Software Enginnering-2
Software Engineering Program

Course Content in Software Enginnering-2
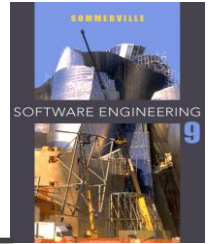
Topics covered

✧ Redundancy and diversity

  ▪ Fundamental approaches to achieve fault tolerance.

✧ Dependable processes

  ▪ How the use of dependable processes leads to dependable systems

✧ Dependable systems architectures

 ▪ Architectural patterns for software fault tolerance

✧ Dependable programming

 ▪ Guidelines for programming to avoid errors.

Dependable system architectures

✧ Dependable systems architectures are used in situations where fault tolerance is essential. These architectures are generally all based on redundancy and diversity.

✧ Examples of situations where dependable architectures are used:

- Flight control systems, where system failure could threaten the safety of passengers
- Reactor systems where failure of a control system could lead to a chemical or nuclear emergency
- Telecommunication systems, where there is a need for 24/7 availability.

Protection systems
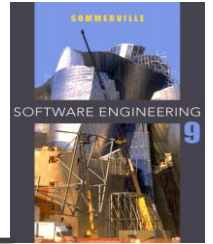
✧ A specialized system that is associated with some other control system, which can take emergency action if a failure occurs.

- System to stop a train if it passes a red light
- System to shut down a reactor if temperature/pressure are too high

✧ Protection systems independently monitor the controlled system and the environment.

✧ If a problem is detected, it issues commands to take emergency action to shut down the system and avoid a catastrophe.

Protection system architecture
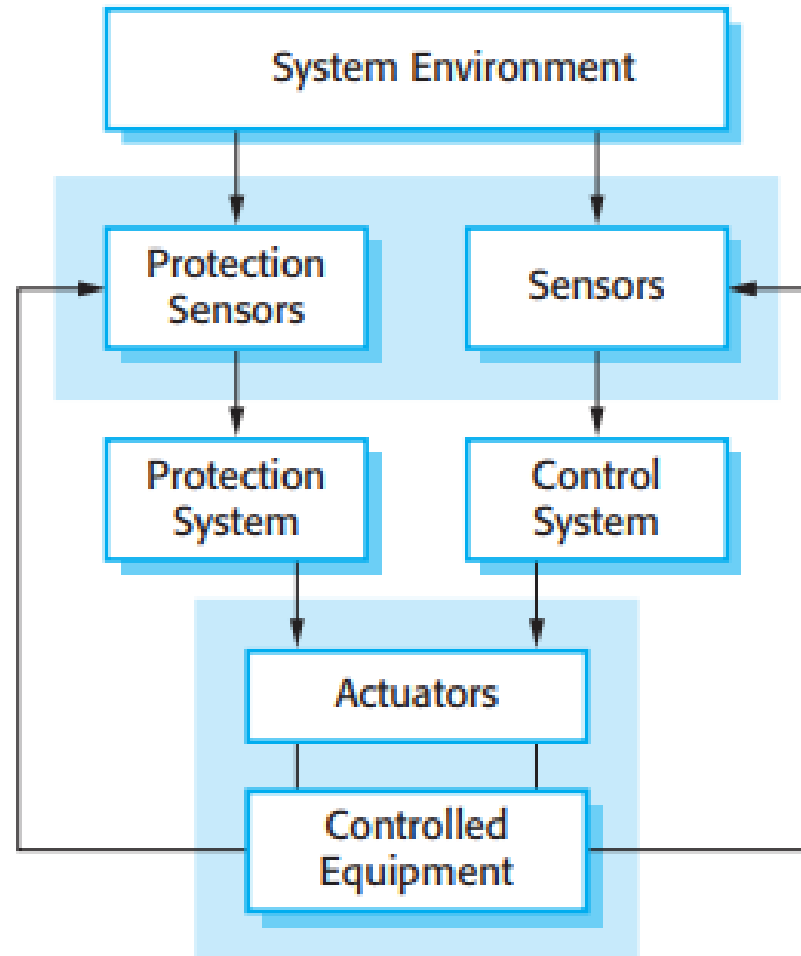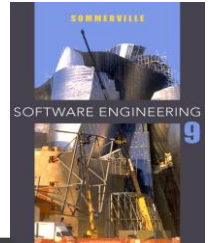
Protection system functionality

✧ Protection systems are redundant because they include monitoring and control capabilities that replicate those in the control software.

