Software Engineering
7th edition

Solutions to selected exercises

Solutions to the exercises are organised by chapter and I have provided solutions for 6 or 7 exercises for each chapter in the book. In some cases, where the material is likely to be unfamiliar or where I have found students to have particular difficulties, a larger number of solutions are given. Overall, I have provided solutions for about 60% of the exercises. For exercises concerned with ethical issues, there are of course, no definitive solutions. For these exercises, I have included issues that might be addressed.

However, the solutions here are simply indications of what might be expected from students attempting the exercises. Many of the exercises have been deliberately designed so that they may be adapted to local situations; therefore they are not specified in a rigid way. Instructors, therefore, may use these solutions as a guide but many other possible, equally valid, solutions may also be generated.

There are still a small number of chapters where there are fewer than 6 solutions to exercises. These additional solutions will be available in the next release of this document in September 2004.

Chapter 1 Introduction

Solutions provided for Exercises 1.2, 1.3, 1.4, 1.6, 1.7 and 1.8.

1.2 The essential difference is that in generic software product development, the specification is owned by the product developer. For custom product development, the specification is owned by the customer. Of course, there may be differences in development processes but this is not necessarily the case.

1.3 For important attributes are maintainability, dependability, performance and usability. Other attributes that may be significant could be reusability (can it be reused in other applications), distributability (can it be distributed over a network of processors), portability (can it operate on multiple platforms) and inter-operability (can it work with a wide range of other software systems). Decompositions of the 4 key attributes e.g. dependability decomposes to security, safety, availability, etc. are also possible answers.

1.4 A software process is what actually goes on when software is developed. A software process model is an abstraction and simplification of a process. Process models can be used to help understand real processes and to identify which aspects of these processes could be supported by CASE tools.

1.6 Method support provided by CASE tools:

Editors for specific graphical notations used
Checking of the 'rules' and guidelines of the method
Advice to tool users on what to do next
Maintenance of a data dictionary - all names used in the system
Automatic generation of skeleton code from the system models
Generation of reports on the design

1.7 Problems and challenges for software engineering

Developing systems for multicultural use
Developing systems that can be adapted quickly to new business needs
Designing systems for outsourced development
Developing systems that are resistant to attack
Developing systems that can be adapted and configured by end-users
Finding ways of testing, validating and maintaining end-user developed systems

There are obviously lots of other problems that could be mentioned here.

1.9 Advantages of certification

- Certification is a signal to employers of some minimum level of competence.
- Certification improves the public image of the profession.
- Certification generally means establishing and checking educational standards and is therefore a mechanism for ensuring course quality.
- Certification implies responsibility in the event of disputes.
  Certifying body is likely to be accepted at a national and international level as ‘speaking for the profession’.
- Certification may increase the status of software engineers and attract particularly able people into the profession.

Disadvantages of certification

- Certification tends to lead to protectionism where certified members tend not to protect others from criticism.
- Certification does not guarantee competence merely that a minimum standard was reached at the time of certification.
- Certification is expensive and will increase costs to individuals and organisations.
- Certification tends to stultify change. This is a particular problem in an area where technology developments are very rapid.

These are possible discussion points - any discussion on this will tend to be wide ranging and touch on other issues such as the nature of professionalism, etc.

Chapter 2 Computer-based system engineering

Solutions provided for Exercises 2.1, 2,2, 2.3, 2.4, 2.6, 2.7, and 2.8.

2.1 Other systems in the system's environment can have unanticipated effects because they have relationships with the system over and above whatever formal relationships (e.g. data exchange) are defined in the system specification. For example, the system may share an electrical power supply and air conditioning unit, they may be located in the same room (so if there is a fire in one system then the other will be affected) etc.

2.2 This is an inherently wicked problem because of the uncertainties associated with the problem. It is impossible to anticipate exactly when and where a disaster will occur, the numbers of people involved, the effects on the environment, the technology available to the emergency services, etc. Planning can only be in very general terms and detailed software specifications to cope with specific situations are almost impossible to write.

2.3 When a car is decommissioned, not all of its parts are worn out. Software systems can be installed in the car to monitor the different parts and to compute the lifetime which they are likely to have left. When the car is to be decommissioned, the parts which can potentially be reused can then easily be discovered.

2.4 An overall architectural description should be produced to identify sub-systems making up the system. Once these have been identified, they may be specified in parallel with other systems and the interfaces between sub-systems defined.

2.6 The key features of the solution are:

- Database with different types of data
- Video control system
- Operator console system
- River data collection
- Weather system links
- Communication control system

See Figure 2.1.

2.7 Possible issues covered in the solution might be:

- Museums are conservative places and some staff may resent the introduction of new technology.
- Existing museum staff may be asked to deal with problems of the equipment not working and may not wish to appear unable to deal with this.
- Other areas of the museum may oppose the system because they see it as diverting resources from their work.
- Different museums may have different preferred suppliers for the equipment so that all equipment used is not identical thus causing support problems.
- The new displays take up a lot of space and this displaces other displays. The maintainers of these displays may oppose the introduction of the system.
- Some museums may have no mechanism for providing technical support for the system.

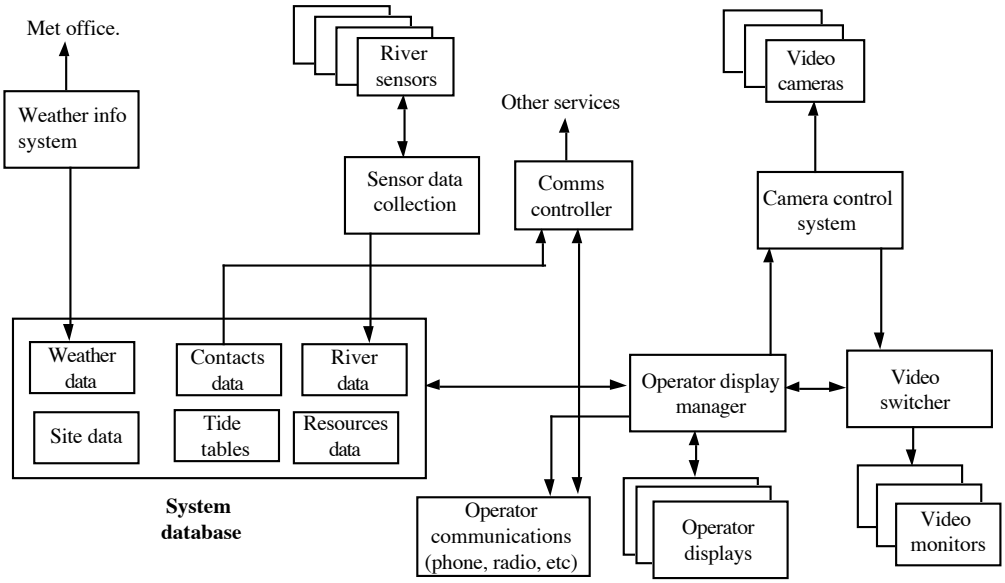

Figure 2.1 Block diagram of the flood control system

2.8 Legacy systems may be critical for the successful operation of a business for two basic reasons

- They may be an intrinsic part of one or more processes which are fundamental to the operation of a business. For example, a university has a student admissions process and systems which support this are critical. They must be maintained.
- They may incorporate organisational and business knowledge which is simply not documented elsewhere. For example, exceptions on student admissions may simply have been coded directly into the system with no paper record of these. Without this system, the organisation loses valuable knowledge.

Chapter 3 Critical systems

Solutions provided for Exercises 3.2, 3.5, 3.6, 3.7, 3.8, 3.10 and 3.11.

3.2 Six reasons why dependability is important are:

a) Users may not use the system if they don't trust it.

b) System failure may lead to a loss of business.

c) An undependable system may lose or damage valuable data.

d) An undependable system may damage its external environment.

d) The reputation of the company who produced the system may be damaged hence affecting other systems.

e) The system may be in breach of laws on consumer protection and the fitness of goods for purpose.

3.5 Internet server: **Availability** as failure of availability affects a large number of people, reputation of the supplier and hence its current and future income.

A computer-controlled scalpel: **Safety** as safety-related failures can cause harm to the patient.

A directional control system: **Reliability** as mission failure could result from failure of the system to perform to specification.

An personal finance management system: **Security** because of potential losses to users.

3.6 Possible domestic appliances that may include safety-critical software include:

Microwave oven
Power tools such as a drill or electric saw
Lawnmower
Central heating furnace
Garbage disposal unit
Food processor or blender

3.7 Ensuring system reliability does not necessarily lead to system safety as reliability is concerned with meeting the system specification (the system 'shall') whereas safety is concerned with excluding the possibility of dangerous behavior (the system 'shall not'). If the specification does not explicitly exclude dangerous behavior then a system can be reliable but unsafe.

3.8 Possible hazard is delivery of too much radiation to a patient. This can arise because of a system failure where a dose greater than the specified dose is delivered or an operator failure where the dose to be delivered is wrongly input.

Possible software features to guard against system failure are the delivery of radiation in increments with a operator display showing the dose delivered and the requirement that the operator confirm the delivery of the next increment. To reduce the probability of operator error, there could be a feature that requires confirmation of the dose to be delivered and that compares this to previous doses delivered to that patient. Alternatively, two different operators could be required to independently input the dose before the machine could operate.

3.10 An **attack** is an exploitation of a system vulnerability. A **threat** is a circumstance that has the potential to cause loss or harm. An attack can lead to a threat if the exploitation of the vulnerability leads to a threat. However, some attacks can be successful but do not lead to threats as other system features protect the system.

3.11 The ethics of delivery of a faulty system are complex. We know that this happens all the time, especially with software. Issues that might be discussed include the probability of the fault occurring and the consequences of the fault – if the fault has potentially serious consequences then the decision may be different than if it is a minor, easily recoverable fault. Other issues are the price charged for the system (if its low, then what level of quality is it reasonable for the customer to expect). The recovery mechanisms built into the system and the compensation mechanisms that are in place if consequential damage occurs. Making the customer aware of the fault is the honest decision to make but may be unwise from a business perspective.

Claims about the reliability of the software should not be made in such circumstances as the software provider does not know how the software will be used and so cannot estimate the probability of occurrence of the fault.

Chapter 4 Software Processes

Solutions provided for Exercises 4.1, 4.3, 4.7, 4.9, 4.10 and 4.12.

4.1 (a) *Anti-lock braking system* Safety-critical system so method based on formal transformations with proofs of equivalence between each stage.

(b) *Virtual reality system* System whose requirements cannot be predicted in advance so exploratory programming model is appropriate.

(c) *University accounting system* System whose requirements should be stable because of existing system therefore waterfall model is appropriate.

(d) *Interactive timetable* System with a complex user interface but which must be stable and reliable. Should be based on throw-away prototyping to find requirements then either incremental development or waterfall model.

4.3 The waterfall model is accommodated where there is a low specification risk and no need for prototyping etc. for risk resolution. The activities in the 2nd quadrant of the spiral model are skipped. The prototyping model is accommodated when the specification phase is limited and the prototyping (risk resolution) phase predominates. The activities in the 3rd quadrant of the spiral model are skipped or reduced in scope.

4.4 Solution to be added.

4.7 Components of a design method are:

A defined set of system models
Rules that apply to these models
Guidelines for design 'good practice'
A model of the design process
Formats for reports on the design

4.9 Systems must change because as they are installed in an environment the environment adapts to them and this adaptation naturally generates new/different system requirements. Furthermore, the system's environment is dynamic and constantly generates new requirements as a consequence of changes to the business, business goals and business policies. Unless the system is adapted to reflect these requirements, its facilities will become out-of-step with the facilities needed to support the business and, hence, it will become less useful.

4.10 A classification scheme can be helpful for system procurement because it helps identify gaps in the CASE tool coverage in an organisation. Procurement may be aimed at filling these gaps. Alternatively, a classification scheme may be used to find tools which support a range of activities - these may represent the most cost effective purchases if funds are limited.

4.12 There are obviously different views here and a lot depends on the development of CASE technology in the future. A major difference between the introduction of CASE technology and, for example, the introduction of CAD technology which made draftsmen redundant, is that the routine elements in the design and development of software are relatively minor parts of the whole development process. Therefore, savings are not that large. However, if AI technology develops so that truly intelligent tools can be developed than, obviously, this situation will change.

Chapter 5 Project management

Solutions provided for Exercises 5.2, 5.3, 5.6, 5.9,5.10 and 5.11.

5.2 Management activities such as proposal writing, project planning and personnel selection require a set of skills including presentation and communication skills, organisational skills and the ability to communicate with other project team members. Programming skills are distinct from these (indeed, it is a common criticism of programmers that they lack human communication skills) so it does not follow that good programmers can re-orient their abilities to be good managers.

5.3 Project planning can only be based on available information. At the beginning of a project, there are many uncertainties in the available information and some information about the project and the product may not be available. As the project develops, more and more information becomes available and uncertainties are resolved. The project plan therefore must be reviewed and updated regularly to reflect this changing information environment.

5.6 The activity chart and bar chart are shown as Figures 5.1 and 5.2.

5.9 Other possible risks are:

Technology: Communications network saturates before expected transaction limit is reached.
People: Level of skill of available people is lower than expected.
Organisational: Organisational changes mean that the project schedule is accelerated.
Tools: CASE tools cannot handle the volume of data available for large systems.
Requirements: New non-functional requirements are introduced that require changes to the system architecture.
Estimation: The difficult of the software is underestimated.

