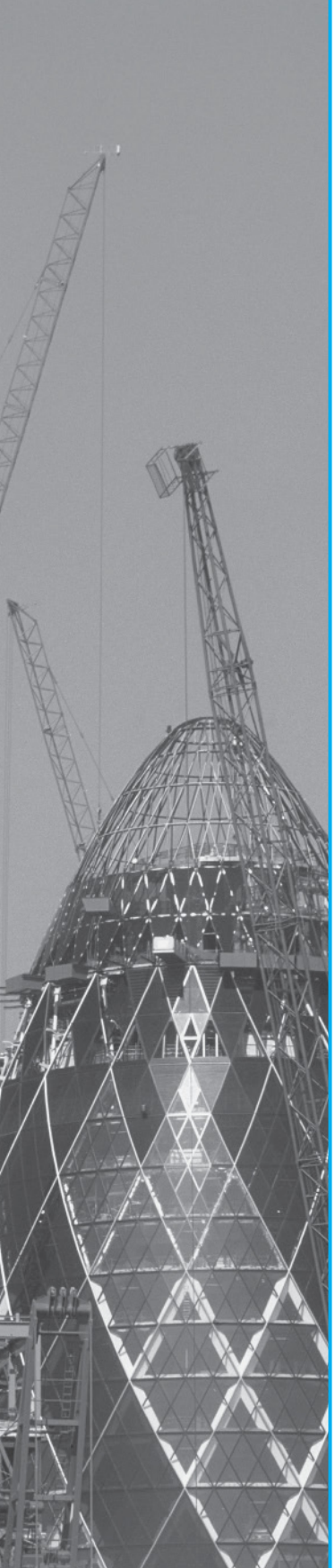

10 Dependable systems

Objectives

The objective of this chapter is to introduce the topic of software dependability and what is involved in developing dependable software systems. When you have read this chapter, you will:

- understand why dependability and security are important attributes for all software systems;
- understand the five important dimensions of dependability, namely, availability, reliability, safety, security, and resilience;
- understand the notion of sociotechnical systems and why we have to consider these systems as a whole rather than just software systems;
- know why redundancy and diversity are the fundamental concepts used in achieving dependable systems and processes;
- be aware of the potential for using formal methods in dependable systems engineering.

Contents

As computer systems have become deeply embedded in our business and personal lives, the problems that result from system and software failure are increasing. A failure of server software in an e-commerce company could lead to a major loss of revenue and customers for that company. A software error in an embedded control system in a car could lead to expensive recalls of that model for repair and, in the worst case, could be a contributory factor in accidents. The infection of company PCs with malware requires expensive clean-up operations to sort out the problem and could lead to the loss of or damage to sensitive information.

Because software-intensive systems are so important to governments, companies, and individuals, we have to be able to trust these systems. The software should be available when it is needed, and it should operate correctly without undesirable side effects, such as unauthorized information disclosure. In short, we should be able to depend on our software systems.

The term *dependability* was proposed by Jean-Claude Laprie in 1995 to cover the related systems attributes of availability, reliability, safety, and security. His ideas were revised over the next few years and are discussed in a definitive paper published in 2004 (Avizienis et al. 2004). As I discuss in Section 10.1, these properties are inextricably linked, so having a single term to cover them all makes sense.

The dependability of systems is usually more important than their detailed functionality for the following reasons:

1. *System failures affect a large number of people* Many systems include functionality that is rarely used. If this functionality were left out of the system, only a small number of users would be affected. System failures that affect the availability of a system potentially affect all users of the system. Unavailable systems may mean that normal business is impossible.
2. *Users often reject systems that are unreliable, unsafe, or insecure* If users find that a system is unreliable or insecure, they will refuse to use it. Furthermore, they may also refuse to buy or use other products from the company that produced the unreliable system. They do not want a repetition of their bad experience with an undependable system.
3. *System failure costs may be enormous* For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system. Failures in systems that control critical infrastructure such as the power network have widespread economic consequences.
4. *Undependable systems may cause information loss* Data is very expensive to collect and maintain; it is usually worth much more than the computer system on which it is processed. The cost of recovering lost or corrupt data is usually very high.

However, a system can be useful without it being very dependable. I don't think that the word processor that I used to write this book is a very dependable system. It sometimes freezes and has to be restarted. Nevertheless, because it is very useful,

Critical systems

Some classes of system are "critical systems" where system failure may result in injury to people, damage to the environment, or extensive economic losses. Examples of critical systems include embedded systems in medical devices, such as an insulin pump (safety-critical), spacecraft navigation systems (mission-critical), and online money transfer systems (business critical).

