Chapter 1

Software Test and Analysis in a Nutshell

Before considering individual aspects and techniques of software analysis and testing, it is useful to view the "big picture" of software quality in the context of a software development project and organization. The objective of this chapter is to introduce the range of software verification and validation activities and a rationale for selecting and combining them within a software development process. This overview is necessarily cursory and incomplete, with many details deferred to subsequent chapters.

1.1 Engineering Processes and Verification

Engineering disciplines pair design and construction activities with activities that check intermediate and final products so that defects can be identified and removed. Software engineering is no exception: Construction of high quality software requires complementary pairing of design and verification activities throughout development.

Verification and design activities take various forms ranging from those suited to highly repetitive construction of non-critical items for mass markets to highly customized or highly critical products. Appropriate verification activities depend on the engineering discipline, the construction process, the final product, and quality requirements.

Repetition and high levels of automation in production lines reduce the need for verification of individual products. For example, only a few key components of products like screens, circuit boards, and toasters are verified individually. The final products are tested statistically. Full test of each individual product may not be economical, depending on the costs of test, the reliability of the production process, and the costs of field failures.

Even for some mass market products, complex processes or stringent quality requirements may require both sophisticated design and advanced product verification procedures. For example, computers, cars, and aircraft, despite being produced in series, are checked individually before release to customers. Other products are not built in series, but are engineered individually through highly evolved processes and tools. Custom houses, race cars, and software are not built in series. Rather, each house, each racing car, and each software package is at least partly unique in its design and functionality. Such products are verified individually both during and after production to identify and eliminate faults.

Verification of goods produced in series, e.g., screens, boards, or toasters, consists in repeating a predefined set of tests and analyses that indicate whether the products meet the required quality standards. In contrast, verification of a unique product, such as a house, requires the design of a specialized set of tests and analyses to assess the quality of that product. Moreover, the relationship between the test and analysis results and the quality of the product cannot be defined once for all items, but must be assessed for each product. For example, the set of resistance tests for assessing the quality of a floor must be customized for each floor, and the resulting quality depends on the construction methods and the structure of the building.

The difficulty of verification grows with the complexity and variety of the products. Small houses built with comparable technologies in analogous environments can be verified with standardized procedures. The tests are parameterized to the particular house, but are nonetheless routine. Verification of a skyscraper or of a house built in an extreme seismic area, on the other hand, may not be easily generalized, instead requiring specialized tests and analyses designed particularly for the case at hand.

Software is among the most variable and complex of artifacts engineered on a regular basis. Quality requirements of software used in one environment may be quite different and incompatible with quality requirements of a different environment or application domain, and its structure evolves and often deteriorates as the software system grows. Moreover, the inherent non-linearity of software systems and uneven distribution of faults complicates verification. If an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load, but if a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 or 12 elements, as well as on 257 or 1023.

The cost of software verification often exceeds half the overall cost of software development and maintenance. Advanced development technologies and powerful supporting tools can reduce the frequency of some classes of errors, but we are far from eliminating errors and producing fault-free software. In many cases new development approaches introduce new subtle kinds of faults, which may be more difficult to reveal and remove than classic faults. This is the case, for example, with distributed software, which can present problems of deadlock or race conditions that are not present in sequential programs. Likewise, object-oriented development introduces new problems due to the use of polymorphism, dynamic binding and private state that are absent or less pronounced in procedural software.

The variety of problems and the richness of approaches make it challenging to choose and schedule the right blend of techniques to reach the required level of quality within cost constraints. There are no fixed recipes for attacking the problem of verifying a software product. Even the most experienced specialists do not have pre-cooked solutions, but need to design a solution that suits the problem, the requirements, and the development environment.

1.2 Basic Questions

To start understanding how to attack the problem of verifying software, let us consider a hypothetical case. The Board of Governors of Chipmunk Computers, an (imaginary) computer manufacturer, decides to add new on-line shopping functions to the company web presence to allow customers to purchase individually configured products. Let us assume the role of quality manager. To begin, we need to answer a few basic questions:

- When do verification and validation start? When are they complete?
- What particular techniques should be applied during development of the product to obtain acceptable quality at an acceptable cost?
- How can we assess the readiness of a product for release?
- How can we control the quality of successive releases?
- How can the development process itself be improved over the course of the current and future projects to improve products and make verification more cost-effective?

1.3 When Do Verification and Validation Start and End?

Although some primitive software development processes concentrate test and analysis at the end of the development cycle, and the job title "tester" in some organizations still refers to a person who merely executes test cases on a complete product, today it is widely understood that execution of tests is a small part of the verification and validation process required to assess and maintain the quality of a software product.