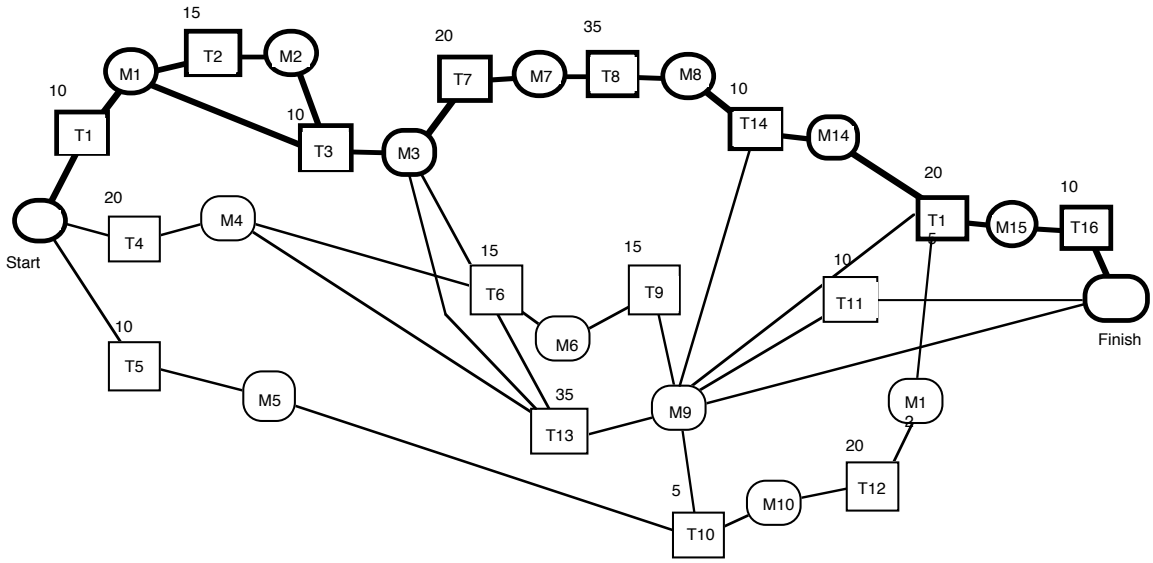

Figure 5.1 Activity chart

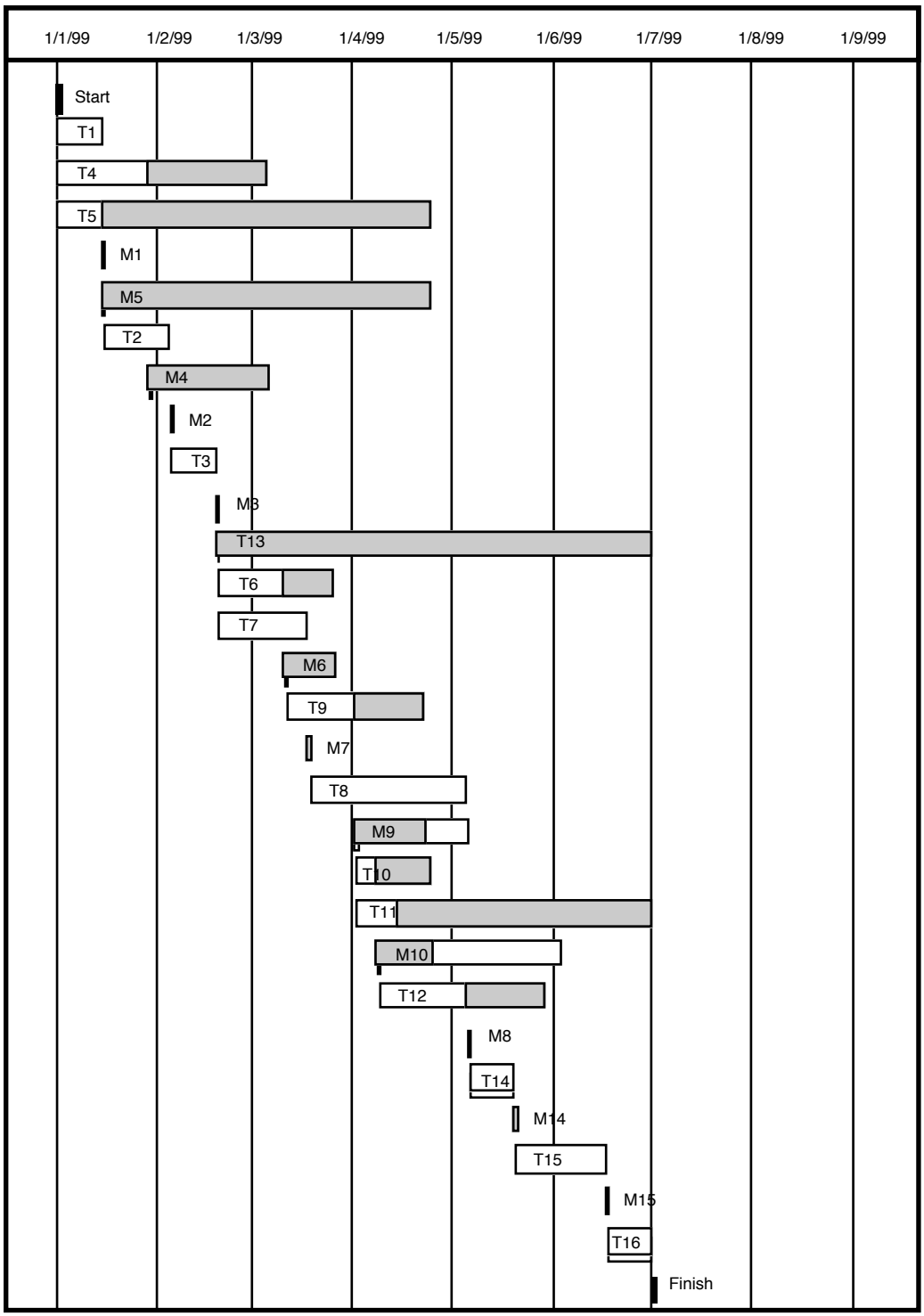

Figure 5.2 Task bar chart

5.10 Fixed price contracts increase the chances of product risks because they remove options from the development process. Because the contract is fixed-price, the contractor is naturally reluctant to increase the effort or time expended on the project as this will reduce their profits on the work. Therefore, if problems arise they will look for ways to reduce the scope of the product or to reduce the costs of product development (e.g. by reducing the effort devoted to testing). Both of these factors can lead to products that are not as expected by the customer.

5.11 Issues which might be covered include the problems of finding a balance between family life and organisational demands, whether or organisations should expect people to behave as professionals. This perhaps implies working the number of hours required to complete some job but also implies that engineers should have a degree of autonomy about how they arrange their working lives (e.g. they may *choose* to work from home or their own working hours).

Factors which affect this decision might be the financial state of the company, the general company culture and attitude, the availability of alternative local employment, particular personal circumstances (e.g. are people single parents, do they have babies which don't sleep well, etc.)

Chapter 6 Software requirements

Solutions provided for Exercises 6.1, 6.3, 6.6, 6.7, 6.8 and 6.9.

6.1 **Functional requirements** that specify some services or functionality to be provided by the system.

Non-functional requirements that define operational constraints on the behaviour of the system

Design requirements that define constraints on the system design and implementation

Process requirements that define constraints on the system development process.

6.3 Ambiguities and omissions include:

- Can a customer buy several tickets for the same destination together or must they be bought one at a time?
- Can customers cancel a request if a mistake has been made?
- How should the system respond if an invalid card is input?
- What happens if customers try to put their card in before selecting a destination (as they would in ATM machines)?
- Must the user press the start button again if they wish to buy another ticket to a different destination?
- Should the system only sell tickets between the station where the machine is situated and direct connections or should it include all possible destinations?

6.6 Note that Figure 6.1 is a top-level requirements definition for the whole system. Figures 6.2 and 6.3 are more detailed function definitions.

1. Fuel delivery system

1.1 **The system should provide an unattended fuel delivery service where a specified amount of fuel is delivered to customers, The cost is deducted from the customer's credit card account.**

1.2 The sequence of actions to dispense fuel should be:

1. The customer selects the type of fuel to be delivered.
2. The customer inputs either a cash limit or a maximum amount of fuel to be delivered
3. The customer validates the transaction by providing credit card account details.

Rationale: The amount of fuel allowed depends on the credit limit but customers may wish to 'fill up' rather than have a specified amount of fuel. By specifying a maximum, the system can check if credit is available. Note that the definition does not set out how credit card details should be provided.

4. The pump is activated and fuel is delivered, under customer control.
5. The transaction is terminated either when the pump nozzle is returned to its holster for 15 seconds or when the customers fuel or cash limit is reached.

Rationale: Termination should not be immediate when the nozzle is returned as the customer may wish to restart the transaction e.g. to fill a fuel can as well as the car fuel tank. If a pump display is available, it may be appropriate to issue a 'Please wait for your receipt' message.

6. A receipt is printed for the customer.
7. The fuel stock is updated.

Specification: PUMP_SYS/FS. Section 1

Figure 6.1 Requirements for a fuel delivery system

2. Dispensing cash

2.1 **The system must provide a facility which allows a specified amount to cash to be issued to customers.** The amount is requested by the customer but the system may reduce this amount if the customer's daily limit or overdraft limit is reached.

2.1.1 The sequence of actions to dispense cash should be:

1. The customer inputs the amount of cash required
2. The system checks this against daily card limits and the customer's overdraft limit.
3. If the amount breaches either of these limits, then a message is issued which tells the customer of the maximum amount allowed and the transaction is cancelled.
4. If the amount is within limits, the requested cash should be dispensed
5. The customer's account balance and daily card limit should be reduced by the amount of cash dispensed.

Specification: ATM/Customer functionality/FS. Section 2.1

Figure 6.2 ATM system - cash dispensing

7.2 Spell checking

7.2.1 **The system shall provide a user-activated facility which checks the spelling of words in the document against spellings in the system dictionary and user-supplied dictionaries.**

7.2.2 When a word is found in the document which is not in any dictionary, a user query should be issued with the following options:

1. Ignore this instance of the word
2. Ignore all instances of the word
3. Replace the word with a suggested word from the dictionary
4. Replace the word with user-supplied text
5. Ignore this instance and add the word to a specified dictionary

7.2.3 When a word is discovered which is not in the dictionary, the system should propose 10 alternative words based on a match between the word found and those in the dictionaries.

Specification: NewWP/Tools/FS. Section 7.2

Figure 6.3 Spell checking

6.7 There are many possibilities here. Some suggestions are shown in Figure 6.4.

Non-functional requirement	Description	Examples
Performance	Performance requirements set out limits to the performance expected of the system. These may be expressed in different ways depending on the type of system e.g. number of transactions processed per second, response time to user requests, etc.	The system must process at least 150 transactions per second. The maximum response time for any user request should be 2 seconds.
Implementation	Defines specific standards or methods which must be used in the development process for the system	The system design must be developed using an object-oriented approach based on the UML process. The system must be implemented in C++, Version 3.0.
Usability	Defines requirements which relate to the usability of the system by end-users.	All operations which are potentially destructive must include an undo facility which allows users to reverse their action. (This is an example of a functional requirement which is associated with a non-functional requirement) All operations which are potentially destructive must be highlighted in red in the system user interface.
Safety	Safety requirements are concerned with the overall safe operation of the system	The system must be certified according to Health and Safety Regulations XYZ 123.

Figure 6.4 Non-functional requirements

6.8 Possible non-functional requirements for the ticket issuing system include:

1. Between 0600 and 2300 in any one day, the total system down time should not exceed 5 minutes.

2. Between 0600 and 2300 in any one day, the recovery time after a system failure should not exceed 2 minutes.

3. Between 2300 and 0600 in any one day, the total system down time should not exceed 20 minutes.

All these are availability requirements – note that these vary according to the time of day. Failures when most people are traveling are less acceptable than failures when there are few customers.

4. After the customer presses a button on the machine, the display should be updated within 0.5 seconds.

5. The ticket issuing time after credit card validation has been received should not exceed 10 seconds.

6. When validating credit cards, the display should provide a status message for customers indicating that activity is taking place.

This tells customer that the potentially time consuming activity of validation is still in progress and that the system has not simply failed.

7. The maximum acceptable failure rate for ticket issue requests is 1: 10000.

Note that this is really ROCOF. I have not specified the acceptable number of incorrect tickets as this depends on whether or not the system includes trace facilities that allow customer requests to be logged. If so, a relatively high failure rate is acceptable as customers can complain and get refunds. If not, only a very low failure rate is acceptable.

6.9 Keeping track of the relationships between functional and non-functional requirements is difficult because non-functional requirements are sometimes system level requirements rather than requirements which are specific to a single function or group of functions.

One approach that can be used is to explicitly identify system-level non-functional requirements and list them separately. All system requirements which are relevant for each functional requirement should be listed. Then produce a table of requirements as shown in Figure 6.5.

Functional requirement	Related non-functional system requirements	Non-functional requirements
The system shall provide an operation which allows operators to open the release valve to vent steam into the atmosphere.	Safety requirement: No release of steam shall be permitted if maintenance work is being carried out on any steam generation plant.	Timing requirement: The valve must open completely within 2 seconds of the operator initiating the action.

Figure 6.5 Functional and non-functional requirements

Notice that in this example, the system non-functional requirement would normally take precedence over the timing requirement which applied to the specific operation.

Chapter 7 Requirements engineering processes

Solutions provided for Exercises 7.1, 7.4, 7.6, and 7.9.

7.1 The stakeholders in a student records system include:

- University central administration including those responsible for registration, payment of fees, examinations and assessment and graduation.
- The students whose details are recorded in the system.
- University departmental administrators who supply information to the system and use information from it.
- Academic staff who use information from the system.
- Data protection officers (local and national).
- Potential employers of students (who may require information from the system).

7.2 Solution to be added.

7.3 You can tackle this problem using a brainstorming approach. Obviously, there are many alternatives to the solutions suggested here. Note the printing conflict is deliberate.

Viewpoint: Library manager

Requirement: Access to the LIBSYS system shall be restricted to accredited users of the library.

Requirement: The LIBSYS system shall provide a reporting facility that allows usage reports (who used the system, how often, what libraries were accessed) to be created and printed.

Requirement: The LIBSYS system shall be configured so that only document printing on specific library servers is permitted.

Viewpoint: Users

Requirement: The LIBSYS system shall be accessible from any location, including locations away from the university campus.

Requirement: It shall be possible to save LIBSYS queries, recall them and modify them for subsequent use.

Requirement: The LIBSYS system shall allow documents to be printed on user printers.

Viewpoint: System managers

Requirement: The restart time of the LIBSYS system after failure shall not exceed 5 minutes.

Requirement: The LIBSYS system shall provide a backup facility for user's personal workspaces.

Requirement: The LIBSYS system shall be available for a range of platforms including Windows 2000, Windows XP and Mac OS X.

7.4 Important non-functional attributes for the cataloging services might be:

- Availability (because the system may be required at any time)
- Security (because the books data base musn't be corrupted)
- Efficiency (because the system must respond quickly to each transaction)

For the browsing services, usability is also very important as these services should be easy to use without extensive training.

7.6 An example of a system where social and political factors influence system requirements is a system to manage the costs of public healthcare. Politicians are anxious both to control costs and to ensure that the best public image of the healthcare system is provided. There is a potential conflict in such a system between administrators who are driven by treatment costs and doctors who are (or should be) driven by treatment effectiveness. The requirements for the system might therefore embed particular policies which are included as a result of rganizations factors (e.g. ensure that administrators can vet treatment costs) rather than technical requirements.

7.9 Figure 7.2 shows a change process which may be used to maintain consistency between the requirements document and the system. The process should assign a priority to changes so that emergency changes are made but these changes should then be given priority when it comes to making modifications to the system requirements. The changed code should be an input to the final change process but it may be the case that a better way of making the change can be found when more time is available for analysis.

7.10 The best way to tackle this problem is to demonstrate by example that the analysis method is inadequate. You should prepare a scenario of system use where social factors are important then try to represent this using the notations proposed in the analysis method. If this is impossible, you have then demonstrated that the standard is incomplete.

However, this does not mean the method should completely discarded. Rather, you should discuss with your manager how the method can be supplemented with additional information to represent e.g. social factors.

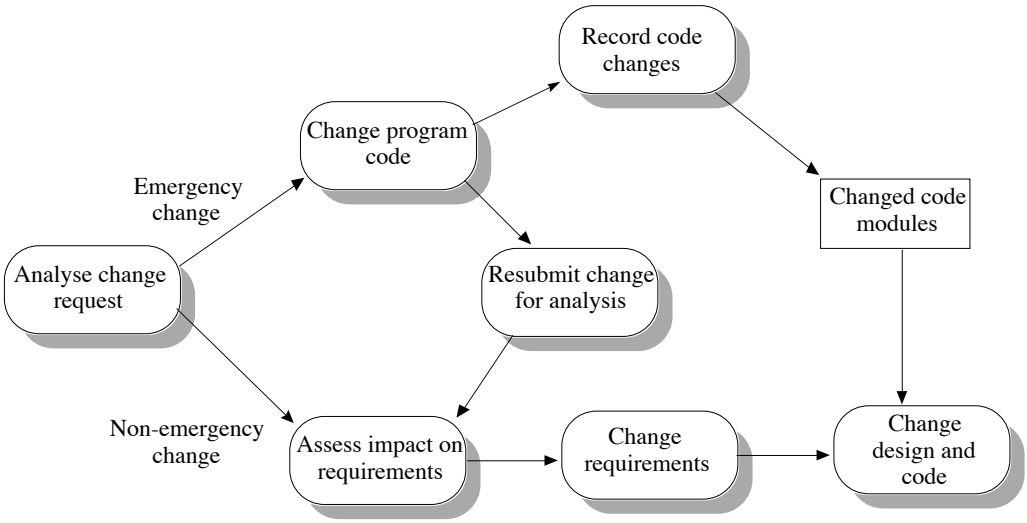

Figure 7.2 A change process for emergency changes

Chapter 8 System Models

Solutions provided for Exercises 8.1, 8.2, 8.5, 8.8, and 8.9.