Critical systems are very expensive to develop. Not only must they be developed so that failures are very rare, but they must also include recovery mechanisms to be used if and when failures occur.

http://software-engineering-book.com/web/critical-systems/

I am prepared to tolerate occasional failure. However, to reflect my lack of trust in the system, I save my work frequently and keep multiple backup copies of it. I compensate for the lack of system dependability by actions that limit the damage that could result from system failure.

Building dependable software is part of the more general process of dependable systems engineering. As I discuss in Section 10.2, software is always part of a broader system. It executes in an operational environment that includes the hardware on which the software executes, the human users of that software and the organizational or business processes where the software is used. When designing a dependable system, you therefore have to consider:

1. *Hardware failure* System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors, because of environmental factors such as dampness or high temperatures, or because components have reached the end of their natural life.
2. *Software failure* System software may fail because of mistakes in its specification, design, or implementation.
3. *Operational failure* Human users may fail to use or operate the system as intended by its designers. As hardware and software have become more reliable, failures in operation are now, perhaps, the largest single cause of system failures.

These failures are often interrelated. A failed hardware component may mean system operators have to cope with an unexpected situation and additional workload. This puts them under stress, and people under stress often make mistakes. These mistakes can cause the software to fail, which means more work for operators, even more stress, and so on.

As a result, it is particularly important that designers of dependable, software-intensive systems take a holistic sociotechnical systems perspective rather than focus on a single aspect of the system such as its software or hardware. If hardware, software, and operational processes are designed separately, without taking into account the potential weaknesses of other parts of the system, then it is more likely that errors will occur at the interfaces between the different parts of the system.

10.1 Dependability properties

All of us are familiar with the problem of computer system failure. For no obvious reason, our computers sometimes crash or go wrong in some way. Programs running on these computers may not operate as expected and occasionally may corrupt the data that is managed by the system. We have learned to live with these failures, but few of us completely trust the personal computers that we normally use.

The dependability of a computer system is a property of the system that reflects its trustworthiness. Trustworthiness here essentially means the degree of confidence a user has that the system will operate as they expect and that the system will not "fail" in normal use. It is not meaningful to express dependability numerically. Rather, relative terms such as "not dependable," "very dependable," and "ultra-dependable" can reflect the degree of trust that we might have in a system.

There are five principal dimensions to dependability, as I have shown in Figure 10.1.

1. *Availability* Informally, the availability of a system is the probability that it will be up and running and able to deliver useful services to users at any given time.
2. *Reliability* Informally, the reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.
3. *Safety* Informally, the safety of a system is a judgment of how likely it is that the system will cause damage to people or its environment.
4. *Security* Informally, the security of a system is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.
5. *Resilience* Informally, the resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks. Resilience is a more recent addition to the set of dependability properties that were originally suggested by Laprie.

The dependability properties shown in Figure 10.1 are complex properties that can be broken down into several simpler properties. For example, security includes "integrity" (ensuring that the systems program and data are not damaged) and "confidentiality" (ensuring that information can only be accessed by people who are authorized). Reliability includes "correctness" (ensuring the system services are as specified), "precision" (ensuring information is delivered at an appropriate level of detail), and "timeliness" (ensuring that information is delivered when it is required).

Of course, not all dependability properties are critical for all systems. For the insulin pump system, introduced in Chapter 1, the most important properties are reliability (it must deliver the correct dose of insulin) and safety (it must never deliver a dangerous dose of insulin). Security is not an issue as the pump does not store confidential information. It is not networked and so cannot be maliciously attacked. For

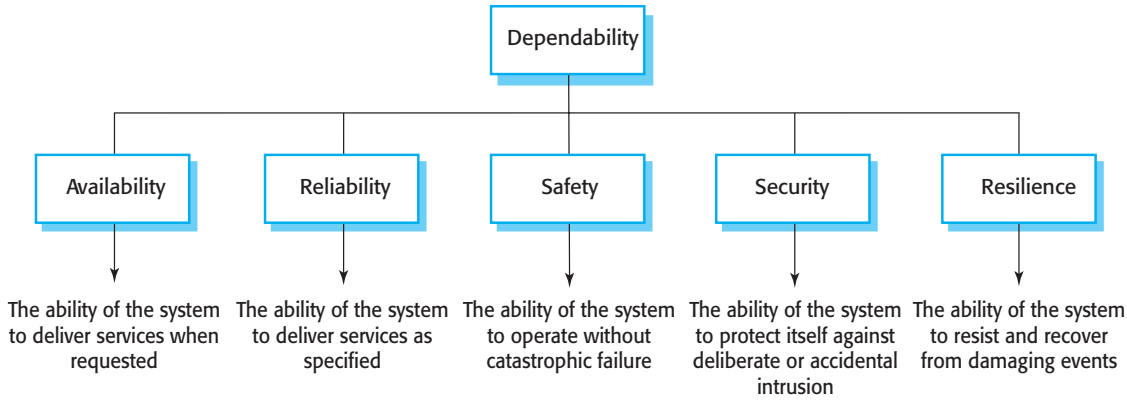

Figure 10.1 Principal dependability properties

the wilderness weather system, availability and reliability are the most important properties because the costs of repair may be very high. For the Mentcare patient information system, security and resilience are particularly important because of the sensitive private data that is maintained and the need for the system to be available for patient consultations.