Verification and validation start as soon as we decide to build a software product, or even before. In the case of Chipmunk Computers, when the Board of Governors asks the IT manager for a feasibility study, the IT manager considers not only functionality and development costs, but also the required qualities and their impact on the overall cost.

The Chipmunk software quality manager participates with other key designers in the feasibility study, focusing in particular on risk analysis and the measures needed to assess and control quality at each stage of development. The team assesses the impact of new features and new quality requirements on the full system and considers the contribution of quality control activities to development cost and schedule. For example, migrating sales functions into the Chipmunk web site will increase the criticality of system availability and introduce new security issues. A feasibility study that ignored quality could lead to major unanticipated costs and delays and very possibly to project failure.

The feasibility study necessarily involves some tentative architectural design, e.g., a division of software structure corresponding to a division of responsibility between a human interface design team and groups responsible for core business software ("business logic") and supporting infrastructure, and a rough build plan breaking the project into a series of incremental deliveries. Opportunities and obstacles for cost-effective verification are important considerations in factoring the development effort into subsystems and phases, and in defining major interfaces.

Overall architectural design divides work and separates qualities that can be verified independently in the different subsystems, thus easing the work of the testing team as well as other developers. For example, the Chipmunk design team divides the system into a presentation layer, back-end logic, and infrastructure. Development of the three subsystems is assigned to three different teams with specialized experience, each of which must meet appropriate quality constraints. The quality manager steers the early design toward a separation of concerns that will facilitate test and analysis.

In the Chipmunk web presence, a clean interface between the presentation layer and back end logic allows a corresponding division between usability testing (which is the responsibility of the human interface group, rather than the quality group) and verification of correct functioning. A clear separation of infrastructure from business logic serves a similar purpose. Responsibility for a small kernel of critical functions is allocated to specialists on the infrastructure team, leaving effectively checkable rules for consistent use of those functions throughout other parts of the system.

Taking into account quality constraints during early breakdown into subsystems allows for a better allocation of quality requirements and facilitates both detailed design and testing. However, many properties cannot be guaranteed by one subsystem alone. The initial breakdown of properties given in the feasibility study will be detailed during later design and may result in "cross quality requirements" among subsystems. For example, to guarantee a given security level, the infrastructure design team may require the verification of absence of some specific security holes, e.g., buffer overflow, in other parts of the system.

The initial build plan also includes some preliminary decisions about test and analysis techniques to be used in development. For example, the preliminary prototype of Chipmunk online sales functionality will not undergo complete acceptance testing, but will be used to validate the requirements analysis and some design decisions. Acceptance testing of the first release will be primarily based on feedback from selected retail stores, but will also include complete checks to verify absence of common security holes. The second release will include full acceptance test and reliability measures.

If the feasibility study leads to a project commitment, verification and validation activities will commence with other development activities, and like development itself will continue long past initial delivery of a product. Chipmunk's new web-based functions will be delivered in a series of phases, with requirements reassessed and modified after each phase, so it is essential that the V&V plan be cost-effective over a series of deliveries whose outcome cannot be fully known in advance. Even when the project is "complete," the software will continue to evolve and adapt to new conditions, such as a new version of the underlying data base, or new requirements, such as the opening of a European sales division of Chipmunk. V&V continues through each small or large change to the system.

Why Combine Techniques?

No single test or analysis technique can serve all purposes. The primary reasons for combining techniques, rather than choosing a single "best" technique, are

- Effectiveness for different classes of faults. For example, race conditions are very difficult to find with conventional testing, but they can be detected with static analysis techniques.
- Applicability at different points in a project. For example, we can apply inspection techniques very early to requirements and design representations that are not suited to more automated analyses.
- Differences in purpose. For example, systematic (non-random) testing is aimed at maximizing fault detection, but cannot be used to measure reliability; for that, statistical testing is required.
- Trade-offs in cost and assurance. For example, one may use a relatively expensive technique to establish a few key properties of core components (e.g., a security kernel) when those techniques would be too expensive for use throughout a project.

1.4 What Techniques Should be Applied?

The feasibility study is the first step of a complex development process that should lead to delivery of a satisfactory product through design, verification and validation activities. Verification activities steer the process towards the construction of a product that satisfies the requirements by checking the quality of intermediate artifacts as well as the ultimate product. Validation activities check the correspondence of the intermediate artifacts and the final product to users' expectations.