8.1 There are obviously many different possibilities here depending on exactly what systems are included. One possible model is shown in Figure 8.1.

8.2 One possible model of the data processing is shown in Figure 8.2. There are other alternatives depending on the details of the system. Note that this DFD has to include some control data as this is an event-driven system.

8.4 Solution to be added.

8.5 See Figure 8.3.

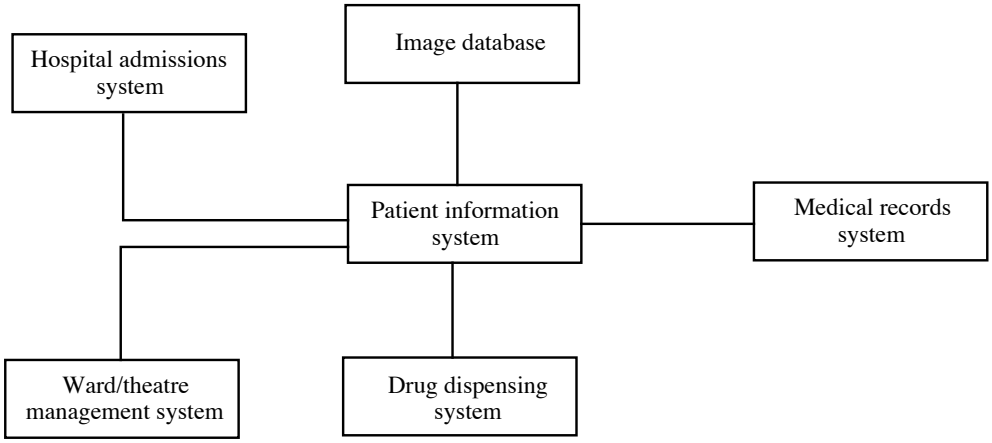

Figure 8.1 Context model of a patient information system

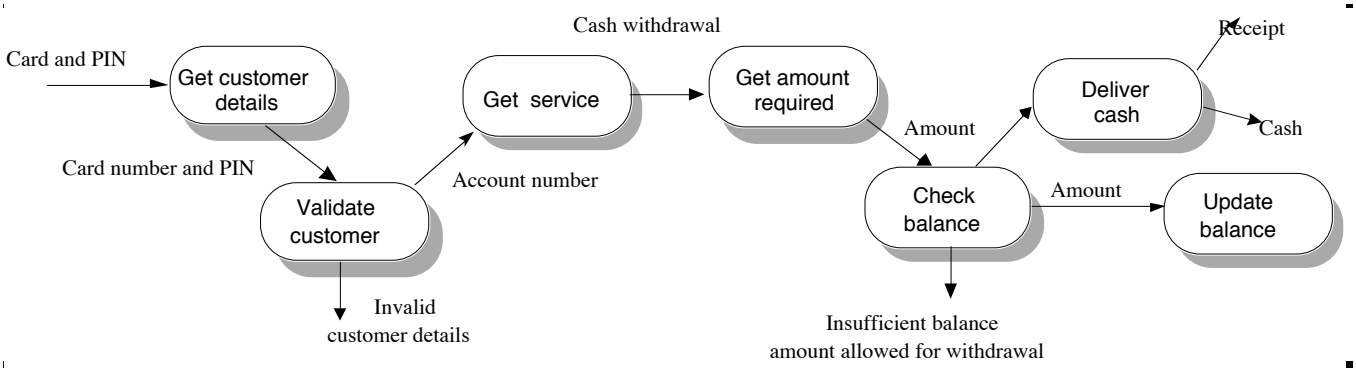

Figure 8.2 Data processing for cash withdrawal in an auto-teller system

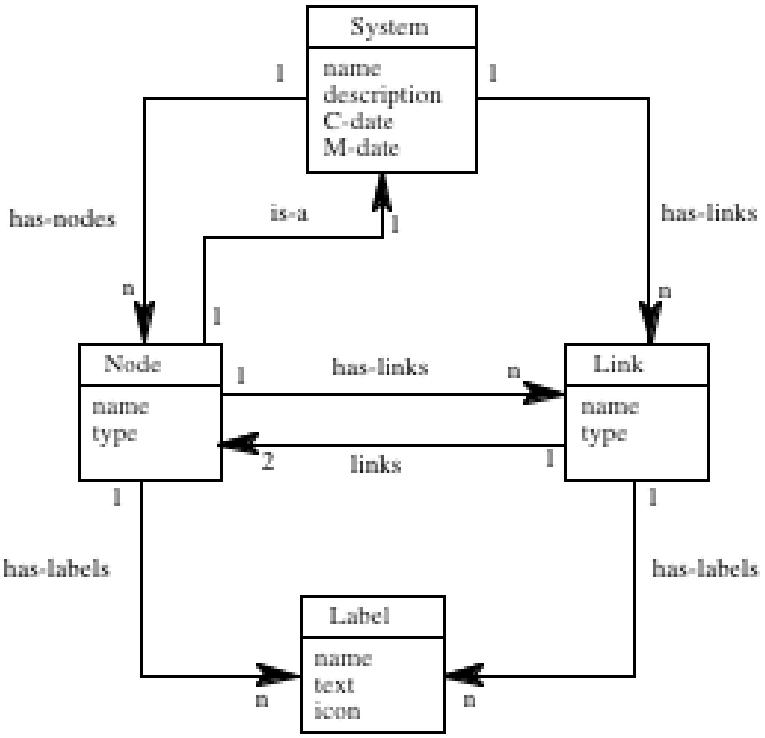

Figure 8.3 Semantic model of a software system

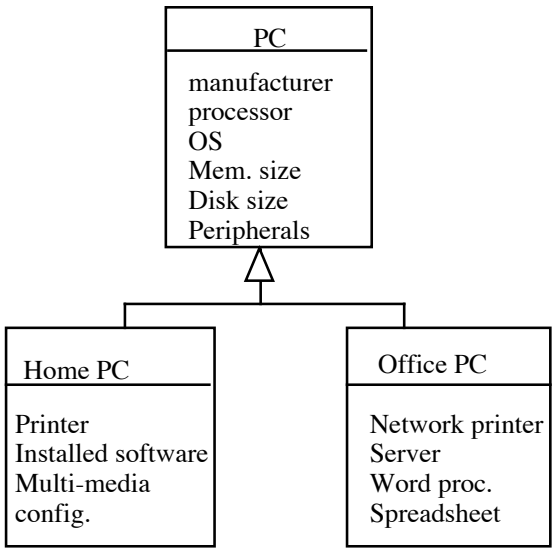

Figure 8.4 Class hierarchy for a PC

8.8 There are many possible rganizations for the class hierarchy. I show a simple one in Figure 8.4 with only two levels. A three-level hierarchy would also be OK but more than that would be too much. The aggregation diagram shows the part-of relationships between objects. This is shown in Figure 8.5. Obviously, further decomposition of the lowest level is possible.

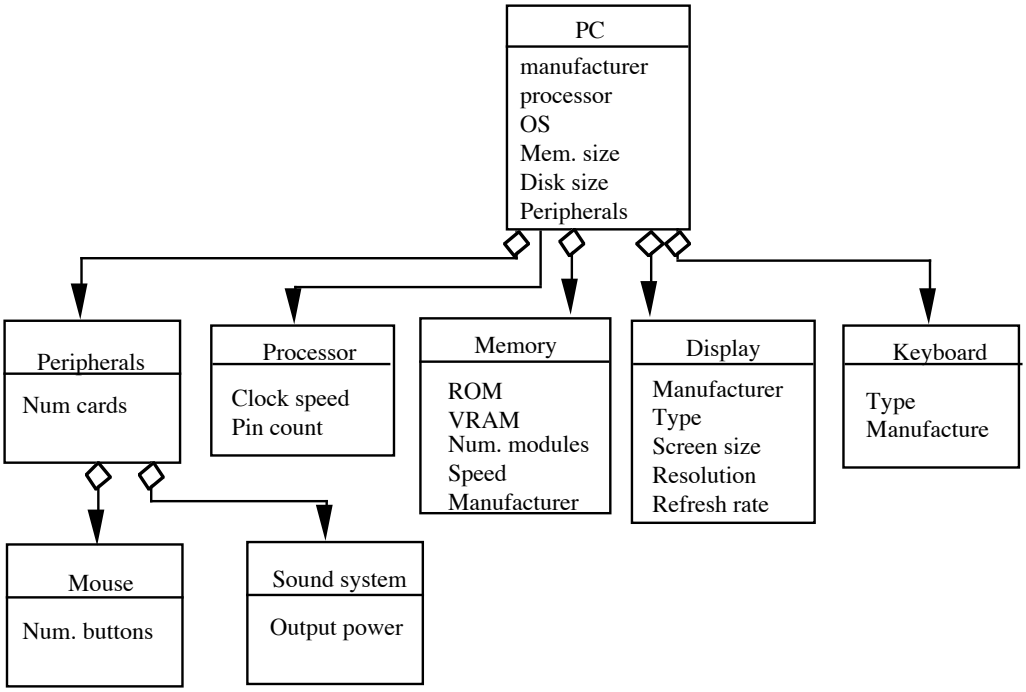

Fig 8.5 Aggregation diagram for a PC

8.9 See Figure 8.6.

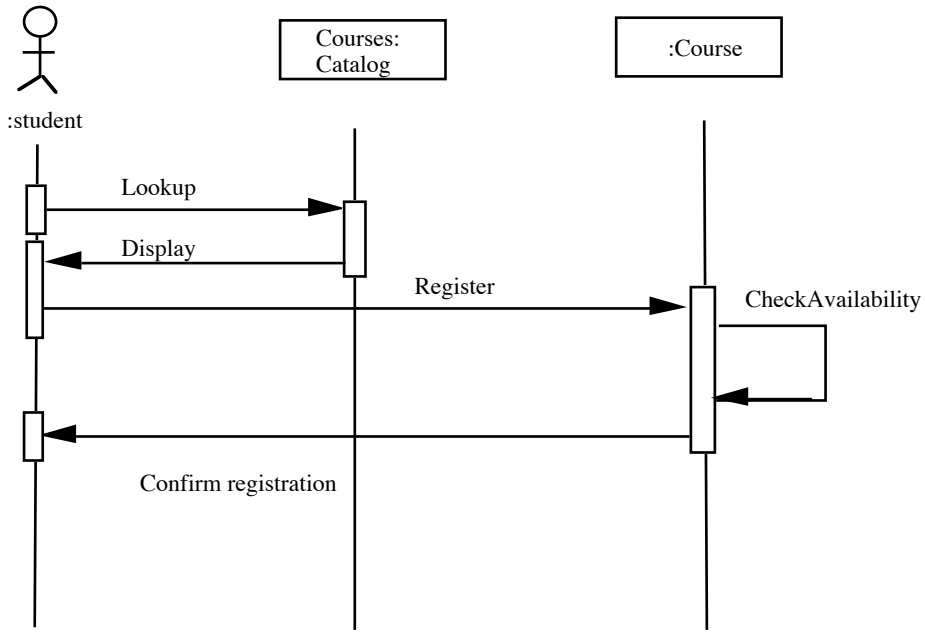

Figure 8.6 Sequence diagram of course registration

8.10 Structured methods are not advisable when a prototyping approach is adopted to software development. Structured methods involve developing a set of system models and maintaining the consistency of these models when the system is changing rapidly is very expensive. In practice, the models are unlikely to be maintained.

They are also unsuitable when software systems are developed by configuring existing off the shelf systems. In those cases, the architecture of the system is determined by the underlying systems and there is little to be gained by producing a set of system models.

Chapter 9 Critical Systems Specification

Solutions provided for Exercises 9.2, 9.3, 9.6, 9.8, 9.9, 9.10 and 9.12.

9.2 Possible user errors are:

Maximum daily dose set wrongly
Maximum single dose set wrongly
Failure to replace empty insulin reservoir
Insulin reservoir improperly fitted
Needle improperly fitted

Examples of safety requirements to avoid these errors are:

1. When the maximum dose and the maximum daily dose is changed, the user should be asked to input the changed values twice.
2. If the maximum daily dose has already been set by the user then the new daily dose should be no more than 1.25 and no less than 0.75 of the previous maximum daily dose.
3. The insulin reservoir case should be designed so that it is only possible to fit the insulin bottle the right way and the case should not close unless the bottle is properly seated.
4. If the back pressure from the needle assembly is more than XX then the system should shut down and issue an audible and text warning. (this caters for blocked needles as well as improperly fitted needles).

Other examples are, of course, possible.

9.3 *Hazards*:

1. Incorrect dosage of radiation computed
2. Radiation delivered to the wrong site on patient's body
2. Data for wrong patient used to control machine
3. Data transfer failure between database and therapy machine

Software protection:

1. Comparison with previous doses delivered. Establishment of a maximum monthly dose which may never be exceeded. Feasibility checks (e.g. for negative dosages). Confirmation of dose to be delivered by operator. Continuous visual display of dose being delivered.

2. Comparison with delivery site in previous treatment. Light used to illuminate site of radiation delivery. Operator confirmation of site before machine can operate.

3. Patient asked to verify name, address and age before machine starts by pressing button. Issue patient with a personal treatment card which is handed over to identify patient. Maintain separate list of patients to be treated each day and correlate with patient databases. Force machine operator to verify list and database consistency before starting machine.

4. Dual display of information in therapy machine and database. Highlighting of differences in operator display. Locking of machine until information is consistent. Use of check digits and other error checking codes in the data. Duplicate communication channels between machine and database.

9.6 It is not usually appropriate to use hardware reliability metrics because of the different types of failure which normally occurs in hardware. Most hardware system failures are a result of component failures due to faulty manufacture or because a component has come to the end of its normal life. Once a component has failed, it must be replaced or repaired before normal system services can be resumed. However, most software failures are transient and are a

consequence of design errors or timing problems. The component can continue to deliver normal service without repair.

Hardware metrics such as 'mean time to failure' are based on component lifetimes and therefore cannot be applied directly to software systems.

An example might be a bank teller system which includes a hardware component to open the door to deliver cash and a software component to deliver signals to that door. When the hardware component fails, the whole system is out of action until that component is repaired. If the software component fails in one specific circumstance then cash may not be delivered in that case but delivery could resume with the next transaction.

9.8 See Figure 9.1. Note that the values in this table are really quite arbitrary and you need to know more about the domain to set accurate values. Any values which take into account the type of system involved are equally good.

System	Reliability metric	Suggested value	Rationale
Patient monitoring system	Availability	System should be unavailable for less than 20 minutes per month.	The system needs to be continuously available as patients may be admitted or discharged at any time. The chosen figure is acceptable because, if necessary, critical system functions can be taken over manually.
Word processor	ROCOF	Failures resulting in loss of data should not occur more than once per 1200 hours of use.	
Vending machine controller	POFOD (Probability of failure on demand)	Failure acceptable in 1:5000 demands	Not a critical system so relatively high failure rate is OK
Braking system controller	POFOD	The software should never fail within the predicted lifetime of the system.	Very critical system. Failure is unacceptable at any time.
Refrigeration unit control	Availability	20 minutes per month	Non-stop system but not critical. Short periods of failure are not a real problem as temperature takes some time to rise.
Management report generator	ROCOF	1 fault/100 hours of use	Not a critical system. Faults are unlikely to cause severe disruption

Figure 9.1 Reliability specification

9.9 *Data base corruption* Serious. Appropriate metric is rate of fault occurrence (ROCOF). Appropriate time unit is number of transactions. Reliability specification is based on no more than 1 fault per week. ROCOF = 1/Estimated number of weekly transactions.

Lack of system service Appropriate metric is availability. Appropriate time unit is calendar time. If normal opening hours are from 8 am to 8 pm, system should be down for no more than 5 minutes per 12 hour period. Availability is therefore 1/144.