Other system properties are closely related to these five dependability properties and influence a system's dependability:

1. *Repairability* System failures are inevitable, but the disruption caused by failure can be minimized if the system can be repaired quickly. It must be possible to diagnose the problem, access the component that has failed, and make changes to fix that component. Repairability in software is enhanced when the organization using the system has access to the source code and has the skills to make changes to it. Open-source software makes this easier, but the reuse of components can make it more difficult.

2. *Maintainability* As systems are used, new requirements emerge, and it is important to maintain the value of a system by changing it to include these new requirements. Maintainable software is software that can be adapted economically to cope with new requirements, and where there is a low probability that making changes will introduce new errors into the system.

3. *Error tolerance* This property can be considered as part of usability and reflects the extent to which the system has been designed, so that user input errors are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.

The notion of system dependability as an encompassing property was introduced because the dependability properties of availability, security, reliability, safety, and resilience are closely related. Safe system operation usually depends on the system being available and operating reliably. A system may become unreliable because an intruder has corrupted its data. Denial-of-service attacks on a system are intended to

compromise the system's availability. If a system is infected with a virus, you cannot then be confident in its reliability or safety because the virus may change its behavior.

To develop dependable software, you therefore need to ensure that:

1. You avoid the introduction of accidental errors into the system during software specification and development.
2. You design verification and validation processes that are effective in discovering residual errors that affect the dependability of the system.
3. You design the system to be fault tolerant so that it can continue working when things go wrong.
4. You design protection mechanisms that guard against external attacks that can compromise the availability or security of the system.
5. You configure the deployed system and its supporting software correctly for its operating environment.
6. You include system capabilities to recognize external cyberattacks and to resist these attacks.
7. You design systems so that they can quickly recover from system failures and cyberattacks without the loss of critical data.

The need for fault tolerance means that dependable systems have to include redundant code to help them monitor themselves, detect erroneous states, and recover from faults before failures occur. This affects the performance of systems, as additional checking is required each time the system executes. Therefore, designers usually have to trade off performance and dependability. You may need to leave checks out of the system because these slow the system down. However, the consequential risk here is that the system fails because a fault has not been detected.

Building dependable systems is expensive. Increasing the dependability of a system means that you incur extra costs for system design, implementation, and validation. Verification and validation costs are particularly high for systems that must be ultra-dependable such as safety-critical control systems. As well as validating that the system meets its requirements, the validation process may have to prove to an external regulator that the system is safe. For example, aircraft systems have to demonstrate to regulators, such as the Federal Aviation Authority, that the probability of a catastrophic system failure that affects aircraft safety is extremely low.

Figure 10.2 shows the relationship between costs and incremental improvements in dependability. If your software is not very dependable, you can get significant improvements fairly cheaply by using better software engineering. However, if you are already using good practice, the costs of improvement are much greater, and the benefits from that improvement are less.

There is also the problem of testing software to demonstrate that it is dependable. Solving this problem relies on running many tests and looking at the number of failures that occur. As your software becomes more dependable, you see fewer and

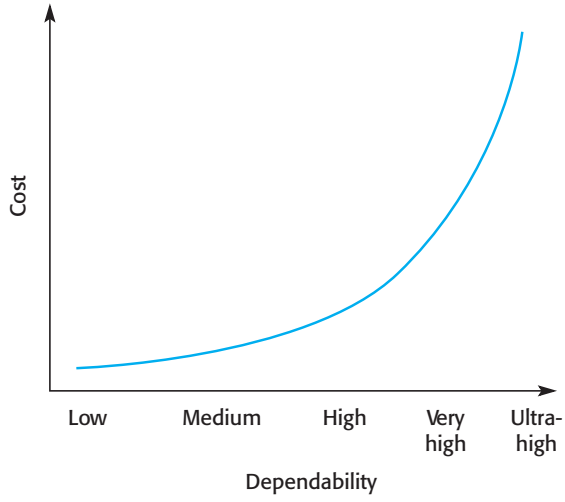

Figure 10.2 Cost/dependability curve

fewer failures. Consequently, more and more tests are needed to try and assess how many problems remain in the software. Testing is a very expensive process, so this can significantly increase the cost of high-dependability systems.