The choice of the set of test and analysis techniques depends on quality, cost, scheduling and resource constraints in development of a particular product. For the business logic subsystem, the quality team plans to use a preliminary prototype for validating requirements specifications. They plan to use automatic tools for simple structural checks of the architecture and design specifications. They will train staff for design and code inspections, which will be based on company checklists that identify deviations from design rules for ensuring maintainability, scalability, and correspondence between design and code.

Requirements specifications at Chipmunk are written in a structured, semi-formal format. They are not amenable to automated checking, but like any other software artifact they can be inspected by developers. The Chipmunk organization has compiled a check-list based on their rules for structuring specification documents and on experience with problems in requirements from past systems. For example, the check-list for inspecting requirements specifications at Chipmunk asks inspectors to confirm that each specified property is stated in a form that can be effectively tested.

The analysis and test plan requires inspection of requirements specifications, design specifications, source code, and test documentation. Most source code and test documentation inspections are a simple matter of soliciting an offline review by one other developer, though a handful of critical components are designated for an additional review and comparison of notes. Component interface specifications are inspected by small groups that include a representative of the "provider" and "consumer" sides of the interface, again mostly offline with exchange of notes through a discussion service. A larger group and more involved process, including a moderated inspection meeting with three or four participants, is used for inspection of a requirements specification.

Chipmunk developers produce functional unit tests with each development work assignment, as well as test oracles and any other scaffolding required for test execution. Test scaffolding is additional code needed to execute a unit or a subsystem in isolation. Test oracles check the results of executing the code and signal discrepancies between actual and expected outputs.

Test cases at Chipmunk are based primarily on interface specifications, but the extent to which unit tests exercise the control structure of programs is also measured. If less than 90% of all statements are executed by the functional tests, this is taken as an indication that either the interface specifications are incomplete (if the missing coverage corresponds to visible differences in behavior), or else additional implementation complexity hides behind the interface. Either way, additional test cases are devised based on a more complete description of unit behavior.

Integration and system tests are generated by the quality team, working from a catalog of patterns and corresponding tests. The behavior of some subsystems or components is modeled as finite-state machines, so the quality team creates test suites that exercise program paths corresponding to each state transition in the models.

Scaffolding and oracles for integration testing are part of the overall system architecture. Oracles for individual components and units are designed and implemented by programmers using tools for annotating code with conditions and invariants. The Chipmunk developers use a home-grown test organizer tool to bind scaffolding to code, schedule test runs, track faults, and organize and update regression test suites.

The quality plan includes analysis and test activities for several properties distinct from functional correctness, including performance, usability, and security. Although these are an integral part of the quality plan, their design and execution is delegated in part or whole to experts who may reside elsewhere in the organization. For example, Chipmunk maintains a small team of human factors experts in its software division. The human factors team will produce look-and-feel guidelines for the web purchasing system, which together with a larger body of Chipmunk interface design rules can be checked during inspection and test. The human factors team also produces and executes a usability testing plan.

Parts of the portfolio of verification and validation activities selected by Chipmunk are illustrated in Figure 1.1. The quality of the final product and the costs of the quality assurance activities depend on the choice of the techniques to accomplish each activity. Most important is to construct a coherent plan that can be monitored. In addition to monitoring schedule progress against the plan, Chipmunk records faults found during each activity, using this as an indicator of potential trouble spots. For example, if the number of faults found in a component during design inspections is high, additional dynamic test time will be planned for that component.

