order to implement the actual solution better.

2. High coupling among components usually increases the number of components that must be modified to implement a particular change, and also makes it difficult to determine which components must be changed. As a result, there are more opportunities for faults to be introduced into the system as the result of a change.

3. Nearly all aspects of system "success" depend on the system's documentation. In particular, the quality of the technical documentation generated during system development will affect how well it is maintained, and whether or not the reliability of the system will degrade over time as it is changed. Also related to maintenance, the documentation needs to be easy to update, or it will become useless over time even if it was of high quality to begin with.

5. Maintenance programming is challenging because the programmer must work, to some extent, within the style and paradigm used by the original programmer, even if that style seems unnatural, or even undesirable. Maintenance programmers must have exceptional "people skills," especially communication skills, in order to effectively communicate with original programmers to determine their intent and reasoning with respect to code that is difficult to understand. Maintenance programmers must also be able to do this in a non-judgmental and diplomatic way, even if they consider the original programmer's work to be poor. Other desirable characteristics of a maintenance programmer are good analytic code reading abilities.

6. The advantage of writing documentation during development is that it is more likely to be correct (because the writer is not relying on memory). The drawbacks are that it takes time away from development, and that often a developer has a better sense of the system as a whole (i.e. the "big picture") when they are not involved in the coding of individual components.

7. Some issues to consider:

   - Can you easily tell, from the documentation, what components implement each function of the system?

- Is it easy to trace the control flow of the program from the documentation?
- When changing a particular component, could you tell from the documentation which other components are likely to need changing?
- Does a large system need these types of documentation more or less than a small system? Why?

8. It's very important to retain formal test data and test scripts for maintenance in order to perform regression test after changes have been made, and also to guide the maintainers in designing new tests.

9. When a component has a single entry and a single exit, it is much easier to trace the execution steps during testing. There is only one way to get into a program, and one way to get out. If an execution trace of a program shows that a component was at least partially executed, then the tester knows that the code at the entry point was executed. And if control did not leave the component, then the tester knows that the code at the exit point did not execute. When there are multiple entrances and exits, it is extremely difficult to trace all the different possibilities.

10. Low coupling and high cohesion in a system implies a good modular structure at the component level, which would mean that restructuring would more likely happen at a higher, and easier, level, such as the subsystem level. Other types of rejuvenation would be facilitated as well because of the modular structure. However, such a structure would most likely decrease the need for rejuvenation.

    Good exception identification and handling functions might hinder redocumentation efforts because it would be difficult to understand the intent of the exception handling code if it is not already well documented. Restructuring, however, should be facilitated if the exception handling functions are easily identified and separate from the rest of the system. Reverse engineering might have some of the same problems as redocumentation, and consequently so would reengineering.

    Extensive fault handling and tolerance functionality is usually difficult to implement in separate routines, so it is often embedded in other functions of the system. This makes all types of rejuvenation more difficult because the fault handling and tolerance strategy (which might be quite sophisticated and complicated) must be understood in order to understand any of the code.

11. The Ariane-5 software is an E-system because it is subject to change in response to many types of changes in its environment. It was such a change (in hardware) that was not responded to adequately that caused the disaster.

12. The cyclomatic number allows us to order components by cyclomatic number only. That is, we can only say that one component has more independent paths than another. We cannot, with only the cyclomatic number, say that one component has higher quality or lower complexity, in general, than another. One aspect of complexity that is not captured by the cyclomatic number is interface complexity, i.e. the complexity of the data passed into and out of a component and how many other components are related to that component.

13. Yes, because a prediction model such as Porter and Selby's is not meant to be used in this way. The model only points to components that are likely to fail. It does not imply the converse, i.e. that other components will not fail.

14. Below are outlined the functional criteria and the ways in which they contribute to ease of maintenance:

    a. **Record versions or references to them** – helps the maintainer find when and where changes have been made and to track the history of a component

    b. **Retrieve any version on demand** – same as above

    c. **Record relationships** – helps the maintainer understand relationships between components that might not be obvious from the code itself

    d. **Record relationships between versions to which the tool controls access and those to which it does not** – notifies the maintainer when there are other versions that need to be taken into consideration; otherwise such versions might be overlooked

    e. **Control security and record authorizations** – prevents uncontrolled and undocumented changes to code and other artifacts

    f. **Record changes to a file** – provides documentation of all changes

    g. **Record a version's status** – allows the maintainer to keep track of maintenance progress

h. **Assist in configuring a version** – helps the maintainer to keep track of all the detailed requirements of each configuration

i. **Relate to a project control tool** – helps in managing the maintenance project

j. **Produce reports** – same as above

k. **Control releases** – helps the maintainer to document and manage all the release details

l. **Control itself** – reduces the overhead burden in using any tool

m. **Archive and retrieve infrequently-used files** – prevents the loss of information which may at some point be useful while consuming minimal resources in the maintenance of that information