Incorrect information delivery to terminal Appropriate metric is rate of fault occurrence. Appropriate time unit is number of transactions. Reliability specification is based on no more than 1 fault per day. ROCOF = 1/Estimated number of daily transactions.

9.10 The most appropriate reliability metric is Probability of Failure on demand (POFOD). This is the probability that the system will respond correctly when a request is made for service at a given point in time. This metric is used for protection systems where demands for service are intermittent and relatively infrequent over the lifetime of the system.

There are several different possibilities here (some examples below)

1. The system shall ensure that the train brakes are applied when a 'red signal' is received.
2. The system shall sound an alarm in the driver's cabin when a 'red signal' is received.
3. The system shall compare the train speed with the segment speed limit once per second.
4. If the train speed exceeds the segment speed limit and the train throttle position is not zero then the throttle position should be reset to zero.
5. If the train speed exceeds the segment speed limit and the train deceleration is less than the comfortable decleration limit then the train brakes should be applied.

9.12 There is no definitive answer for this question. However, I would expect students to demonstrate that they understand this by putting forward coherent arguments for and against whatever position that they choose to adopt. They should show that they understand that hacking is unlawful whatever position that they adopt but in some cases it can be ethically justified. They should also show that they understand in this case that there could be negative as well as positive results of unauthorised access.

For example, acceptance of the request. Hacking is unlawful but, in this case, the benefits which might accrue from accessing the data justify the breaking of the law. Accepting the request does not presuppose guilt on the part of the company - if the request is not accepted, they may be unjustly accused of supplying torture equipment if, in fact, they do not do so. However, using unauthorised means to access data might damage the standing of the campaign especially if the company is not guilty and access is detected. Therefore, it may do more harm than good.

Chapter 10 Formal Specification

Solutions provided for Exercises 10.1, 10.2, 10.4, 10.6, 10.7 and 10.8.

10.1 The architectural design is a means of structuring the system into (relatively) autonomous parts which can be separately specified using formal or other techniques. This serves to structure the specification as well as the system. Of course, the specification could be structured in some other way but then there is a problem of mapping the specification to the system structure.

10.2 To explain the advantages of formal specification to practising engineers, it is important to focus on what it brings to the practice of software development rather than on more abstract advantages such as the ability to mathematically analyse the specification. Advantages that might be stressed are:

1. The detailed analysis of the requirements that is necessary to produce a formal specification. This results in the discovery and resolution of ambiguities and errors at an early stage in the process.

2. The unambiguous specification of interfaces. Interface problems are one of the major problems in system integration and a reduction in such problems can significantly reduce software costs.

3. The ability to mix formal and informal specifications. The whole system need not be formally specified but only those parts where most benefit can be gained.

10.4 See Figure 10.1.

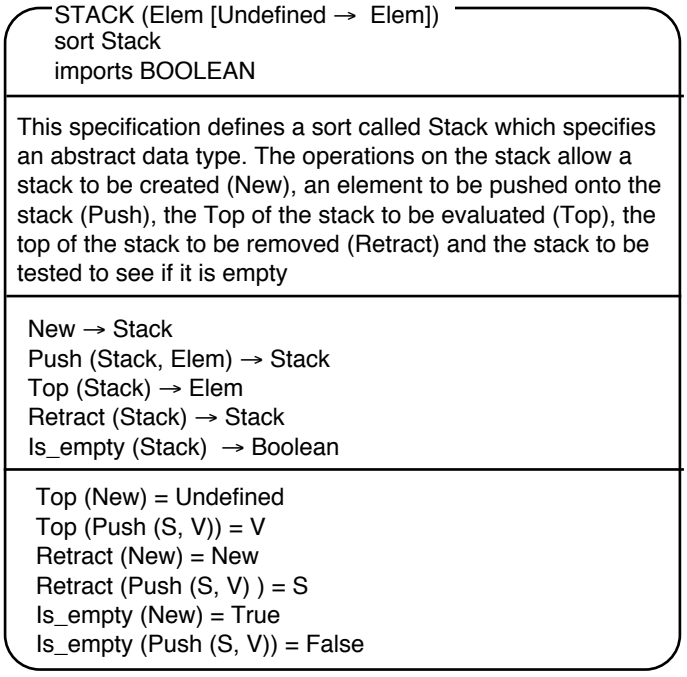

Figure 10.1 Stack specification

10.6 The Z schemas are shown in Figure 10.2. There are many different possibilities here depending on how much information is maintained. This is one of the simplest which

assumes that customers may not withdraw money if there is an insufficient cleared balance in their account (a cleared balance is where all cheques paid in to the account have been cleared for payment).

BankAccount

CustomerNumber: ℕ
AccounrNumber: ℕ
DateOpened: Date
DateClosed: Date
CustomerPIN: ℕ
Balance: Money
ClearedBalance:Money
AvailabletoWithdraw: Money

DateOpened < DateClosed
ClearedBalance ≤ Balance
AvailabletoWithdraw ≤ ClearedBalance

Validate

Ξ BankAccount
Account? ℕ
PIN?: ℕ
TransactionOK!: boolean

Account? = AccountNumber
PIN? = CustomerPIN
TransactionOK!'

Withdraw

DBankAccount
Amount?: Money

Amount? <= ClearedBalance
Amount? <= AvailabletoWithdraw
Balance' = Balance - Amount?

Figure 10.2 Bank account schemas

10.7 To specify that the manual delivery button can only have a non-zero value if the switch is in the manual position, you should add the following invariant to the state schema.

$$switch? \neq manual \Rightarrow ManualDeliveryButton = 0$$

10.8 The SELF_TEST schema is shown in Figure 10.3.

SELF_TEST

Δ INSULIN_PUMP_STATE

(HardwareTest? = OK ∧ Needle? = present ∧ InsulinReservoir? = present ⇒
status' = running ∧ alarm! = off ∧ display1!= "") ∨
(
status' = error
alarm! = on
(
Needle? = notpresent ⇒ display1! = display1! ∪ "No needle unit" ∨
(InsulinReservoir? = notpresent ∨ insulin_available < max_single_dose)
⇒ display1! = display1! ∪ "No insulin" ∨
HardwareTest? = batterylow ⇒ display1! = display1! ∪ "Battery low" ∨
HardwareTest? = pumpfail ⇒ display1! = display1! ∪ "Pump failure" ∨
HardwareTest? = sensorfail ⇒ display1! = display1! ∪ "Sensor failure" ∨
HardwareTest? = deliveryfail ⇒ display1! = display1! ∪ "Needle failure" ∨
)
)

Figure 10.3 Self-test schema

The RUN schema should be modified to check that HardwareTest? is true before continuing operation.

Chapter 11 Architectural Design

Solutions provided for Exercises 11.1, 11.4, 11.5, 11.6, 11.7, 11.8 and 11.9.

11.1 The architecture may have to be designed before specifications are written to provide a means of structuring the specification and developing different sub-system specifications concurrently, to allow manufacture of hardware by sub-contractors and to provide a model for system costing.

11.4 *Ticket issuing system*. The most appropriate architectural model is a centralised model with a shared repository of route and pricing information. This means that changes are immediately available to all machines. As little local processing is necessary, there is no real advantage in a client-server architecture. The centralised system also allows global information and route use to be collected and processed.

Video conferencing system. The most appropriate is a client-server model. The reason for this is the need for a lot of local processing to handle multimedia data.

Robot floor cleaner. The most appropriate model is a repository model where all sub-systems place information in the repository for other sub-systems to use. In the case of AI systems as this would be, a special kind of repository called a blackboard is normally used.

11.5 a) Should have a centralised database with sub-systems to handle communications, route information and price information. Also sub-systems for statistical processing. Each ticket machine should be connected to this.

b) Should include a network with a range of clients/servers on it. These should include a floor controller, video server, display clients, etc.

c) Should include a centralised repository with sensors adding information to it, decision systems taking information from it and actuators using sensor information to move the machine.

11.6 The call return model assumes a sequence of actions whereas real-time systems must respond to events from different hardware interfaced to the system. This normally means that the control must be responsive rather than sequential.

11.7 The most appropriate control models for the systems suggested are:

- Salary system. Call return model of control. Each operation involves identifying particular options then calling subroutines to retrieve or compute the required information. There are no unexpected events to be processed
- Software toolset. Broadcast model of control is most appropriate. Tools need not know which other tools are available and this approach allows tools which operate on different types of computers to work together.
- Television controller. Centralised (polling) control model. This is the most appropriate approach as there is no need for the very rapid response required from interrupt driven systems.

11.8 Both models can be distributed where each transformation in a DFD is implemented as a separate process and each object is implemented as process. Problem with functional decomposition is the need for shared state which must also be implemented as one or more processes. In the object model, distributing objects is a problem if inheritance is involved as this creates a lot of network traffic.

11.9 You can make the comparison between the CASE toolsets by taking the different components of the reference model in turn than assess how well the CASE toolset being studied provides these services. You also have to look at how these services are used in particular toolsets. In this case, comparisons would be drawn using:

1. Data repository services. What kind of data management is supported?

2. Data integration services. How well can data be interchanges and what support is provided for configuration management?

3. User interface services. What facilities are supported to allow presentation integration? How well integrated at the user interface level are different parts of the systems?

4. Task management services. Do the toolsets provide for process definition and enactment? What process models (if any) do they assume?

5. Message services. How do different tools in the toolset communicate?

Chapter 12 Distributed systems architectures

Solutions provided for Exercises 12.2, 12.3, 12.4, 12.6, 12.8 and 12.10.

12.2 In a fat-client system, some of the application processing is carried out on the client whereas in a thin client system only the user interface is displayed on the client and all of the application processing is carried out on the server. The use of Java blurs the distinction as it allows the development of applets which can be downloaded to the client at execution time. Depending on the functionality of the applet, this allows a degree of control over the processing that is carried out on the client.

12.3 In this case, I would chose a fat client model with company information located on a central server (this is critical information and its important that it is consistent for all dealers). Simulations would run on the dealers workstation as these are used in different ways depending on the individual dealers.

12.4 Problems might arise when converting the legacy system to a client-server architecture because there is no clear separation between data management, application processing and information presentation. In older systems, these are usually all intertwined and it may be very difficult to disentangle these systems to produce the required separation.

As an example of this, consider the processing of inputs produced by the user from some standard form. The presentation of the form and the collection of data is clearly a presentation activity but, as the data is input, it is validated by the system. Some of this validation is in the data management system (e.g. is a number within range) but some is application specific and relies on checking relationships between different fields in the form (e.g. in a medical records form, if the sex is male then there should not be a record under the date of birth of the first child) that are not simple data checks.

12.6 The use of distributed objects and an ORB simplifies the implementation of scaleable systems as it is very easy to add additional servers in the form of distributed objects to the system. These can be introduced without perturbing other parts of the system. For example, a web server can be easily replicated as the number of users increases.

12.8 Advantages of decentralized p2p architectures:

Less vulnerable to denial of service attack

Scaleable – system performance should not be adversely affected by adding peers.

Disadvantages of decentralized p2p architectures:

Peer communications more difficult to organize.

More expensive to discover what peers are available on the network.

Advantages of semi-centralised p2p architectures:

Easy to keep track of what peers are available.

Peer data exchange is simplified – it takes place via the server.

Disadvantages of semi-centralised p2p architectures:

Failure of server results in system unavailability.

Vulnerable to denial of service attack on server

12.11 The in car information system is an embedded system with limited functionality and processing power. The aggregation service runs on a powerful external server so its operation is not limited by resources such as memory or processor capability. It is therefore much simpler for the aggregation service to cope with failures of individual information provision services rather than embed this in the local in-car system. It need only cope with binding to alternative aggregation services.

Chapter 13 Application architectures

Solutions provided for Exercise 13.2, 13.3,13.4, 13.7, 13.8 and 13.10.

13.2 Point of sale system: Transaction processing

Subscription reminder system: Batch processing

Photo album system: Event processing

Web page reader: Language processing

Interactive game: Event-processing

Inventory control system: Transaction processing

13.3 A possible data flow diagram for 'Compute salary' is shown in Figure 13.1. Obviously there are many variations to this.

13.4 A transaction mechanism is necessary to ensure database consistency. As a transaction is an atomic operation, all of the changes in the transaction are not committed until it is complete. If these changes were made in sequence without a transaction mechanism, a system failure could mean that the update of the database was unfinished and the data left in an inconsistent state.

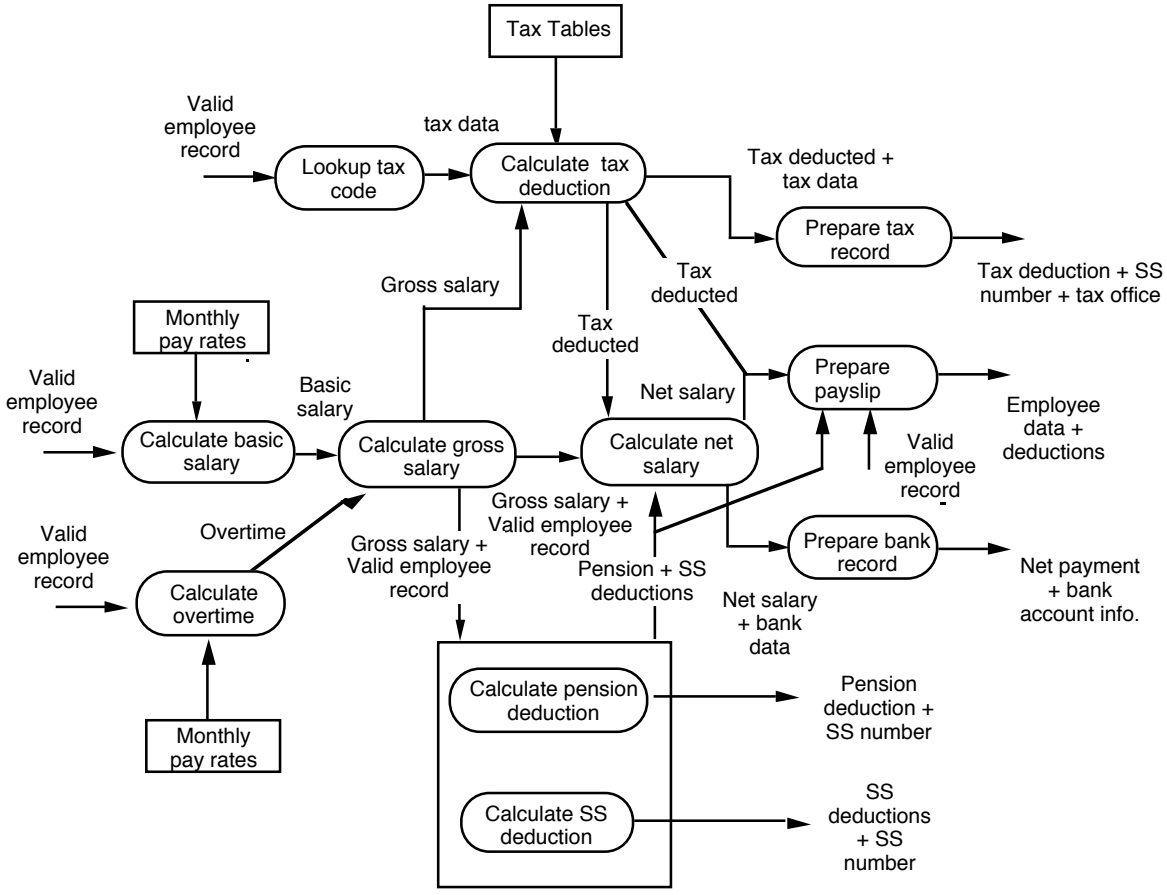

Figure 13.1 Data flow diagram for salary computation

13.7 The Event object communicates directly with the editor data structure to allow more efficient operation. Some commands that are implicit, such as inserting a character by pressing a key on a keyboard, require very fast response and, rather than lookup the command in a separate object, the event processing object interprets these directly and makes changes to the data structure.