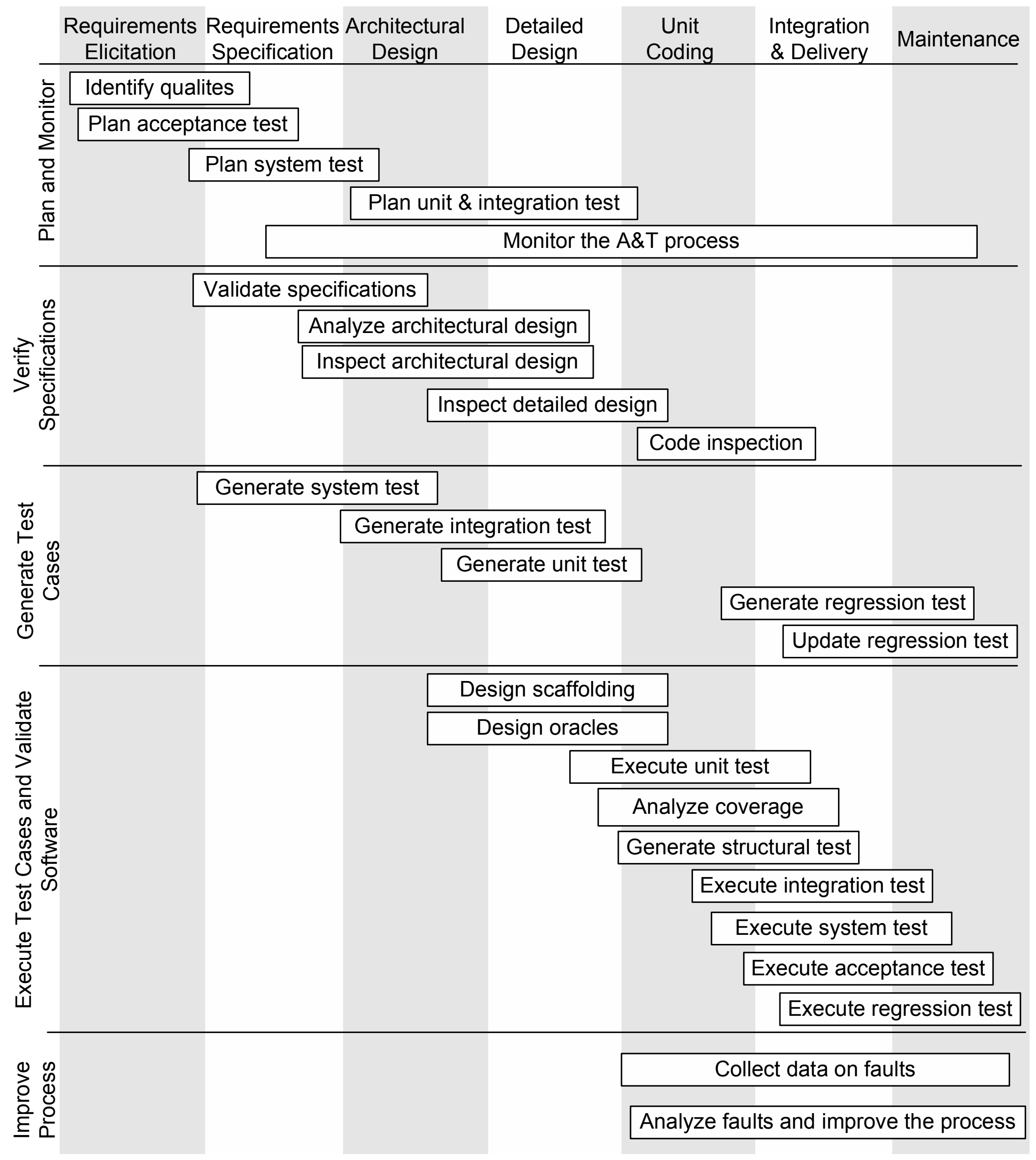

Figure 1.1: Main analysis and testing activities through the software life cycle.

1.5 How Can We Assess the Readiness of a Product?

Analysis and testing activities during development are intended primarily to reveal faults so that they can be removed. Identifying and removing as many faults as possible is a useful objective during development, but finding all faults is nearly impossible and seldom a cost-effective objective for a non-trivial software product. Analysis and test cannot go on forever: Products must be delivered when they meet an adequate level of functionality and quality. We must have some way to specify the required level of dependability, and to determine when that level has been attained.

Different measures of dependability are appropriate in different contexts. *Availability* measures the quality of service in terms of running versus down time; *mean time between failures (MTBF)* measures the quality of the service in terms of time between failures, i.e., length of time intervals during which the service is available. *Reliability* is sometimes used synonymously with availability or MTBF, but usually indicates the fraction of all attempted operations (program runs, or interactions, or sessions) that complete successfully.

Both availability and reliability are important for the Chipmunk web presence. The availability goal is set (somewhat arbitrarily) at an average of no more than 30 minutes of down time per month. Since 30 one minute failures in the course of a day would be much worse than a single 30 minute failure, mean time between failures is separately specified as at least one week. In addition, a reliability goal of less than 1 failure per 1000 user sessions is set, with a further stipulation that certain critical failures (e.g., loss of data) must be vanishingly rare.

Having set these goals, how can Chipmunk determine when it has met them? Monitoring systematic debug testing can provide a hint, but no more than a hint. A product with only a single fault could be have a reliability of zero if that fault results in a failure on every execution, and there is no reason to suppose that a test suite designed for finding faults is at all representative of actual usage and failure rate.