Chapter 12: Evaluating products, processes and resources

1. A set of facets with which to start could be:

   - phase in the development cycle
   - programming language
   - application domain
   - intended audience
   - type of book

   It is very difficult to know when enough facets have been defined. The set of facets can be tested by use to see if they are at the right level of detail, if they cover all relevant aspects, and if there are enough of them. There are enough when each book can be accurately described and when any two books with the same values for all facets really are very similar. The degree of similarity between two such books depends on the level of detail of the classification. At the lowest level of detail (for example, if author and title are both facets), all book descriptions will be unique. An example of such a detailed cataloguing system is the Library of Congress system for books. In the LOC system, every book has a unique identifier. The LOC system is also a faceted system because every part of the identifier actually refers to some particular attribute, or facet, of the book.

2. The cost of reusing software includes the cost of developing that software to begin with (the cost of "producer reuse") and the cost of adapting that software into future systems (the cost of "consumer reuse"). Producer and consumer reuse take place in different projects, so the total cost of reuse must take into account costs in more than one project.

3. Some examples:

   - For each time the component was reused, a description of the modifications that had to be made to the component – this would help potential reusers know what types of modifications might have to be made in order to make the component work in their system.
   - For each time the component was reused, the effort required to modify it – this would help potential reusers know how much effort they might plan on expending to reuse the component.
   - Error history – this would give an idea of the quality of the component.
   - Total number of times the component was reused – this would give managers and analysts an idea of the most heavily used components, with the aim of improving those components with the largest impact.

4. Some possibilities are:

   - plan further training for that developer,
   - assign that developer to more appropriate tasks in the future
   - change hiring practices so that developers with similar deficiencies will not be hired in the future

5. Some questions you might use to answer the last part of the question:

   - Will doing these things differently lower the cost of the next project?
   - Will doing these things differently make the next project shorter?
   - Will doing these things differently improve the quality of the next product?
   - Will doing these things differently decrease the risk of unwanted "surprises" during the next project, especially late in the project?
   - Will doing these things differently decrease the amount of stress and frustration experienced by the personnel in the next project?
   - Will doing these things differently improve our relationship with the customer on the next project?
   - Will doing these things differently cause other problems on the next project that we didn't have on this project?

   It is not likely that you will be able to answer any of these questions definitively because many improvements must be tried before it is known whether they will work or not. This type of evaluation is part of the job of case studies and experiments.

6. One example from Boehm's model is accuracy, which could be measured objectively as the percentage of test cases passed. On the other hand, communicativeness might be measured subjectively by usability testers. An example from the ISO 9126 model is testability, which is sometimes measured objectively by measuring "ambiguous phrases" in the requirements document. There are specific lexical rules for ambiguous phrases, such as the use of "should" or "can." The idea is that if the requirements document is ambiguous, then testing will be harder. A subjective example from Dromey's model might be reusability, measured using developer ratings.

7. The inconsistency between the one-letter codes used on most aeronautical charts and those expected by the software system is really where the error lies. Such an error has to do with defining the system boundary and recognizing the interaction between the system within the boundary and systems that lie outside the boundary. Boehm's model addresses some attributes that, had they been explicitly tested for in this system, might have revealed the error (e.g. consistency, robustness/integrity). However, the model does not explicitly deal with the system's relationship to its environment. The same can be said for the ISO 9126 and the Dromey models, although some of the portability attributes, in particular conformance in the ISO model, might have been used to test for such errors. Another way to view this error is as a usability problem. That is, with a very narrow view of the system boundary, the pilot can be said to have caused the failure because he entered invalid input. All the quality models address usability, and two of them list an attribute communicativeness which seems to relate to this problem. Communicating to the user the effects of his/her actions could have ameliorated the effects of the error. It is unlikely that measurement could have helped to avoid this particular problem, but it can be used during usability and other types of testing to measure the number and severity of errors which occur, in order to evaluate the reliability and usability of the system and how likely it is to permit a severe error. Also, safety analysis techniques, such as FMEA and HAZOPS, applied to the design or to the system before delivery might have caught the problem before it took lives.