13.8 A possible generic architecture for a spreadsheet is shown in Figure 13.2.

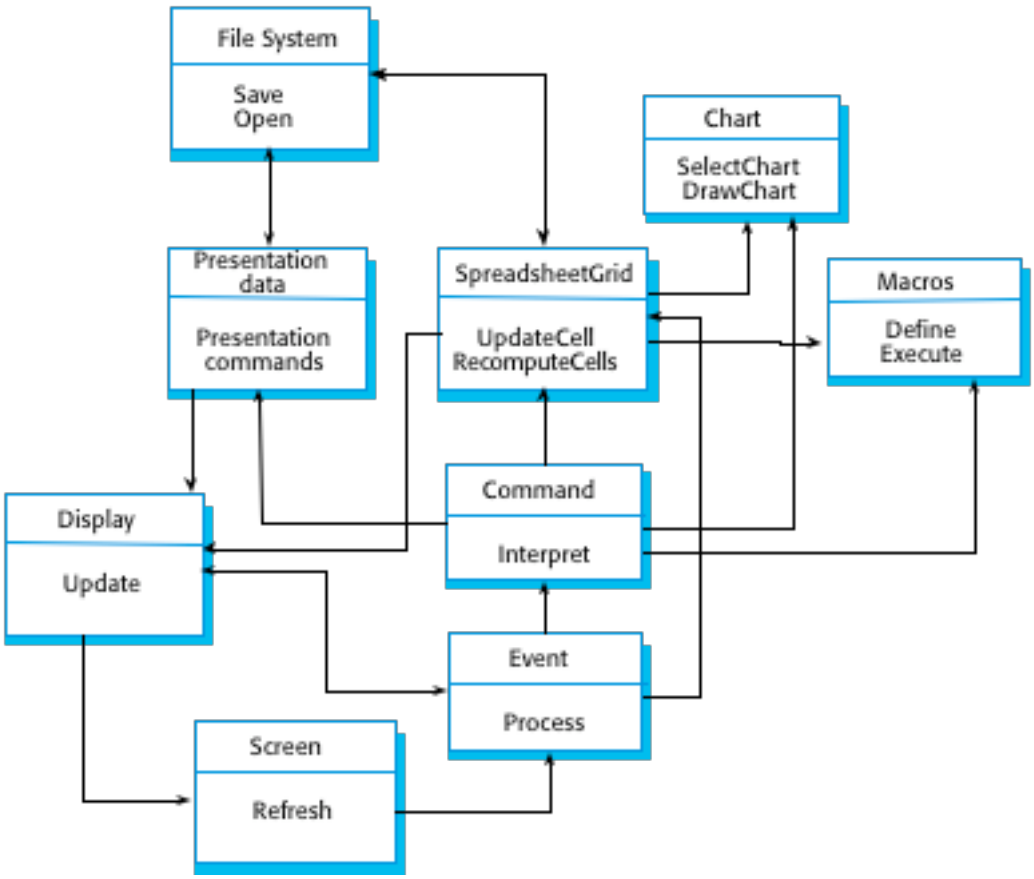

Figure 13.2 Spreadsheet architecture

13.10 A possible architecture for the natural language command processing system is shown in Figure 13.3 – I have used a pipeline rather than a repository model. I assume that the analysis process involves parsing the natural language command to identify the action then using expected parameters for that action to complete the generation. The system builds an abstract syntax tree and then generates SQL commands by traversing that tree. A dictionary is used to identify the parts of speech of the words in the command before parsing.

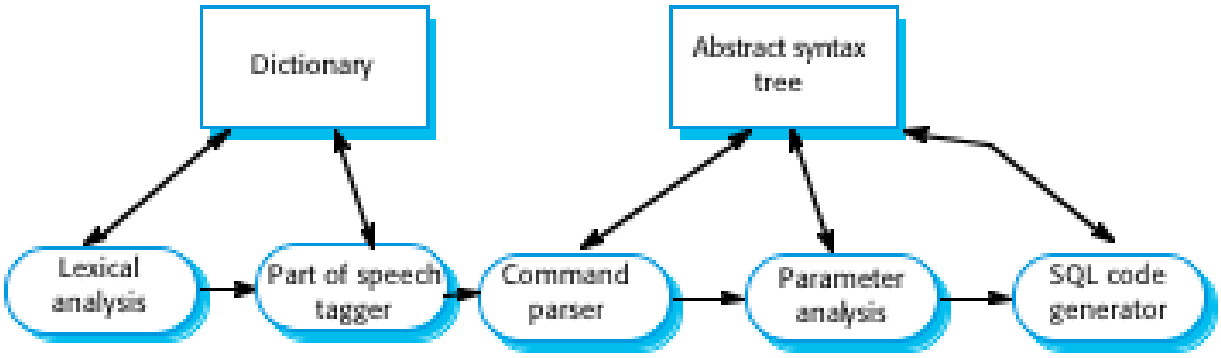

Figure 13.3 SQL generator from natural language text

Chapter 14 Object-oriented Design

Solutions provided for Exercises 14.1, 14.2, 14.3, 14.4, and 14.7.

14.1 There are two major problems encountered when modifying systems

1. Understanding which entities in a program are logically part of some greater entity

2. Ensuring that changes do not have unanticipated side-effects i.e. a change to one entity has an undesirable effect on some other entity.

Object-oriented development helps to reduce these problems as it supports the grouping of entities (in object classes) so therefore simplifies program understanding. It also provides protection for entities declared within objects so that access from outside the object is controlled (the entity may not be accessible, its name may be accessible but not its representation or it may be fully accessible). This reduces the probability that changes to one part of the system will have undesirable effects on some other part.

14.2 An object class is a generic description of a set of entities (or objects) which have common characteristics and which are recognisably the same in some or all respects. Objects are specific instances in the real-world or in a system where values have been assigned to the characteristics defined in the object class. The set of values assigned to the object characteristics may distinguish that object from all other objects but need not do so. In the real-world, we only see objects and construct object classes as abstract entities. In programs, we often only define object classes and construct objects whose lifetime is no longer than the execution time of the program.

An example of an object class is a BOOK which has attributes (characteristics) such as AUTHOR, TITLE, PUBLISHER, DATE OF PUBLICATION, etc.

An example of an object or instance of this object class is the specific book:

AUTHOR:	Ian Sommerville
TITLE:	Software Engineering
EDITION:	7
PUBLISHER	Addison-Wesley
DATE	2004

If we wished to define a book object which was distinct from all other objects, we would need to add another characteristic to the object class such as OWNER.

14.3 Concurrent objects may be used in a system that you know will be distributed or in a real-time system where objects are associated with autonomous sensors or actuators. The real-time system should be a 'soft' real-time system normally as it is often difficult to compute deadlines when object-oriented programming is involved because of the unpredictable overhead when methods are called.

14.4 Objects shown in Figure 14.1

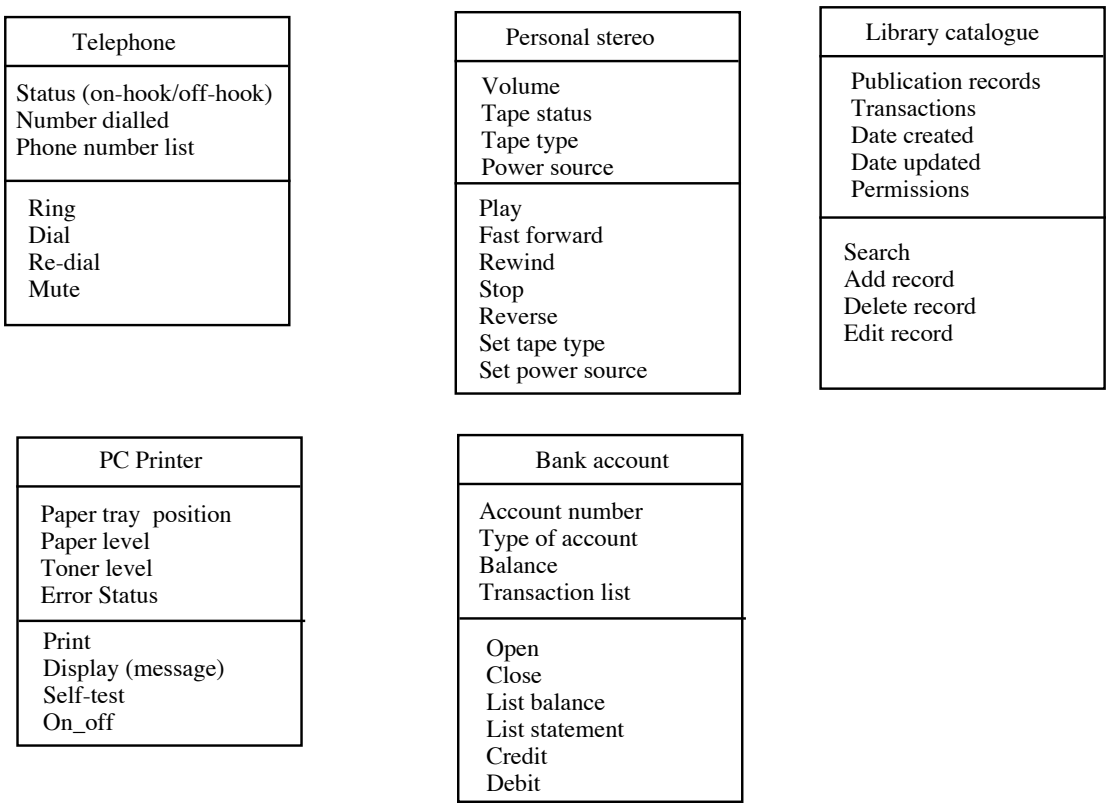

Figure 14.1 Objects, attributes and operations

14.7 In this example, I have shown possible principal objects, their attributes and operations (Figure 14.2). Notice that there is a single diary object with different operations for group appointments and personal appointments. Each participant in the system has her/his own personal diary plus a shared group diary.

Object	**Attributes**	**Operations**
Diary	Year Weeks_of_year Days_of_week Time_slots Access permissions	Make appointment Cancel appointment Move appointment Make group appointment Find free slot Reserve slots Book slots Free slots Display diary Check_slot_status
Appointment	Time Duration Place Participants Reason	
User	Diary	Check time slot

Figure 14.2 Diary objects, attributes and operations

(b) I have not produced a complete design here but have simply identified the objects, their attributes and the operations associated with each object in the fuel tank system in Figure 14.3 and provided a partial description of the system controller. Note the overall system controller coordinates communication between the card reader and the pump.

Object	Attributes	Operations
Pump	Fuel dispensed Price Hose status Trigger status Fuel type	Activate Deactivate Deliver fuel Stock update
Card reader	Card number Card type Card status Credit limit	Read card Check status Print receipt
Fuel tank	Current fuel level	Add fuel Remove fuel
Communication system	Number dialled Credit limit	Send card number Return card status
System controller	Card number Card type Max delivery Price table Fuel delivered	
Price table	Fuel prices	Lookup Amend price

Figure 14.3 Objects in the fuel delivery system

The basic sequence of actions in the controller (which could be described as a sequence diagrams) that controls the operation of the system is:

```
cardReader.Read_card (Card number, Card_type) ;
communicationsSystem.Check_status (Card_type, Card_status, Credit_limit) ;
// Actions taken when invalid card are not shown
if (Card_status == OK )
{
    Max_delivery := Credit_limit / Price_table.Lookup (Pump.Fuel_type) ;
    Pump.Set_maximum (Max_delivery) ;
    Pump.Activate  ( Fuel_delivered ) ;
    Pump.Deactivate ;
    Fuel_tank.Remove ( Fuel_delivered ) ;
    Card_ reader.Print_receipt ( Fuel_delivered * Price_table.Lookup (Pump.Fuel_type)) ;
}
```

Chapter 15 Real-time software design

Solutions provided for Exercises 15.2, 15.3, 15.7, 15.8 and 15.9.

15.2 An object-oriented approach may result in unacceptable timing delays because structuring a system into objects probably means that there will be a large number of small tasks currently active in a system. The overhead of task communications will slow down the system and may cause timing problems.

15.3b The state diagram depends on the range of facilities on the CD player. See Figure 15.1.

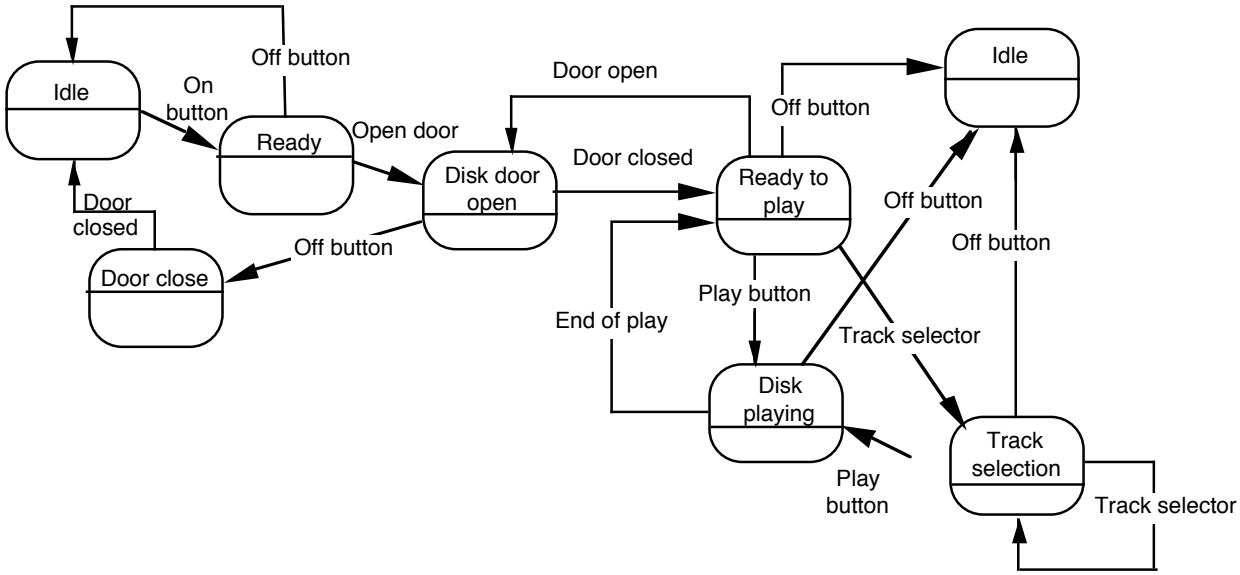

Figure 15.1 State diagram for CD player

15.3c The possible states depend on the detailed facilities of the machine. Figure 15.2 represents one of many possible solutions.