From the experience of many previous projects, Chipmunk has empirically determined that in their organization, it is fruitful to begin measuring reliability when debug testing is yielding less than one fault ("bug") per day of tester time. For some application domains, Chipmunk has gathered a large amount of historical usage data from which to define an operational profile, and these profiles can be used to generate large, statistically valid sets of randomly generated tests. If the sample thus tested is a valid model of actual executions, then projecting actual reliability from the failure rate of test cases is elementary. Unfortunately, in many cases such an operational profile is not available.

Chipmunk has an idea of how the web sales facility will be used, but it cannot construct and validate a model with sufficient detail to obtain reliability estimates from a randomly generated test suite. They decide, therefore, to use the second major approach to verifying reliability, using a sample of real users. This is commonly known as *alpha testing* if the tests are performed by users in a controlled environment, observed by the development organization. If the tests consist of real users in their own environment, performing actual tasks without interference or close monitoring, it is known as *beta testing.* The Chipmunk team plans a very small alpha test, followed by a longer beta test period in which the software is made available only in retail outlets. To accelerate reliability measurement after subsequent revisions of the system, the beta test version will be extensively instrumented, capturing many properties of a usage profile.

1.6 How Can We Assure the Quality of Successive Releases?

Software test and analysis does not stop at the first release. Software products often operate for many years, frequently much beyond their planned life cycle, and undergo many changes. They adapt to environment changes, e.g., introduction of new device drivers, evolution of the operating system, and changes in the underlying data base. They also evolve to serve new and changing user requirements. Ongoing quality tasks include test and analysis of new and modified code, re-execution of system tests, and extensive record-keeping.

Chipmunk maintains a database for tracking problems. This database serves a dual purpose of tracking and prioritizing actual, known program faults and their resolution and managing communication with users who file problem reports. Even at initial release, the database usually includes some known faults, because market pressure seldom allows correcting all known faults before product release. Moreover, “bugs” in the database are not always and uniquely associated with real program faults. Some problems reported by users are misunderstandings and feature requests, and many distinct reports turn out to be duplicates and are eventually consolidated.

Chipmunk designates relatively major revisions, involving several developers, as “point releases,” and smaller revisions as “patch level” releases. The full quality process is repeated in miniature for each point release, including everything from inspection of revised requirements to design and execution of new unit, integration, system, and acceptance test cases. A major point release is likely even to repeat a period of beta testing.

Patch level revisions are often urgent for at least some customers. For example, a patch level revision is likely when a fault prevents some customers from using the software, or when a new security vulnerability is discovered. Test and analysis for patch level revisions is abbreviated, and automation is particularly important so that a reasonable level of assurance can be obtained with very fast turnaround. In particular, when fixing one fault, it is all too easy to introduce a new fault or re-introduce faults that have occurred in the past. Chipmunk maintains an extensive suite of *regression tests* to detect these. The Chipmunk development environment supports recording, classification, and automatic re-execution of test cases. Each point release must undergo complete regression testing before release, but patch level revisions may be released with a subset of regression tests that run unattended overnight. Developers add new regression test cases as faults are discovered and repaired.

1.7 How Can the Development Process be Improved?

As part of an overall process improvement program, Chipmunk has implemented a quality improvement program. In the past, the quality team encountered the same defects in project after project. The quality improvement program tracks and classifies faults to identify the human errors that cause them and weaknesses in test and analysis that allow them to remain undetected.

Chipmunk quality improvement group members are drawn from developers and quality specialists on several project teams. The group produces recommendations that may include modifications to development and test practices, tool and technology support, and management practices. The explicit attention to buffer overflow in networked applications at Chipmunk is the result of failure analysis in previous projects.

Fault analysis and process improvement comprise four main phases: defining the data to be collected and implementing procedures for collecting it; analyzing collected data to identify important fault classes; analyzing selected fault classes to identify weaknesses in development and quality measures; and adjusting the quality and development process.

Collection of data is particularly crucial, and often difficult. Earlier attempts by Chipmunk quality teams to impose fault data collection practices were a dismal failure. The quality team possessed neither carrots nor sticks to motivate developers under schedule pressure. An overall process improvement program undertaken by the Chipmunk software division provided an opportunity to better integrate fault data collection with other practices, including the normal procedure for assigning, tracking, and reviewing development work assignments. Quality process improvement is distinct from the goal of improving an individual product, but initial data collection is integrated in the same bug tracking system, which in turn is integrated with the revision and configuration control system used by Chipmunk developers.