10.2 Sociotechnical systems

In a computer system, the software and the hardware are interdependent. Without hardware, a software system is an abstraction, which is simply a representation of some human knowledge and ideas. Without software, hardware is a set of inert electronic devices. However, if you put them together to form a system, you create a machine that can carry out complex computations and deliver the results of these computations to its environment.

This illustrates one of the fundamental characteristics of a system—it is more than the sum of its parts. Systems have properties that become apparent only when their components are integrated and operate together. Software systems are not isolated systems but are part of more extensive systems that have a human, social, or organizational purpose. Therefore software engineering is not an isolated activity but is an intrinsic part of systems engineering (Chapter 19).

For example, the wilderness weather system software controls the instruments in a weather station. It communicates with other software systems and is a part of wider national and international weather forecasting systems. As well as hardware and software, these systems include processes for forecasting the weather and people who operate the system and analyze its outputs. The system also includes the organizations that depend on the system to help them provide weather forecasts to individuals, government and industry.

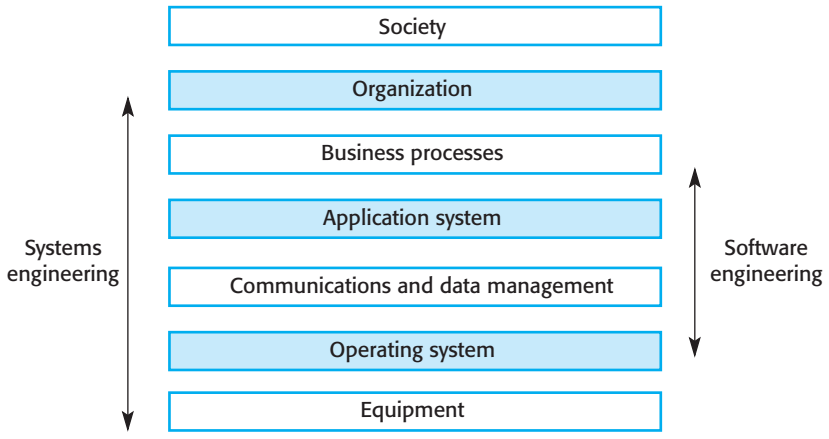

Figure 10.3 The sociotechnical systems stack

These broader systems are called *sociotechnical systems*. They include nontechnical elements such as people, processes, and regulations, as well as technical components such as computers, software, and other equipment. System dependability is influenced by all of the elements in a sociotechnical system—hardware, software, people, and organizations.

Sociotechnical systems are so complex that it is impossible to understand them as a whole. Rather, you have to view them as layers, as shown in Figure 10.3. These layers make up the sociotechnical systems stack:

1. *The equipment layer* is composed of hardware devices, some of which may be computers.
2. *The operating system layer* interacts with the hardware and provides a set of common facilities for higher software layers in the system.
3. *The communications and data management layer* extends the operating system facilities and provides an interface that allows interaction with more extensive functionality, such as access to remote systems and access to a system database. This is sometimes called middleware, as it is in between the application and the operating system.
4. *The application layer* delivers the application-specific functionality that is required. There may be many different application programs in this layer.
5. *The business process layer* includes the organizational business processes, which make use of the software system.
6. *The organizational layer* includes higher-level strategic processes as well as business rules, policies, and norms that should be followed when using the system.
7. *The social layer* refers to the laws and regulations of society that govern the operation of the system.

Notice that there is no separate "software layer." Software of one kind or another is an important part of all of the layers in the sociotechnical system. Equipment is controlled by embedded software; the operating system and applications are software. Business processes, organizations, and society rely on the Internet (software) and other global software systems.

In principle, most interactions should be between neighboring layers in the stack, with each layer hiding the detail of the layer below from the layer above. In practice, however, there can be unexpected interactions between layers, which result in problems for the system as a whole. For example, say there is a change in the law governing access to personal information. This comes from the social layer. It leads to new organizational procedures and changes to the business processes. The application system itself may not be able to provide the required level of privacy, so changes may have to be implemented in the communications and data management layer.

Thinking holistically about systems, rather than simply considering software in isolation, is essential when considering software security and dependability. Software itself is intangible and, even when damaged, is easily and cheaply restored. However, when these software failures ripple through other parts of the system, they affect the software's physical and human environment. Here, the consequences of failure are more significant. Important data may be lost or corrupted. People may have to do extra work to contain or recover from the failure; for example, equipment may be damaged, data may be lost or corrupted, or confidentiality may be breached, with unknown consequences.

You must, therefore, take a system-level view when you are designing software that has to be dependable and secure. You have to take into account the consequences of software failures for other elements in the system. You also need to understand how these other system elements may be the cause of software failure and how they can help to protect against and recover from software failures.