15.7 The stimuli and responses are shown in Figure 15.3.

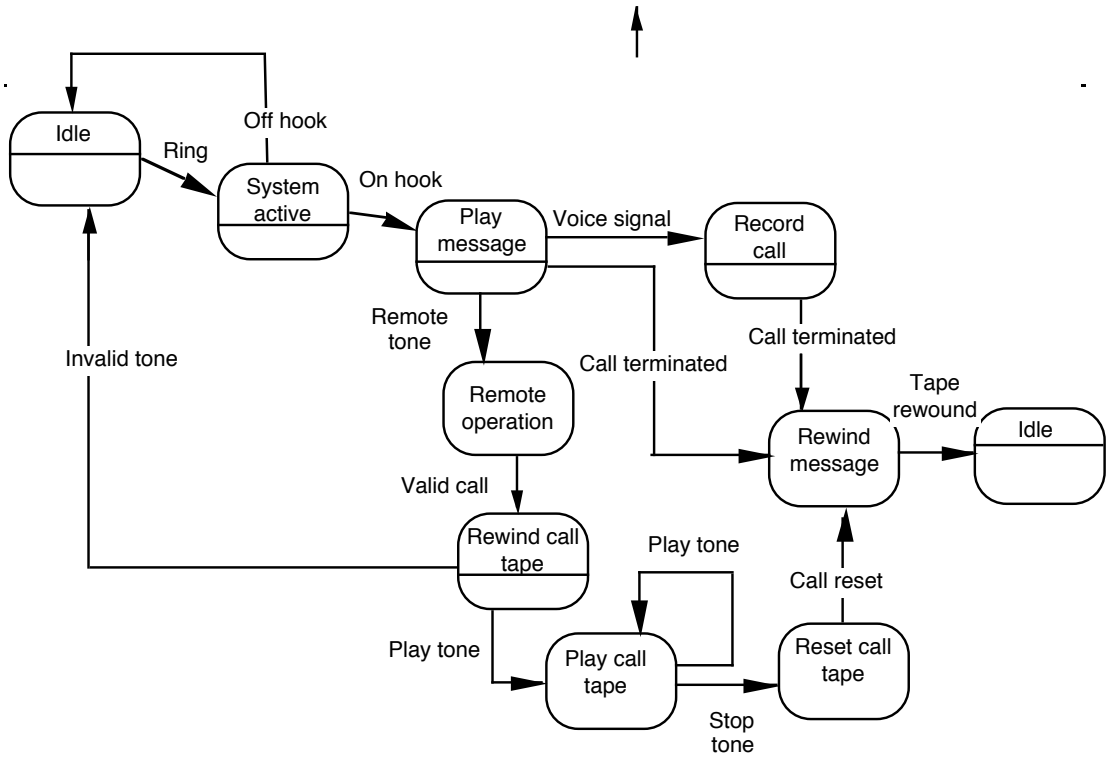

Figure 15.2 State diagram for telephone answering machine

Stimulus	Associated response
Train speed information	Check that the speed is within the speed limit for the current segment. If speed limit is exceeded, audible warning in driver's cabin
Brake status information	If the train speed is within the speed limit, no action; if the train speed exceeds the speed limit then the brakes should be applied
Trackside transmitter signal	Set speed limit for current segment. Set status of current segment. If status is 'red', apply the train brakes

Figure 15.3 Stimuli and responses for train system

15.8 What I am looking for here is a diagram which shows a process associated with each type of sensor (trackside transmitter, train speed and brake status), a data processing process which carries out the computations and a set of processes associated with actuators. There should be a status display process, a warning buzzer process and a brake system process.

15.9 To estimate the period of the process for trackside information collection, we must be able to ensure that the process will always be scheduled during the period can access the trackside information and collect information before it passes out of range of the transmitter. Here we use the maximum train speed (50 m/sec), the transmission time (50 ms) and the fact that the train must be within 10 m of the transmitter before it can collect information.

Simplistically, the train is within 10 m of a transmitter for 400 ms assuming that the distance between the train and the transmitter is negligible. It takes 50 ms to collect the information so

collection must start within 350 ms of entering a segment. Therefore, if the process collecting data from the transmitter is scheduled 3 times/second, it ought to always be able to collect transmitter data.

Chapter 16 User Interface Design

Solutions provided for Exercises 16.1, 16.2, 16.3, 16.5, 16.10, and 16.11.

16.1 (a) Warehouse assistant using a parts catalogue. Objects are parts, part numbers, stock, etc.

(b) Airline pilot using a safety monitoring system. Objects are aircraft sub-systems e.g. engine sub-system and their performance parameters.

(c) Manager manipulating a financial database. Objects are stocks, bonds, investments, interest rates, etc.

(d) Policeman using a patrol car system. Objects are other cars, locations, types of incident, etc.

16.2 A consistent user interface may be impossible to produce for complex systems with a large number of interface options. In such systems, there is a wide imbalance between the extent of usage of different commands so for frequently used commands, it is desirable to have short cuts. Unless all commands have short cuts, then consistency is impossible.

It may also be the case in complex systems that the entities manipulated are of quite different types and it is inappropriate to have consistent operations on each of these types.

An example of such a system is an operating system interface. Even MacOS which has attempted to be as consistent as possible has inconsistent operations that are liked by users. For example, to delete a file it is dragged to the trash but dragging a disk image to the trash does not delete it but unmounts that disk.

16.3 Factors to be taken into account when designing 'walk up and use' systems are:

- System users may be infirm, or disabled so will not be able to respond quickly to requests.
- Users may not be able to speak the native language of the country where the machine is installed.
- System users may be completely unfamiliar with technology and may make almost any kind of error in using the machine. The interface must minimise the number of possible errors and must be resilient to any possible error.
- Some system users are likely to be intimidated by many options. On the other hand, as users gain familiarity with the system, they may expect to use it for a wider range of banking services.
- Different people may understand the meaning of icons in different ways.
- If the system has navigation options, users are almost certain to become lost.
- Most users will want to use the system for very simple functions (e.g. withdraw cash from an ATM) and will want to do this as quickly as possible.

There are many different ATM interfaces so each must be considered separately. Problems which I have found are:

When is it possible to cancel a transaction? What happens when I do so? What will I have to re-input if I restart the transaction?

There is not usually any way of saying give me the maximum amount of money I may withdraw today.

Some machines only support single transactions - there is no way of saying I will be making several transactions and the same validation process is applicable to all of them.

16.5 Advantages are ‘at a glance’ magnitude indication and relative magnitude indication. Any applications where these are important might be mentioned:

- Temperature control

- Speed indicators
- Weather statistics
- Relative comparisons of cars, etc.

16.10 I assume that the object of this exercise is to discover problems rather than objectively compare different systems. The questionnaire should include questions which cover:

- The experience of the word processor user.
- The types of documents he/she produces. Are these all text, include text and graphics, use mathematical symbols, etc.
- Whether the user is familiar with and uses other word processors. What they like/dislike about them.
- The principal problems perceived by the word processor user.
- What facilities they used most. What they felt about the way in which the menus (if any) were arranged.
- Where they felt that they made mistakes when using the system.
- Whether they used the mouse or the keyboard to issue commands. If they used keyboard shortcuts, which did they use? Did they have problems with these?

16.11 In general, I do not think that it is ethical to monitor users without telling them that they are being monitored and without telling them the purpose of the monitoring. Depending on the users, it is arguable whether monitoring is ethical at all in that individuals have a write to privacy and this includes how they use systems. If the users are members of the general public then I believe that this should hold; if, however, the users are all employees of the same organisation then the situation is more difficult and monitoring may be permitted (see cases where employees who have downloaded pornography from the web have been sacked.

The argument for monitoring, of course, is that it allows behaviour to be tracked and the system improved for users. Without monitoring, any improvements are simply guesses. Furthermore, if users know of the monitoring they may change their behaviour so therefore secret monitoring is justified. It can also be argued that it is the system and not the users that is being monitored.

Chapter 17 Rapid software development

Solutions provided for Exercise 17.3, 17.4, 17.5 and 17.8.

17.3 Agile methods should not be used when the software is being developed by teams who are not co-located – if any of these teams use agile methods, it is very difficult to coordinate their work with other teams. Agile methods should probably also be avoided for critical systems where the consequences of a specification error are serious. In those circumstances, a system specification that is available before development starts makes a detailed specification analysis possible. However, some ideas from agile approaches such as test first development are certainly applicable to critical systems.

17.4 Advantages of stories:

They represent real situations that commonly arise so the system will support the most common user operations.

It is easy for users to understand and critique the stories.

They represent increments of functionality – implementing a story delivers some value to the user.

Disadvantages of stories

They are liable to be incomplete and their informal nature makes this incompleteness difficult to detect.

They focus on functional requirements rather than non-functional requirements. Representing cross-cutting system requirements such as performance and reliability is impossible when stories are used.

The relationship between the system architecture and the user stories is unclear so architectural design is difficult.

17.5 Test-first development helps with understanding the requirements because, in order to write a test, you have to analyse the requirements in detail to discover what is intended. In many cases, you may find that writing a test is impossible because the requirements are incomplete. The problem with test-first development is that some tests are very difficult to write because they require a system infrastructure to be in place before anything can be tested.

17.6 Reasons why pair programming is as efficient as the same number of programmers working individually:

Pair programming leads to continuous informal reviewing. This discovers bugs more quickly than individual testing.

Information sharing in pair programming is implicit – it happens during the process. Individual programmers have to spend time explicitly sharing information.

Pair programming encourages refactoring (the code must be understandable to another person). This reduces the costs of subsequent development and change.

In pair programming, people are likely to spend less time in fine-grain optimization as this does not benefit the other programmer.

17.8 *Throw-away prototyping*
- Fast development and rapid feedback from users.
- Likely to result in reasonable requirements.
- Needs multiple development languages.
- Costs more.

Develop using C and X-windows
- Fewer problems with training.

- Known management strategy.
- Requirements likely to be wrong so needs post-delivery modification.

Evolutionary prototyping

- Fast feedback from users.
- Rapid system delivery.
- Readily adapted to evolving requirements.
- Hard to manage.
- Lack of standards for portability etc.
- Likely to be unstructured causing future maintenance problems.

17.9 Issues that have to be taken into account in deciding which approach to use for prototyping in this case include:

1. The need for a very simple user interface that can be used by volunteers with little or no computer experience. This suggests that great care has to be taken with the design and that some form of GUI builder with rapid iteration of the interface design should be used.

2. The technology available – it probably has to run on PCs.

3. Whether or not remote access to the system has to be provided (this suggests that a web based interface should be provided)

4. The system, essentially, has to rely on a database with a number of fields. The prototyping approach has to consider the availability of a database system (on PCs, Microsoft Access is usually available). Whether or not it is desirable to have web pages generated from the database information should also be considered.

Chapter 18 Design with reuse

Solutions provided for Exercises 18.2, 18.3, 18.4, 18.7.

18.2 If savings from reuse were proportional to the amount of code reused, then reusing 2000 lines of code would save twice as much as reusing 1000 lines of code. However, to reuse 2000 lines of code, that code must be understood and the costs of program understanding are not linear – the more code to be understood, the more effort it takes to understand it. Furthermore, more changes may be required, the larger the amount of code reused so this also adds to the costs of reusing more code.

Of course, all this is only true if the code has to be understood before it is reused. If it can be reused without change, then savings from reusing large chunks of code tend to be proportionally greater than savings from reusing small code fragments.

18.3 Circumstances where software reuse is not recommended:

If the business status of the code provider is dubious. If the provider goes out of business, then no support for the reused code may be available.

In critical applications where source code is not available. Testing the code to the required standards may be very difficult.

In small systems where the costs of reuse are comparable to the savings that result if code is reused.

In systems where performance is a critical requirement – specially developed code can usually be made more efficient.

18.4 Patterns are an effective form of design reuse because the reflect accumulated wisdom that has been collected over several applications rather than a single application (By definition, a pattern is something that should appear in more than one application). There are two problems with patterns for reuse. The first is knowing which patterns actually have been documented and then finding these patterns - the time taken to do this can be significant. The second is that patterns are by their nature generalisations so their performance is likely to be limited. If performance is critical, then a special-purpose tailored approach to a problem will almost always be more effective.

18.7 Risks that can arise when systems are constructed using COTS include:

Vendor risks:	Failure of vendor to provide support when required
	Vendor goes out of business or drops product from its portfolio
Product risks:	Incompatible event/data model with other systems
	Inadequate performance when integrated with other systems
	Product is undependable in intended operating environment
Process risk:	Time required to understand how to integrate product is higher than expected.

The risks can be addressed by only dealing with vendors that use an escrow system so that source code is available if they go out of business, by extensive research and testing of product capabilities before use, discussion with other users etc. In general though, because COTS are provided by external vendors, risk reduction is difficult.

18.9 See Figure 18.1.

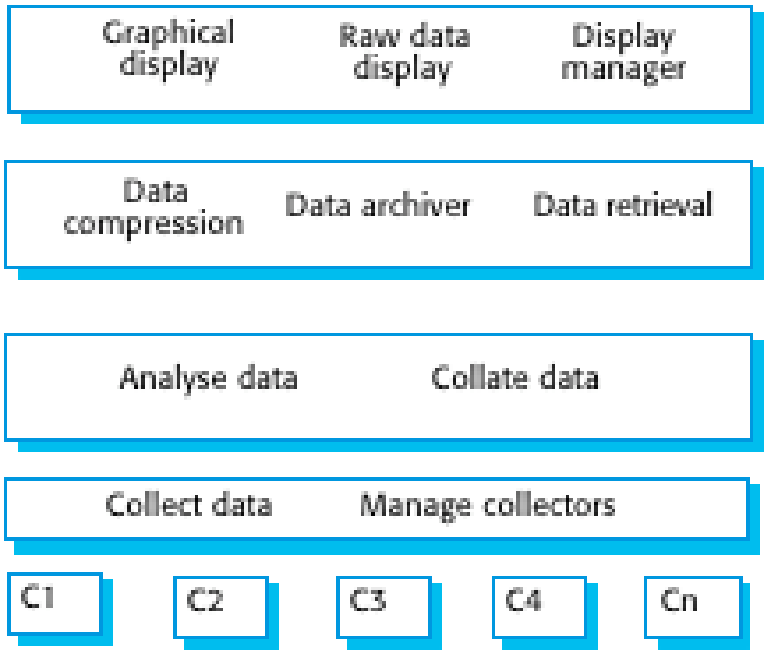

Figure 18.1 Product line architecture of a remote monitoring system

Chapter 19 Component-based software engineering

Solutions provided for Exercises 19.1, 19.3, 19.5, 19.6

19.1 It is important to define all interactions through requires and provides interfaces so that the use of the component is completely independent of its implementation. If component interactions use some knowledge of the components that is not defined in the requires/provides interfaces then the coupling between the components is increased and it is harder to interchange one component for an equivalent component with the same interfaces.

19.3 Differences between components and web services:

Once a component is purchased, it is owned by the user whereas the web service is always owned by the provider. This is significant because it means that the owner has no control over changes to the service – if it changes (or disappears) then this may have adverse consequences for the user. With components, however, the user decides when newer versions are to be used.

Payment for services is by utilization so that users don't have to buy an expensive component that is only occasionally used.

Component interactions can use much more efficient protocols than web services so components are better suited to high throughput/performance applications.

There is a single standard for services against several competing standards for components so service inter-operability is (should be) much better.

19.5 Take, for example, a stack component. This will provide basic operations that are common to all stacks such as Initialise (Create a stack), Push (an item onto the stack), Pop (an item from the stack), Size (the number of items currently on the stack), and perhaps others. However, each application will use stacks in different ways and so may require different versions of these operations and additional operations.

For example, consider a graphical browsing operation that allows users to browse a digital library. The library is divided into areas and the identifier for each area points to the books in that area. When the user enters an area, its identifier is pushed onto a stack and popped from the stack when he or she leaves that area and goes back to the previous area. Thus, the top of the stack always refers to the books in the current area. However, there is a requirement to provide a facility where the user can view all areas visited and this requires an additional stack operation that provides access to all stack elements. This then has to be added to the stack component. It also requires the Pop operation to be modified so that, when an item is popped from the stack it is added to a 'visited areas' list that can be displayed in conjunction with the current stack elements.

19.6 Component validation without source code is very difficult because there is no way of assessing how the component handles exceptions (and this is rarely defined in a component specification). The only validation method that can be used is black-box testing so static techniques cannot be used. Component specifications are rarely complete and this increases the problems of black-box testing. Formal specifications would help because they would precisely define what the component was supposed to do and its actual behaviour could be compared to the specification. However, formal specification rarely cover all exceptions and they do not help with testing performance, dependability or other non-functional characteristics.