8. Boehm's model is different from the other two because it includes the views of different types of developers and users. That is, those who are going to port or maintain the system can be viewed as both developers and users, and their viewpoints are included in Boehm's model. In addition, Boehm's model really begins with a model of usability, so the user view takes priority. The ISO 9126 model takes a completely user view, in that both levels of characteristics are user-oriented, although the model does not provide ways to measure these characteristics, from either a developer or user point of view. Also, the ISO 9126 model is the only one that addresses security, which may be a user concern.

9. It makes sense to have a general model because it does facilitate the comparison of different systems by standardizing terminology. However, although probably any product can utilize the ISO 9126 model to some extent, many products will be better evaluated by a tailored version of the model which eliminates some factors that are irrelevant and adding others that are relevant to that particular product. More unusual products will require more tailoring of the model, and it is conceivable that there are products for which ISO 9126 could be said not to apply at all.

10. Security is a component of functionality in the ISO 9126 model, so ensuring the security of the system is considered one of the functions that the system is meant to provide. This means that the security needs of the system must be specified along with the other functional requirements, and then the resulting product must be tested against those security requirements to see if they are satisfied. This approach implies that security is something that either exists in the system or it doesn't. If it doesn't, the system does not function as specified. Another view of security is that it is a continuum, i.e. that a system can exhibit some level of security, just as it can exhibit some level of reliability. Requirements can then be specified for the level of security of the system, rather than specific security functions. With this view of security, it makes more sense to relate security to reliability in the ISO 9126 quality model.

11. The effectiveness of the review technique could be evaluated with an experiment in which two projects are compared. In one project, the technique would be used and in the other it would not. The quality of the resulting requirements documents and other downstream products could be measured to evaluate the effect of the technique. Ideally, to control all variables, all aspects of the two projects would have to be identical: same personnel, problem, customer, budget, schedule, etc. Practically speaking, this is not possible, but it is important that the two projects be as similar as possible. A good compromise would be to compare two projects with personnel of similar backgrounds, the same application domain, the same computing environment (hardware, supporting software, etc.), approximately the same size, etc. Another approach would be to use the same personnel on both projects, to make sure that differences between personnel do not confound the results. In this case, the first project would

have to be the one not using the review technique. Then, the personnel would have to be taught the technique, then given another similar (but not the same) project to work on.

12. For example, a goal that would help fulfill the "software project planning" KPA would be to "expend between 10% and 15% of each project's budget on planning activities." This goal is measurable using effort data. The growth, and then stability, of this percentage over time would reflect the growing maturity of the process. Another example would be the "organization process definition" KPA, which could have the corresponding goal of "100% of all projects use the organization's process definition." This is also measurable, and the growth in the measure towards 100 would indicate increasing process maturity in this area.

13. One way to test this hypothesis is to examine data from a large set of projects, which varied in terms of team cohesion and resulting quality, but were similar in other respects. Team cohesion could be measured subjectively by surveying project team members. The survey could include such questions as "How much (on a scale of 1 to 5) did you enjoy working with the other team members?," "How likely are you to choose to work with the same team again?," etc. Product quality could be measured objectively using defect data, or subjectively by surveying users about their satisfaction with the products.

Chapter 13: Improving predictions, products, processes and resources

1. We could investigate the role of increased understanding of the product in decreasing fault density by comparing projects that used the inspection process with those that didn't around the same time. Two such projects would most likely exhibit about the same level of understanding of the product group. Or, we could look at the level of experience of the project participants and compare projects with similar experience levels. The other factor, sloppy inspection and development activities, might result in lower fault densities if many of the faults in the product are not found. To test this, we would have to look at field error data, i.e. failures reported by customers after the products are operational.

2. The benefit of incorporating such assumptions into the model is that it results in more realistic and accurate models. However, the drawback is that it is very difficult to ensure that the assumptions are correct. To be confident in our assumptions, we have to base them on empirical evidence. That is, they must be supported by data from past projects.

3. A systems dynamics model can be built in which the two major variables are security and performance. Incorporated into the model would be a number of assumptions about how different factors affect and are affected by security and performance. For example, the model might include assumptions about how much it costs to implement added security features into the system, and how much it costs to develop the system with higher performance.