It is important to ensure that, wherever possible, software failure does not lead to overall system failure. You must therefore examine how the software interacts with its immediate environment to ensure that:

1. Software failures are, as far as possible, contained within the enclosing layer of the system stack and do not seriously affect the operation of other layers in the system.
2. You understand how faults and failures in the other layers of the systems stack may affect the software. You may also consider how checks may be built into the software to help detect these failures, and how support can be provided for recovering from failure.

As software is inherently flexible, unexpected system problems are often left to software engineers to solve. Say a radar installation has been sited so that ghosting of the radar image occurs. It is impractical to move the radar to a site with less interference, so the systems engineers have to find another way of removing this

ghosting. Their solution may be to enhance the image-processing capabilities of the software to remove the ghost images. This may slow down the software so that its performance becomes unacceptable. The problem may then be characterized as a software failure, whereas, in fact, it is a failure in the design process for the system as a whole.

This sort of situation, in which software engineers are left with the problem of enhancing software capabilities without increasing hardware cost, is very common. Many so-called software failures are not a consequence of inherent software problems but rather are the result of trying to change the software to accommodate modified system engineering requirements. A good example was the failure of the Denver airport baggage system (Swartz 1996), where the controlling software was expected to deal with limitations of the equipment used.

10.2.1 Regulation and compliance

The general model of economic organization that is now almost universal in the world is that privately owned companies offer goods and services and make a profit on these. We have a competitive environment so that these companies may compete on cost, on quality, on delivery time, and so on. However, to ensure the safety of their citizens, most governments limit the freedom of privately owned companies so that they must follow certain standards to ensure that their products are safe and secure. A company therefore cannot offer products for sale more cheaply because they have reduced their costs by reducing the safety of their products.

Governments have created a set of rules and regulations in different areas that define standards for safety and security. They have also established regulators or regulatory bodies whose job is to ensure that companies offering products in an area comply with these rules. Regulators have wide powers. They can fine companies and even imprison directors if regulations are breached. They may have a licensing role (e.g., in the aviation and nuclear industries) where they must issue a license before a new system may be used. Therefore, aircraft manufacturers have to have a certificate of airworthiness from the regulator in each country where the aircraft is used.

To achieve certification, companies that are developing safety-critical systems have to produce an extensive safety case (discussed in Chapter 13) that shows that rules and regulations have been followed. The case must convince a regulator that the system can operate safely. Developing such a safety case is very costly. It can be as expensive to develop the documentation for certification as it is to develop the system itself.

Regulation and compliance (following the rules) applies to the sociotechnical system as a whole and not simply the software element of that system. For example, a regulator in the nuclear industry is concerned that in the event of overheating, a nuclear reactor will not release radioactivity into the environment. Arguments to convince the regulator that this is the case may be based on software protection systems, the operational process used to monitor the reactor core and the integrity of structures that contain any release of radioactivity.

Each of these elements has to have its own safety case. So, the protection system must have a safety case that demonstrates that the software will operate correctly and shut down the reactor as intended. The overall case must also show that if the software protection system fails, there are alternative safety mechanisms, which do not rely on software, that are invoked.

10.3 Redundancy and diversity

Component failures in any system are inevitable. People make mistakes, undiscovered bugs in software cause undesirable behavior, and hardware burns out. We use a range of strategies to reduce the number of human failures such as replacing hardware components before the end of their predicted lifetime and checking software using static analysis tools. However, we cannot be sure that these will eliminate component failures. We should therefore design systems so that individual component failures do not lead to overall system failure.

Strategies to achieve and enhance dependability rely on both redundancy and diversity. Redundancy means that spare capacity is included in a system that can be used if part of that system fails. Diversity means that redundant components of the system are of different types, thus increasing the chances that they will not fail in exactly the same way.

We use redundancy and diversity to enhance dependability in our everyday lives. Commonly, to secure our homes we use more than one lock (redundancy), and, usually, the locks used are of different types (diversity). This means that if intruders find a way to defeat one of the locks, they have to find a different way of defeating the other locks before they can gain entry. As a matter of routine, we should all back up our computers and so maintain redundant copies of our data. To avoid problems with disk failure, backups should be kept on a separate, diverse, external device.

Software systems that are designed for dependability may include redundant components that provide the same functionality as other system components. These are switched into the system if the primary component fails. If these redundant components are diverse, that is, not the same as other components, a common fault in replicated components will not result in a system failure. Another form of redundancy is the inclusion of checking code, which is not strictly necessary for the system to function. This code can detect some kinds of problems, such as data corruption, before they cause failures. It can invoke recovery mechanisms to correct problems to ensure that the system continues to operate.