19.7 Solution to be added.

19.8 In sequential composition, Component C is created by composing A and B in sequence i.e. A; B. In hierarchical composition, Component C is created from Component B calling component C. In additive composition, Component C is created by integrating the interfaces of component A and component B to create the interface of component C.

An example of sequential composition might be a component that

19.10 Certification authority – advantages are clearly the existence of some trusted 3rd party guarantee than the component is trustworthy but disadvantages are who would pay for such an authority and to what extent would be usage of the component be certified. **(more here)**

Chapter 20 Critical systems development

Solutions provided for Exercises 20.2, 20.3, 20.4, 20.10 and 20.11.

20.2 Examples of diverse, redundant activities are:

Program testing and program inspections.

Graphical state machine modeling and formal specification.

Tabular and natural language specification.

20.3 Inheritance is potentially error-prone because all of the information that is required to understand the operation of an object is not collected together in one place. (*1 mark)*. This makes it more difficult for a program reader to understand the object and hence more likely that errors will not be discovered (*1 mark)*. Another problem, if dynamic binding is used, is that the timing behaviour of the object is unpredictable. (*1 mark)*

20.4 The problems of developing and maintaining non-stop systems are:

- Providing facilities in the system which try to ensure that any faults in the system do not result in unrecoverable failures.
- Providing automatic restart facilities so that, if unrecoverable failures do occur, the system restarts automatically and quickly with minimal loss of data or services.
- If appropriate, providing facilities which, in the event of failure or data corruption, routes requests for service to other systems (important in telephone exchanges e.g.)
- Providing self-testing facilities which constantly check for corruption of system data or tables.
- Providing dynamic re-configuration so that system modules may be replaced without taking the system out of service.
- Ensuring that upgrades to different instances of the system are made at the same time so that all systems always offer the same facilities (this may be very difficult in practice)

An exception handler which includes code to recover from system faults may be associated with critical components. This is invoked through programming language mechanisms when an exceptional situation such as a numeric error or a data range error arises. The exception handler may take action to recover from the error and restart the component.

If fault recovery is not possible, the exception may be propagated to a higher level in the system and handled in a different way. Automatic system restrart facilities may be invoked or control may be switched to some backup system.

20.6 Solution to be provided.

20.10 Advantages of N-version programming

Increases design diversity so probability of faults that result in failures should be reduced

Increases availability of the system

Disadvantages

Increased cost because of the need to use independent development teams

Increased software complexity because of the need for a fault tolerant controller. Increased complexity increases the probability of error

Improvement in reliability in practice is limited because of the possibility of common errors made by different development teams.

N-version programming would not be a good design strategy for this type of software. There is no need for high availability and the increased complexity and cost would make the overall cost of the machine too high.

20.11 There may be a specification error which is reflected in both versions.

The problem may be a numeric error which has not been explicitly trapped.

The specification may be slightly ambiguous and interpreted wrongly by both teams.

Chapter 21 Software evolution

Solutions provided for Exercise 21.1, 21.4, 21.7, 21.8 and 21.9

21.1 Systems must change or become progressively less useful for a number of reasons:

The presence of the system changes the ways of working in its environment and this generated new requirements. If these are not satisfied, the usefulness of the system declines.

The business in which the system is used changes in response to market forces and this also generates new system requirements.

The external legal and political environment for the system changes and generates new requirements.

21.4 This is a very open question. Basically, the students should identify factors which affect maintainability such as (program and data complexity, use of meaningful identifiers, programming language, program documentation etc.). They should then suggest how these can be evaluated in existing systems whose maintenance cost is known and discuss problems of interaction. The approach should be to discover those program units which have particularly high maintenance costs and to evaluate the cost factors for these components and for other components. Then check for correlations.

Other factors may account for anomalies so these should be looked for in the problem components.

21.5 See Figure 21.1. (To be added).

21.7 Successful software re-engineering requires there to be staff available who are familiar with the technology used to develop the system and where automated tools to support re-engineering are available. Without these, it is unlikely that any software reengineering project can be cost-effective.

21.8 There is no hard and fast answer to this as obviously it depends on local circumstances. Examples of where software might be scrapped and rewritten are:

• When the cost of maintenance is high and the organisation has decided to invest in new hardware. This will involve significant conversion costs anyway so the opportunity might be taken to rewrite the software.

• When a business process is changed and new software is required to support the process.

• When support for the tools and language used to develop the software is unavailable. This is a particular problem with early 4GLs where, in many cases, the vendors are no longer in business.

21.9 The strategic options for legacy system evolution are:

a. Abandon maintenance of the system and replace it with a new system.

b. Continue maintaining the system as it is.

c. Perform some re-engineering (system improvement) that makes the system easier to maintain and continue maintenance.

d. Encapsulate the existing functionality of the system in a wrapper and add new functionality by writing new code which calls on the existing system as a component.

e. Decompose the system into separate units and wrap them as components. This is similar to the solution above but gives more flexibility in how the system is used.

You would normally choose the replacement option in situations where the hardware platform for the system is being replaced, where the company wishes to standardize on some approach to development that is not consistent with the current system, where some major sub-system is being replaced (e.g. a database system) or where the technical quality of the existing system is low and there are no current tools for re-engineering.

Chapter 22 Verification and Validation

Solutions provided for Exercises 22.1, 22.2, 22.4, 22.5, 22.7 and 22.8.

22.1 Verification is demonstrating conformance to a specification whereas validation is checking that a system meets the customer's needs. Validation is difficult because there are many different stakeholders who may use the system with different needs. therefore, a system that meets one user's needs may not meet the needs of a different user. Furthermore, needs change as a system is developed so the needs as identified when the system was specified may be different by the time that the system is tested.

22.2 A program need not be completely free of defects before delivery if:

1. Remaining defects are minor defects that do not cause system corruption and which are transient i.e. which can be cleared when new data is input.

2. Remaining defects are such that they are recoverable and a recovery function that causes minimum user disruption is available.

3. The benefits to the customer's business from the system exceed the problems that might be caused by remaining system defects.

Testing cannot completely validate that a system is fit for its intended purpose as this requires a detailed knowledge of what that purpose will be and exactly how the system will be used. As these details inevitably change between deciding to procure a system and deploying that system, the testing will be necessarily incomplete. In addition, it is practically impossible for all except trivial system to have a complete test set that covers all possible ways that the system is likely to be used.

22.4 Program inspections are effective for the following reasons:

1. They can find several faults in one pass without being concerned about interference between program faults.

2. They bring a number of people with different experience of different types of errors. Hence, the team approach offers greater coverage than any individual can bring.

3. They force the program author to re-examine the program in detail - this often reveals errors or misunderstandings.

The types of errors that inspections are unlikely to find are specification errors or errors that are based on a misunderstanding of the application domain (unless there are domain experts in the team).

22.5 An organisation with a competitive elitist culture is unlikely to find that program inspections are effective for the following reasons:

1. Program authors are unlikely to be open about their program because it exposes them to competition. The best programmers, in particular, are unlikely to want to be involved in inspections.

2. The inspection team, rather than being cooperative, are likely to try to compete with each other to find the most errors. While competition can sometimes be helpful, in this situation if someone is obviously falling behind they may then stop participating actively in the inspection.

3. Women often feel intimidated by competitive cultures and may therefore opt of the process.

22.7 The list in Figure 22.8 can also be specialised for each particular programming language e.g. storage management faults need to be checked for in C and C++ but not in Java.

Potential programming errors. To some extent these are language-independent but the extent of the compiler checking varies from one language to another.

a) Initialisation of a variable to the wrong value (the compiler can check for initialisation but not the right initialisation).

b) Inheritance from the wrong super-class (a problem if the inheritance is not from the root of the class hierarchy).

c) Use of incorrect constants (e.g. previous-value instead of current-value where previous-value and current-value are of the same type).

d) Use of incorrect conditions e.g. < rather than <=. This can't be checked by a compiler.

e) Visibility faults where names that should be declared as private are actually public.

22.8 Formal methods can be cost-effective in the development of safety-critical software systems because the costs of system failure are very high and so additional cost in the development process is justified. Most safety-critical systems have to gain regulatory approval before they are used and it is a very expensive process to convince a regulator that a system is safe. The use of a formal specification and associated correctness argument may be less than the costs e.g. of additional testing to convince the regulator of the safety of the system.

Some developers of systems are against the use of formal methods because they are unfamiliar with the technology and unconvinced that a formal specification can be complete representation of the system. Furthermore, the problem with formal specifications are that they cannot be understood by system customers so they may conceal errors and give a false picture of the correctness of the system.

Chapter 23 Software Testing

Solutions provided for Exercise 21.1, 23.3, 23.6, 23.7.

23.1 Assume that exhaustive testing of a program, where every possible valid input is checked, is impossible (true for all but trivial programs). Test cases either do not reveal a fault in the program or reveal a program fault. If they reveal a program fault then they demonstrate the presence of an error. If they do not reveal a fault, however, this simply means that they have executed a code sequence that – for the inputs chosen – is not faulty. The next test of the same code sequence – with different inputs – could reveal a fault.

23.3 Regression testing is the process of running tests for functionality that has already been implemented when new functionality is developed or the system is changed. Regression tests check that the system changes have not introduced problems into the previously implemented code. Automated tests and a testing framework radically simplify regression testing as the entire test set can be run automatically each time a change is made. The automated tests include their own checks that the test has been successful or otherwise so the costs of checking the success or otherwise of regression tests is low.

23.4 A possible scenario for high-level testing of the weather station system is:

John is a meteorologist responsible for producing weather maps for the state of Minnesota. These maps are produced from automatically collected data using a weather mapping system and they show different data about the weather in Minnesota. John selects the area for which the map is to be produced, the time period of the map and requests that the map should be generated. While the map is being created, John runs a weather station check that examines all remotely collected weather station data and looks for gaps in that data – this would imply a problem with the remote weather station.

23.5 The sequence diagram in Figure 8.14 is concerned with the issue of electronic items where a LIBSYS user looks up items in a catalogue then has them delivered to their computer. A licence for use is issued before the items can be delivered. Tests that might be developed are:

1. Lookup the catalogue for an item that you know is present.
2. Lookup the catalogue for an item that you know is not present.
3. Ask for a catalogue item to be issued. Accept the licence
4. Ask for a catalogue item to be issued. Refuse the licence.
5. Test that a compressed item has been downloaded.
6. Test that a compressed item has been properly decompressed by the computer receiving the item.

23.6 The major problems in developing performance tests for such a system are:

1. Because it is impossible to replicate the real use of the system in practice, you have to write a simulator that simulates its use under load. However, as you don't have experience of the load, then its hard to test the accuracy of the simulator.
2. The LIBSYS system will not necessarily run on dedicated computers – they may be running other applications at the same time. The performance of the overall distributed system is affected by these other applications. During testing, this situation cannot be replicated.
3. The distribution in data in practice is unknown. The data distribution affects the number of hits on a particular server so the initial assumptions about data may be inaccurate.

23.7 There are several reasons why interface testing is a necessary stage after unit testing:

- The interface to the module may have been incorrectly specified. The validation process is based on this specification rather than actual usage of the module or sub-system.
- The assumptions made by other modules about the behaviour of a given module (A say) in response to particular interface stimuli may be incorrect. That is, these modules expect A to behave in a way in which it was never designed to operate.
- Interface testing can reveal omissions in the interface design. It may be discovered, when integrated with other modules, that the interface must be augmented in some way.

Chapter 24 Critical Systems Validation

Solutions provided for Exercises 24.1, 24.2, 24.5, 24.6, 24.8 and 24.10.

24.1 You need to validate these specifications by implementing a transaction simulator which delivers signals corresponding to those which would be generated with a bar code reader. This can operate much more quickly than a bar code reader, therefore can deliver a very high volume of transactions in a relatively short time.

You need to run multiple instances of this simulator to simulate a network of systems. You should also write a program to capture actual transaction information from existing bar code readers (if possible) so that this may be used as a valid operational profile for the software.

24.2 To measure reliability you need to have statistically valid failure data for the system so you need to induce more failures than are specified in the given time period. However, because the number of failures is so low, this will take an unrealistically large amount of time.

24.5 Reliability is a measure of *how well a system meets its specification*. That is, it measures how well the system does what it is supposed to do. Safety is often concerned with what the *system must not do* and this may be impossible to express completely in a system specification. Furthermore, *specifications may be in error* and these errors may result in safety-related failures.

24.6 There are two potential safety problems with this code:

- Say the door was unlocked when the door entry code was entered. Line 13 checks if the state is safe and, if it is safe then unlocks the door. However, if the door was unlocked to begin with, there is no locking action if the state is unsafe so therefore a potential safety loop hole exists.

- If the radiation level is less than the danger level then line 8 sets the state to be safe. However, line 10 checks the shields to see if they are in place. If they are not in place, the state is unchanged although, in fact, the system is unsafe if the shields are down. Therefore, the door can be opened with the shields down and a safety loophole exists.

There are two changes which should be made to ensure that the code is safe:

- An initial statement which locks the door and sets door locked to be true.

- The if statement if shield-status == Shield.inPlace then should be changed to

```
if (shield_status == Shield.inPlace())
  state := safe;
else
  state := unsafe;
```

There are other ways to do this with nested if statements.

24.10 Safety cases would normally be required for any system that needs to be certified by a regulator before it is used e.g.:

Systems used to control equipment in the nuclear industry where there is a possibility of the release of radioactivity.

Air traffic control software

Signalling and control systems in the railway industry.

Software for critical aircraft functions such as flight control systems.

24.8 Validating a password protection system

1. Identify possible threats. The principal threats are

a. Attacker gains access without a password

b. Attacker guesses a password of an authorised user

c. Attacker uses a password cracking tool to discover passwords of authorised users

d. Users make passwords available to attackers

e. Attacker gains access to an unencrypted password file

2. Develop tests that cover each of these threats

a. Test system for all authorised used to check that they have set a password.

b. Test system heuristically for commonly used passwords such as names of users, festivals, other proper names, strings such as '12345' etc.

c. Check that all user passwords are not words that are in a dictionary. A password cracking tool usually checks encrypted passwords against the same encryptions of words in a dictionary.

d. This is very hard to check. To stop users writing down passwords you need to allow words that are in the dictionary and are hence easy to remember.

e. Check that access to the password file is very limited. Check that all copy actions on the password file are logged.

24.9 Solution to be added.

24.11 Solution to be added.

Chapter 25 Managing People

Solutions provided for Exercise 25.2,25.3, 25.7 25.8 and 25.10.

25.2 Key factors are experience with the domain, platform and tools, personality of the individuals, in particular their ability to fit into an existing team and their communication skills. For an eye surgery machine, the system is safety critical so domain and platform experience is particularly important as inexperienced staff are liable to miss critical features that could affect the safety of the system. Teamworking ability and the ability to respond constructively to criticism is also important as reviews are an essential part of critical systems development.

25.3 Examples of general activities that Alice might introduce to improve team motivation are:

1. Regular appraisals where she discusses personal goals with team members and how the company can help satisfy these goals. Even in situations where this is impossible, the fact that the goals have been discussed is itself motivating.

2. Opportunities for team members to attend conferences to help keep them in touch with the latest developments in their field of work.

3. A dedicated training budget that can support people attending courses to develop new skills.

4. 'Brown-bag' seminars where talks are held over lunch on technical topics – these can be from team members or from outside speakers.

25.6 While the notion of devolving management decisions to the team is attractive in terms of motivation, there are two types of problem that can arise:

1. Decisions are liable to be primarily influenced by technical considerations rather than business decisions. This is natural given the type of people on an XP team – it is difficult for them to take a business perspective.

2. Because of the focus on rapid iteration, management decisions tend to be short-term and pay insufficient attention to long-term issues. While this is in keeping with the XP philosophy, there is sometimes a need for a more detached, longer-term perspective which can be taken by a manager.