4. The danger is that the SEL experience does not apply to your organization. If your organization differs from the SEL in terms of application domain, computing environment, typical project budget and schedule, or a variety of other factors, then you may not be as successful, or you may be more successful, at implementing Cleanroom. This is why a strategy of careful study and experimentation is important in deciding if a particular technology is well suited to a particular environment. Although the specific experiences the SEL had with Cleanroom might not be universally applicable, you could use the strategy that the SEL used to investigate Cleanroom in your organization.

5. The first use of a new technology such as rapid prototyping must be done very carefully. If possible, it should have been tried previously on a small, non-critical project. Ideally, it could be tried first on a parallel project, i.e. a project that is implemented twice, once with rapid prototyping and once without. In any case, its first few uses must be carefully monitored, both to ensure that rapid prototyping is actually being carried out (we cannot say we are evaluating the rapid prototyping process if that is not really what was being followed) and to find out what effect it is having, both positive and negative. This monitoring would be done via collection and analysis of data, including effort data (to make sure the rapid prototyping process is not actually requiring more effort than expected), schedule data (to make sure that it is not causing delays), and defect data (to make sure that the quality of the product is not suffering). The *post-mortem* analysis is particularly important in a project in which a new technology is introduced. It should concentrate on how the process was affected, both positively and negatively, by the introduction of rapid prototyping, how developers liked the new technique, if the technique was implemented correctly, and, most importantly, how it could be improved when/if it is used next.

6. One strategy is to compare data from projects before and after the certification, with respect to the following measures:

   - effort, normalized by size of project
   - calendar time, normalized by size of project
   - product quality
   - accuracy of estimation
   - revenue

   Also, the cost of the certification process itself must be measured. This includes the effort to implement process improvement practices, collect and analyze data, create and modify documentation, and perform assessments.

7. Some issues to consider:

   - What development activities should a licensed software engineer be able to demonstrate proficiency in?
   - If a software component, written by a licensed software engineer, fails with catastrophic consequences, should that engineer's license be revoked?
   - What should be the minimal educational requirements be for licensing (BS? MS? in computer science? in a related field?)?
   - How can we ensure that the criteria for licensing do not become outdated as software engineering technology evolves?
   - Should a licensed software engineer be able to continually improve his/her own skills, as we expect of development organizations?

Chapter 14: The Future of Software Engineering

1. Consider the activities of a software project manager. Where are decisions made? When are they group decisions? When are they individual decisions?

Answers to this question may vary, as there are numerous possible decisions to be made by a project manager, in all phases of the lifecycle. Individual decisions are typically those which relate directly to the project manager's role or expertise in keeping the project on track, such as how much effort to allocate to different activities, or what kinds of process to use to keep the project on time and under budget (i.e. whether inspections are worth the investment in time and effort). Group decisions are typically those in which the roles and experiences of the other developers can significantly contribute, such as whether it would be worthwhile to invest in a tool that would automate some of the developers' tasks but require the developers to master a steep learning curve, or whether a particular work document is of sufficient quality to pass an inspection.

2. What software technologies have been promoted in the last ten years? Which ones have resulted in widespread adoption and which have not? Can you use the Rogers and Moore frameworks to explain why?

Answers to this question will vary, as there are numerous software technologies from which to choose. Answers should cite literature (the number of books available on a topic is a good indicator of its penetration) or argue from personal experience to establish the level of support available and the amount of evidence as to effectiveness for the technologies chosen.

3. The January/February 2000 issue of *IEEE Software* contains an editorial by McConnell about software engineering's best influences:

- reviews and inspections
- information hiding
- incremental development
- user involvement
- automated revision control
- development using the internet
- programming languages: Fortran, COBOL, Turbo Pascal, Visual Basic
- Capability Maturity Model for Software
- component-based development
- metrics and measurement

For each practice listed, analyze its likely technology adoption. Is it a best practice? What evidence supports its adoption? And what audience is addressed by the evidence?

Answers to this question will vary, but should support their conclusions with reasonable arguments. The article summarizes the level of adoption of each technology in industry, although students should look for industry experience reports that can confirm or deny that impression. (Many conferences, including ICSE, now contain tracks devoted to industry status reports, making these a good place for students to start searching.) Students should also turn to the literature to understand the amount of published evidence on a topic.