In systems for which availability is a critical requirement, redundant servers are normally used. These automatically come into operation if a designated server fails. Sometimes, to ensure that attacks on the system cannot exploit a common vulnerability, these servers may be of different types and may run different operating systems. Using different operating systems is an example of software diversity and

The Ariane 5 explosion

In 1996, the European Space Agency's Ariane 5 rocket exploded 37 seconds after lift-off on its maiden flight. The fault was caused by a software systems failure. There was a backup system but it was not diverse, and so the software in the backup computer failed in exactly the same way. The rocket and its satellite payload were destroyed.

http://software-engineering-book.com/web/ariane/

redundancy, where similar functionality is provided in different ways. (I discuss software diversity in more detail in Chapter 12.)

Diversity and redundancy may also be also used in the design of dependable software development processes. Dependable development processes avoid the introduction of faults into a system. In a dependable process, activities such as software validation do not rely on a single tool or technique. This improves software dependability because it reduces the chances of process failure, where human errors made during the software development process lead to software errors.

For example, validation activities may include program testing, manual program inspections, and static analysis as fault-finding techniques. Any one of these techniques might find faults that are missed by the other methods. Furthermore, different team members may be responsible for the same process activity (e.g., a program inspection). People tackle tasks in different ways depending on their personality, experience, and education, so this kind of redundancy provides a diverse perspective on the system.

However, as I discuss in Chapter 11, using software redundancy and diversity can itself introduce bugs into software. Diversity and redundancy make systems more complex and usually harder to understand. Not only is there more code to write and check, but additional functionality must also be added to the system to detect component failure and to switch control to alternative components. This additional complexity means that it is more likely that programmers will make errors and less likely that people checking the system will find these errors.

Some engineers therefore think that, as software cannot wear out, it is best to avoid software redundancy and diversity. Their view is that the best approach is to design the software to be as simple as possible, with extremely rigorous software verification and validation procedures (Parnas, van Schouwen, and Shu 1990). More can be spent on verification and validation because of the savings that result from not having to develop redundant software components.

Both approaches are used in commercial, safety-critical software systems. For example, the Airbus 340 flight control hardware and software is both diverse and redundant. The flight control software on the Boeing 777 runs on redundant hardware, but each computer runs the same software, which has been very extensively validated. The Boeing 777 flight control system designers have focused on simplicity rather than redundancy. Both of these aircraft are very reliable, so both the diverse and the simple approach to dependability can clearly be successful.

Dependable operational processes

This chapter discusses dependable development processes, but system operational processes are equally important contributors for system dependability. In designing these operational processes, you have to take into account human factors and always bear in mind that people are liable to make mistakes when using a system. A dependable process should be designed to avoid human errors, and, when mistakes are made, the software should detect the mistakes and allow them to be corrected.

http://software-engineering-book.com/web/human-error/

10.4 Dependable processes

Dependable software processes are software processes that are designed to produce dependable software. The rationale for investing in dependable processes is that a good software process is likely to lead to delivered software that contains fewer errors and is therefore less likely to fail in execution. A company using a dependable process can be sure that the process has been properly enacted and documented and that appropriate development techniques have been used for critical systems development. Figure 10.4 shows some of the attributes of dependable software processes.

The evidence that a dependable process has been used is often important in convincing a regulator that the most effective software engineering practice has been applied in developing the software. System developers will normally present a model of the process to a regulator, along with evidence that the process has been followed. The regulator also has to be convinced that the process is used consistently by all of the process participants and that it can be used in different development projects. This means that the process must be explicitly defined and repeatable:

1. An explicitly defined process is one that has a defined process model that is used to drive the software production process. Data must be collected during the process that proves that the development team has followed the process as defined in the process model.
2. A repeatable process is one that does not rely on individual interpretation and judgment. Rather, the process can be repeated across projects and with different team members, irrespective of who is involved in the development. This is particularly important for critical systems, which often have a long development cycle during which there are often significant changes in the development team.

Dependable processes make use of redundancy and diversity to achieve reliability. They often include different activities that have the same aim. For example, program inspections and testing aim to discover errors in a program. The approaches can be used together so that they are likely to find more errors than would be found using one technique on its own.

Process characteristic	Description
Auditable	The process should be understandable by people apart from process participants, who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Robust	The process should be able to recover from failures of individual process activities.
Standardized	A comprehensive set of software development standards covering software production and documentation should be available.

Figure 10.4 Attributes of dependable processes

The activities that are used in dependable processes obviously depend on the type of software that is being developed. In general, however, these activities should be geared toward avoiding the introduction of errors into a system, detecting and removing errors, and maintaining information about the process itself.