The quality improvement group defines the information that must be collected for faultiness data to be useful, and the format and organization of that data. Participation of developers in designing the data collection process is essential to balance the cost of data collection and analysis with its utility, and to build acceptance among developers.

Data from several projects over time are aggregated and classified to identify classes of faults that are important because they occur frequently, because they cause particularly severe failures, or because they are costly to repair. These faults are analyzed to understand how they are initially introduced and why they escape detection. The improvement steps recommended by the quality improvement group may include specific analysis or testing steps for earlier fault detection, but they may also include design rules and modifications to development and even management practices. An important part of each recommended practice is an accompanying recommendation for measuring the impact of the change.

Summary

The quality process has three distinct goals: Improving a software product (by preventing, detecting, and removing faults), assessing the quality of the software product (with respect to explicit quality goals), and improving the long-term quality and cost-effectiveness of the quality process itself. Each goal requires weaving quality assurance and improvement activities into an overall development process, from product inception through deployment, evolution, and retirement.

Each organization must devise, evaluate, and refine an approach suited to that organization and application domain. A well-designed approach will invariably combine several test and analysis techniques, spread across stages of development. An array of fault detection techniques are distributed across development stages so that faults are removed as soon as possible. The overall cost and cost-effectiveness of techniques depends to a large degree on the extent to which they can be incrementally re-applied as the product evolves.

Further Reading

This book deals primarily with software analysis and testing to improve and assess the dependability of software. That is not because qualities other than dependability are unimportant, but rather because they require their own specialized approaches and techniques. We offer here a few starting points for considering some other important properties that interact with dependability. Norman's *Design of Everyday Things* [Nor90] is a classic introduction to design for usability, with basic principles that apply to both hardware and software artifacts. A primary reference on usability for interactive computer software, and particularly for web applications, is Nielsen's *Design Web Usability* [Nie00]. Bishop's text [Bis02] is a good introduction to computer security. The most comprehensive introduction to software safety is Leveson's *Safeware* [Lev95].

Exercises

1.1. Philip has studied "just-in-time" industrial production methods, and is convinced that they should be applied to every aspect of software development. He argues that test case design should be performed just before the first opportunity to execute the newly designed test cases, never earlier. What positive and negative consequences do you foresee for this just-in-time test case design approach?

1.2. A newly hired project manager at Chipmunk questions why the quality manager is involved in the feasibility study phase of the project, rather than joining the team only when the project has been approved, as at the new manager's previous company. What argument(s) might the quality manager offer in favor of her involvement in the feasibility study?

1.3. Chipmunk procedures call for peer review not only of each source code module, but also of test cases and scaffolding for testing that module. Anita argues that inspecting test suites is a waste of time; any time spent on inspecting a test case designed to detect a particular class of fault could more effectively be spent inspecting the source code to detect that class of fault. Anita's project manager, on the other hand, argues that inspecting test cases and scaffolding can be cost-effective when considered over the whole lifetime of a software product. What argument(s) might Anita's manager offer in favor of this conclusion?

1.4. The spiral model of software development prescribes sequencing incremental prototyping phases for risk reduction, beginning with the most important project risks. Architectural design for testability involves, in addition to defining testable interface specifications for each major module, establishing a build order that supports thorough testing after each stage of construction. How might spiral development and design for test be complementary or in conflict?

1.5. You manage an on-line service that sells down-loadable video recordings of classic movies. A typical down-load takes one hour, and an interrupted down-load must be restarted from the beginning. The number of customers engaged in a download at any given time ranges from about 10 to about 150 during peak hours. On average, your system goes down (dropping all connections) about two times per week, for an average of three minutes each time. If you can double availability or double mean time between failures, but not both, which will you choose? Why?

1.6. Having no a priori operational profile for reliability measurement, Chipmunk will depend on alpha and beta testing to assess the readiness of their on-line purchase functionality for public release. Beta testing will be carried out in retail outlets, by retail store personnel and then by customers with retail store personnel looking on. How might this beta testing still be misleading with respect to reliability of the software as it will be used at home and work by actual customers? What might Chipmunk do to ameliorate potential problems from this reliability mis-estimation?

1.7. The junior test designers of Chipmunk Computers are annoyed by the procedures for storing test cases together with scaffolding, test results and related documentation. They blame the extra effort needed to produce and store such data for delays in test design and execution. They argue for reducing the data to store to the minimum required for re-executing test cases, eliminating details of test documentation and limiting test results to the information needed for generating oracles.

What argument(s) might the quality manager use to convince the junior test designers of the usefulness of storing all this information?