I assume here that management decisions on e.g. the performance of team members are not taken by the team. Given the close knit nature of XP teams, it is difficult for the team to take decisions that censure individual team members.

25.7 Open-plan offices are noisy and people are easily distracted. This is a problem when individuals need time to concentrate. They also inhibit informal discussions because the people involved feel they should not be disturbing others. Circumstances where such environments might be better is where there is a need for team members to have awareness of what other people are doing. They can overhear discussions and conversations about the work and so develop a better awareness of the current state of the work and future plans.

25.8 The P-CMM is an effective framework because it summarises good practice in the management of people and provides a basis for process improvement. As new processes are introduced, the level of maturity can be computed using the framework and, over time, this maturity level should increase.

Small organisations, however, would not normally be expected to have i place all of the formal people management procedures that are used in large organisations. To use the P-CMM, you should examine all of the practices are rate them as 'essential as defined', 'essential but may be redefined', 'desirable', 'unnecessary'. This should simplify the approach especially

if the organisation concentrates on putting in place those practices which are deemed to be essential.

25.10 The issue here is that the links between these tests and work performance is unproven (and perhaps unprovable) so it could be argued that organizations that use these tests are not themselves behaving in an ethical way as they are discriminating against people who, for reasons of their background, do not perform as well in such tests. Given this situation, it can certainly be argued that someone being tested may behave in any legal way in order to present the picture of themselves that the organization wants. However, what you should not ever do is present factually incorrect information.

Chapter 26 Software Cost Estimation

Solutions provided for Exercises 26.1, 26.2, 26.4, 26.6, 26.7 and 26.8.

26.1 Circumstances where a high price might be charged:

(a) Where a customer expects the developer to take on a considerable amount of project risk.

(b) Where the customer has special requirements e.g. for very rapid delivery.

(c) When the work is not central to the companies business and so diverts people from other more business-focused activities. The high price is intended to compensate for this.

(d) When the customer has no alternative! Think about the ethics of excessive pricing in this situation.

26.2 Metrics that have been used for productivity measurement are:

- Lines of source code produced per unit time
- Object code instructions per unit time
- Pages of documentation per unit time

Other possibilities are:

- Number of data dictionary entries made per unit time (may be useful if CASE tools are used)
- Number of mathematical definitions produced per unit time (formal specification)
- Number of requirements written per unit time
- Number of design diagrams produced per unit time

All of these, of course, suffer from the same problem as other metrics, that is, they don't take quality into account.

26.4 Possible techniques of risk reduction include:

- Obtain a number of independent estimates using different estimation techniques. If these are widely divergent, generate more costing information iterate until the estimates converge.
- For those parts of the system which are hard to estimate, develop a prototype to find out what problems are likely to arise.
- Reuse software to reduce the amount of estimation required and to reduce overall costs.
- Adopt a design to cost approach to development where the system functionality is adapted to a fixed cost.
- Partition software requirements into critical, desirable and 'gold plating'. Eliminate 'gold-plating' if necessary.

26.6 It is adjusted because the time and effort required to complete a project depends on various factors such as the experience of the development team, the development schedule, the support facilities etc.

In this case, they have to go back to the system description and recognise the factors which might add to the difficulty (and hence the cost) of implementing the system. For example:

- Safety-criticality
- Memory limitations on system
- External interfaces with DBMS
- Unusual hardware ('special-purpose processor)

26.7
- Different languages and development tools
- Different ways of counting lines of code

- Subjective complexity estimates to adjust results
- Different historical cost databases (e.g. different activities costed against projects)

26.8
- Produce worst, best and most likely cost estimates using simple model.
- Identify process and product variables such as team experience, possibility of hardware upgrade, possibility of tool purchase, etc.
- Build a spreadsheet model allowing the effects of these variables on the cost estimates to be computed.

When there are organisational reasons, some approach which is relatively more expensive may be chosen. For example, software engineers whose experience is not ideal may be available so they may be used rather than recruit new staff.

Chapter 27 Quality Management

Solutions provided for Exercises 27.4, 27.5, 27.6, 27.7 and 27.9.

27.2 Standards encapsulate organizational wisdom because they capture good practice that have evolved over the years. Knowledge that might be captured in organizational standards include:

1. Knowledge of specific types of fault that commonly occur in the type of software developed by an organization. This might be encapsulated in a standard review checklist.

2. Knowledge of the types of system model that have proved useful for software maintenance. This can be encapsulated in design documentation standards.

3. Knowledge of tool support that has been useful for a range of projects. This can be encapsulated in a standard for a development environment to be used by all projects.

4. Knowledge of the type of information that is useful to include as comments in code. This can be encapsulated in a code commenting standard.

27.4 The fields in the review form might include:

1. Name of person raising review comment
2. Date comment raised
3. Contact phone number or e-mail address
4. The review comment itself
5. Name of comment assessor
6. Date of comment assessment
7. Action taken from comment (Return for clarification, invalid comment, accepted comment)
8. System change proposed
9. Person responsible for change
10. Date change made
11. Date change checked

27.5b Obviously there is no right and wrong answer to this question. As term projects cover a wide range of topics, the standard should focus on the report organisation. Issues which should be addressed include:

Document structure

The document must include the following sections:

- Title page identifying the project and its author
- Introduction describing the problem being solved, the general approach adopted in the solution and problems encountered during the development of the solution
- Chapters giving a detailed description of the approach used.
- A concluding chapter which critically assesses the solution
- Where appropriate, appendices listing the source code of the solution and user documentation.

Title page

The title page must include the following information

- Project title
- Course identifier
- Author name(s)
- Date of submission
- Name of instructor

Page organisation

Each page must include a header giving the title of the project, centred on the page and a footer including the page number and the version of the document submitted. Other information may be included in the header and footer if appropriate.

27.6 Again, no right or wrong answer. The organisation is interested in quantifying its software development so may collect metrics about its products and about its processes. The type of software which is developed is important as the metrics should take into account its characteristics. In this case, the company is developing database products for microcomputers so:

- As they are shrink-wrapped products, they will run on many different system configurations. Configuration dependent problems may occur. It is important that the system should not hang the machine on which it is running.
- As they are database products, it is important that the system does not corrupt the database

Product metrics

Product metrics should be used to judge the quality and efficiency of the software.
Total number of measured faults detected by testing
Total number of faults which resulted in database corruption
Total number of system failures which forced a system restart
Number of database transactions processed per unit time.
Time to read/write large DB records

Process metrics

Number of different configurations used for system testing
Number of fault reports submitted
Average time required to clear fault after it is reported
Time required to run system regression tests

27.7 Because quality metrics assume that quality is only related to what can actually be measured (such as coupling). In fact, it is very difficult to say what quality really means and it is certainly related to many different program attributes (see Exercise 28.3 simply for those attributes which are related to maintainability). The importance of these attributes varies from system to system and from organisation to organization.

27.9 The basic difficulty arises because the external attributes such as maintainability are not just dependent on a small number of internal products attributes. While the complexity of a system influences its maintainability, other issues such as the use of variable names, the system documentation and, particularly, the skills of the people doing the maintenance have such a large effect on the process that they may mask any maintainability differences arising from different levels of complexity. This does not contradict experiments where a relationship between maintainability and complexity was discovered – however, we don't have enough evidence at the moment to generalize this.

27.8 Chapter 28 Process Improvement

Solutions provided for Exercises 28.1, 28.3, 28.4, 28.6, 28.8 and 28.9.

28.1 I have simply described the processes here as a sequence of activities. Obviously, there is scope for alternative descriptions such as parallelism etc. The activities can also, of course, be decomposed

Lighting a wood fire

Assemble a pile of dry wood
Collect easily combustible material (tinder)
Arrange some or all of the dry wood around the tinder
Strike match
Light tinder
Provide oxygen to fire and shield from drafts

Cooking a three course meal (simplified)

Decide on menu for meal
Check store cupboard for items in stock
Prepare list of groceries required
Buy groceries
Prepare food to be cooked
Cook food
Serve food

Writing a small program

Read and understand program specification
Decide on data types and structures which are required
Decide on processing algorithm required
Prepare a rough program design
Code program
Review program for errors
Compile program. Repeat until syntax errors removed
Prepare test data
Test program

28.3 A methodical process is a process that is based on the application of some defined method such as an object-oriented or function-oriented method. A managed process is a process that has a defined process model and process checks and documentation to ensure that the process model is being followed.

While it is sometimes (not always) the case that methods have an associated process model, these are often very weak and poorly defined and users of the methods rarely follow them exactly. Furthermore, there may not be checks in place to ensure that the method process model is followed. Therefore, although a methodical approach is being used, this is not necessarily a managed process.

28.4 Tools to capture process data from management information.
Process model editing tools
Tools to manage a process training program
Process visualisation tools which present different process views

28.6 Elapsed time. How long it takes to do something. Many possible examples e.g. time taken to carry out design review.

Resource utilisation. The amount of resources used. E.g. the effort required to test a module.

Events which occur. E.g. The number of defects discovered after a system has been delivered.

28.8 Advantages of process improvement frameworks

Provides a means of measuring the state of a process and a structured approach to introducing process improvements.

Useful as a way of building on the experience of others in process improvement.

Disadvantages of process improvement frameworks

Like any measurement system, there is a tendency to introduce improvements to improve the measured rating rather than concentrate on improvements that meet real business goals.

Expensive and bureaucratic to operate. Not suitable for agile approaches.

28.9 The staged version of the CMMI could be used when an organization has experience of and has been assessed using the earlier, staged Capability Maturity Model. The staged CMMI is compatible with this and its use would be a means of continuing improvement according the the CMM approach. You might also use it in organizations that are very immature and, for the earliest activities at least, would like a package of improvements to introduce.

Chapter 29 Configuration Management

Solutions provided for Exercises 29.1, 29.2, 29.3, 29.6, 29.7, 29.9 and 29.10.

29.1 The title should not be used as it is not a unique identifier (several documents from different sub-projects could have the same title). A possible numbering scheme could have the form

<project>: <subproject>: <task>: <doc type>: <document number>: <version>: <date>

29.2 There are many possible answers to this question. What you should look for is basically an understanding of the relationships between change requests, versions and components plus sensible attributes. See Figure 29.1 for one possible schema.

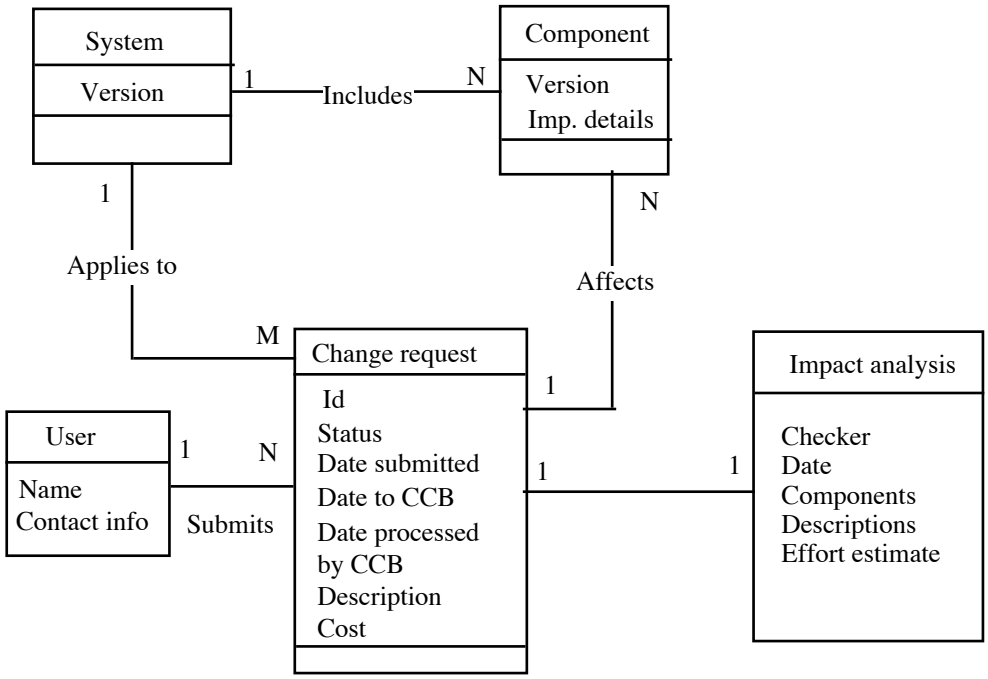

Figure 29.1 Schema for change management

29.3 See Figure 29.2.

29.6 Have all components been included?
Is the right version of all components been included?
Are all configuration/data files included?
Is the right version of the system building tools used?
Are there any problems with full path name references?

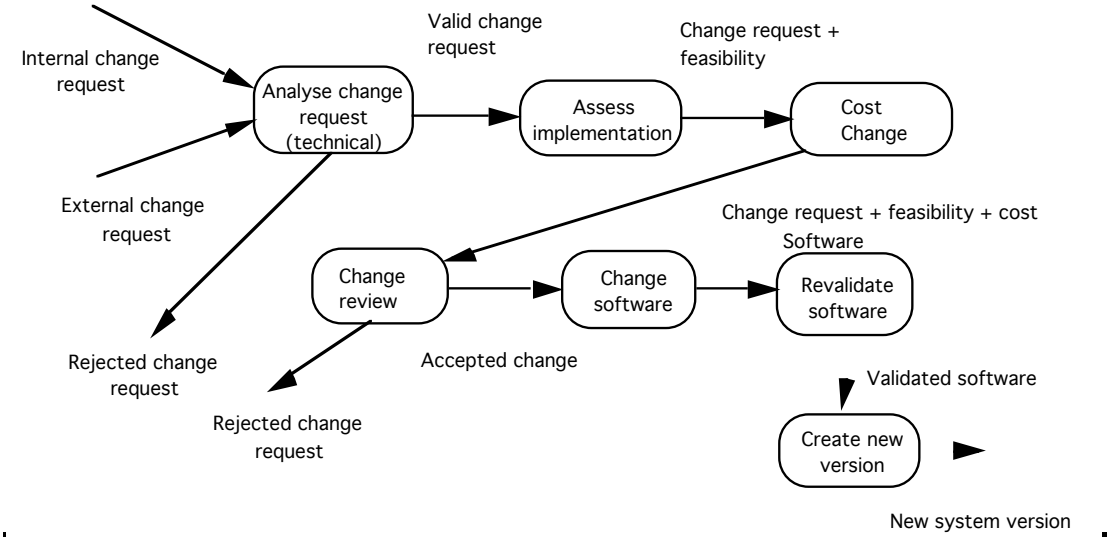

Figure 29.2 Change control

29.7 Obsolete computers may have to be maintained if the software used to build the system (compiler, linker, etc.) is not available for newer hardware. This situation can arise of the compiler vendor has gone out of business and no-one else is supporting their system. It may not be possible to use a compiler on a newer system as the code produced may be different. It can also arise when programs are developed in a programming language that is no longer used – it may be cheaper to maintain the obsolete hardware to run the compiler than to buy a new compiler for occasional use.

29.9 (a) Have all components been included in the build instructions;

(b) Has the right version of each component been specified;

(c) Are all data files available;

(d) Are all data files properly referenced;

(e) Are the correct versions of the compiler or other tools available and specified.

29.10 Two ways of optimising the system building process are:

• Reuse of derived elements. If an element has already been compiled, use the compiled version rather than re-compiling.

•Parallel building. Different components of the system can be built on different nodes of the network.

Derived elements may be reused by maintaining a derived element pool which associates descriptors with derived elements. These descriptors uniquely identify the source element used to create the derived element so different versions of the derived element can be maintained in the derived element pool.

Parallel building is supported by maintaining a list of possible build platforms and, when a build is initiated, choosing a machine from this list. The files to be processed are down-loaded to this machine before processing.