Examples of activities that might be included in a dependable process include:

1. Requirements reviews to check that the requirements are, as far as possible, complete and consistent.
2. Requirements management to ensure that changes to the requirements are controlled and that the impact of proposed requirements changes is understood by all developers affected by the change.
3. Formal specification, where a mathematical model of the software is created and analyzed. (I discussed the benefits of formal specification in Section 10.5.) Perhaps its most important benefit is that it forces a very detailed analysis of the system requirements. This analysis itself is likely to discover requirements problems that may have been missed in requirements reviews.
4. System modeling, where the software design is explicitly documented as a set of graphical models and the links between the requirements and these models are explicitly documented. If a model-driven engineering approach is used (see Chapter 5), code may be generated automatically from these models.
5. Design and program inspections, where the different descriptions of the system are inspected and checked by different people. A checklist of common design and programming errors may be used to focus the inspection process.
6. Static analysis, where automated checks are carried out on the source code of the program. These look for anomalies that could indicate programming errors or omissions. (I cover static analysis in Chapter 12.)
7. Test planning and management, where a comprehensive set of system tests is designed. The testing process has to be carefully managed to demonstrate that these tests provide coverage of the system requirements and have been correctly applied in the testing process.

As well as process activities that focus on system development and testing, there must also be well-defined quality management and change management processes. While the specific activities in a dependable process may vary from one company to another, the need for effective quality and change management is universal.

Quality management processes (covered in Chapter 24) establish a set of process and product standards. They also include activities that capture process information to demonstrate that these standards have been followed. For example, there may be a standard defined for carrying out program inspections. The inspection team leader is responsible for documenting the process to show that the inspection standard has been followed.

Change management, discussed in Chapter 25, is concerned with managing changes to a system, ensuring that accepted changes are actually implemented, and confirming that planned releases of the software include the planned changes. One common problem with software is that the wrong components are included in a system build. This can lead to a situation where an executing system includes components that have not been checked during the development process. Configuration management procedures must be defined as part of the change management process to ensure that this does not happen.

As agile methods have become increasingly used, researchers and practitioners have thought carefully about how to use agile approaches in dependable software development (Trimble 2012). Most companies that develop critical software systems have based their development on plan-based processes and have been reluctant to make radical changes to their development process. However, they recognize the value of agile approaches and are exploring how their dependable development processes can be more agile.

As dependable software often requires certification, both process and product documentation have to be produced. Up-front requirements analysis is also essential to discover possible requirements and requirements conflicts that may compromise the safety and security of the system. Formal change analysis is essential to assess the effect of changes on the safety and integrity of the system. These requirements conflict with the general approach in agile development of co-development of the requirements and the system and minimizing documentation.

Although most agile development uses an informal, undocumented process, this is not a fundamental requirement of agility. An agile process may be defined that incorporates techniques such as iterative development, test-first development and user involvement in the development team. As long as the team follows that process and documents their actions, agile techniques can be used. To support this notion, various proposals of modified agile methods have been made that incorporate the requirements of dependable systems engineering (Douglass 2013). These combine the most appropriate techniques from agile and plan-based development.

10.5 Formal methods and dependability

For more than 30 years, researchers have advocated the use of formal methods of software development. Formal methods are mathematical approaches to software development where you define a formal model of the software. You may then formally analyze this model to search for errors and inconsistencies, prove that a program

is consistent with this model, or you may apply a series of correctness-preserving transformations to the model to generate a program. Abrial (Abrial 2009) claims that the use of formal methods can lead to "faultless systems," although he is careful to limit what he means in this claim.

In an excellent survey, Woodcock et al. (Woodcock et al. 2009) discuss industrial applications where formal methods have been successfully applied. These include train control systems (Badeau and Amelot 2005), cash card systems (Hall and Chapman 2002), and flight control systems (Miller et al. 2005). Formal methods are the basis of tools used in static verification, such as the driver verification system used by Microsoft (Ball et al. 2006).

Using a mathematically formal approach to software development was proposed at an early stage in the development of computer science. The idea was that a formal specification and a program could be developed independently. A mathematical proof could then be developed to show that the program and its specification were consistent. Initially, proofs were developed manually but this was a long and expensive process. It quickly became clear that manual proofs could only be developed for very small systems. Program proving is now supported by large-scale automated theorem proving software, which has meant that larger systems can be proved. However, developing the proof obligations for theorem provers is a difficult and specialized task, so formal verification is not widely used.

An alternative approach, which avoids a separate proof activity, is refinement-based development. Here, a formal specification of a system is refined through a series of correctness-preserving transformations to generate the software. Because these are trusted transformations, you can be confident that the generated program is consistent with its formal specification. This was the approach used in the software development for the Paris Metro system (Badeau and Amelot 2005). It used a language called B (Abrial 2010), which was designed to support specification refinement.

Formal methods based on model-checking (Jhala and Majumdar 2009) have been used in a number of systems (Bochot et al. 2009; Calinescu and Kwiatkowska 2009). These systems rely on constructing or generating a formal state model of a system and using a model-checker to check that properties of the model, such as safety properties, always hold. The model-checking program exhaustively analyzes the specification and either reports that the system property is satisfied by the model or presents an example that shows it is not satisfied. If a model can be automatically or systematically generated from a program, this means that bugs in the program can be uncovered. (I cover model checking in safety-critical systems in Chapter 12.)

Formal methods for software engineering are effective for discovering or avoiding two classes of error in software representations:

1. *Specification and design errors and omissions.* The process of developing and analyzing a formal model of the software may reveal errors and omissions in the software requirements. If the model is generated automatically or systematically from source code, analysis using model checking can discover undesirable states that may occur, such as deadlock in a concurrent system.

Formal specification techniques

Formal system specifications may be expressed using two fundamental approaches, either as models of the system interfaces (algebraic specifications) or as models of the system state. An extra web chapter on this topic shows examples of both of these approaches. The chapter includes a formal specification of part of the insulin pump system.

http://software-engineering-book.com/web/formal-methods/ (web chapter)

2. *Inconsistencies between a specification and a program.* If a refinement method is used, mistakes made by developers that make the software inconsistent with the specification are avoided. Program proving discovers inconsistencies between a program and its specification.

The starting point for all formal methods is a mathematical system model, which acts as a system specification. To create this model, you translate the system's user requirements, which are expressed in natural language, diagrams, and tables, into a mathematical language that has formally defined semantics. The formal specification is an unambiguous description of what the system should do.

Formal specifications are the most precise way of specifying systems, and so reduce the scope for misunderstanding. Many supporters of formal methods believe that creating formal specification, even without refinement or program proof, is worthwhile. Constructing a formal specification forces a detailed analysis of the requirements and this is an effective way of discovering requirements problems. In a natural language specification, errors can be concealed by the imprecision of the language. This is not the case if the system is formally specified.

The advantages of developing a formal specification and using it in a formal development process are:

1. As you develop a formal specification in detail, you develop a deep and detailed understanding of the system requirements. Requirements problems that are discovered early are usually much cheaper to correct than if they are found at later stages in the development process.
2. As the specification is expressed in a language with formally defined semantics, you can analyze it automatically to discover inconsistencies and incompleteness.
3. If you use a method such as the B method, you can transform the formal specification into a program through a sequence of correctness-preserving transformations. The resulting program is therefore guaranteed to meet its specification.
4. Program testing costs may be reduced because you have verified the program against its specification. For example, in the development of the software for the Paris Metro systems, the use of refinement meant that there was no need for software component testing and only system testing was required.

Woodcock's survey (Woodcock et al. 2009) found that users of formal methods reported fewer errors in the delivered software. Neither the costs nor the time needed for software development were higher than in comparable development projects. There were significant benefits in using formal approaches in safety critical systems that required regulator certification. The documentation produced was an important part of the safety case (see Chapter 12) for the system.

In spite of these advantages, formal methods have had limited impact on practical software development, even for critical systems. Woodcock reports on 62 projects over 25 years that used formal methods. Even if we allow for projects that used these techniques but did not report their use, this is a tiny fraction of the total number of critical systems developed in that time. Industry has been reluctant to adopt formal methods for a number of reasons:

1. Problem owners and domain experts cannot understand a formal specification, so they cannot check that it accurately represents their requirements. Software engineers, who understand the formal specification, may not understand the application domain, so they too cannot be sure that the formal specification is an accurate reflection of the system requirements.
2. It is fairly easy to quantify the costs of creating a formal specification, but more difficult to estimate the possible cost savings that will result from its use. As a result, managers are unwilling to take the risk of adopting formal methods. They are unconvinced by reports of success as, by and large, these came from atypical projects where the developers were keen advocates of a formal approach.
3. Most software engineers have not been trained to use formal specification languages. Hence, they are reluctant to propose their use in development processes.
4. It is difficult to scale current formal methods up to very large systems. When formal methods are used, it is mostly for specifying critical kernel software rather than complete systems.
5. Tool support for formal methods is limited, and the available tools are often open source and difficult to use. The market is too small for commercial tool providers.
6. Formal methods are not compatible with agile development where programs are developed incrementally. This is not a major issue, however, as most critical systems are still developed using a plan-based approach.

Parnas, an early advocate of formal development, has criticized current formal methods and claims that these have started from a fundamentally wrong premise (Parnas 2010). He believes that these methods will not gain acceptance until they are radically simplified, which will require a different type of mathematics as a basis. My own view is that even this will not mean that formal methods are routinely adopted for critical systems engineering unless it can be clearly demonstrated that their adoption and use is cost-effective, compared to other software engineering methods.