✧ Protection systems should be diverse and use different technology from the control software.

✧ They are simpler than the control system so more effort can be expended in validation and dependability assurance.

✧ Aim is to ensure that there is a low probability of failure on demand for the protection system.

Self-monitoring architectures

✧ Multi-channel architectures where the system monitors its own operations and takes action if inconsistencies are detected.

✧ The same computation is carried out on each channel and the results are compared. If the results are identical and are produced at the same time, then it is assumed that the system is operating correctly.

✧ If the results are different, then a failure is assumed and a failure exception is raised.

Self-monitoring architecture
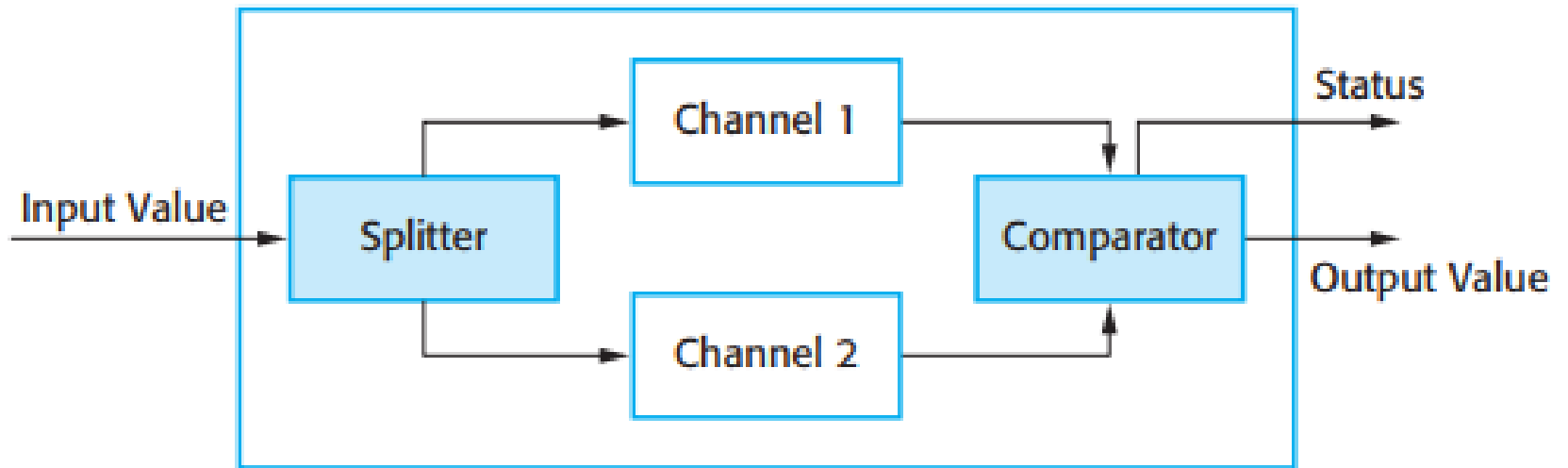
Self-monitoring systems

✧ Hardware in each channel has to be diverse so that common mode hardware failure will not lead to each channel producing the same results.

✧ Software in each channel must also be diverse, otherwise the same software error would affect each channel.

✧ If high-availability is required, you may use several self-checking systems in parallel.

   ▪ This is the approach used in the Airbus family of aircraft for their flight control systems.

Airbus flight control system architecture
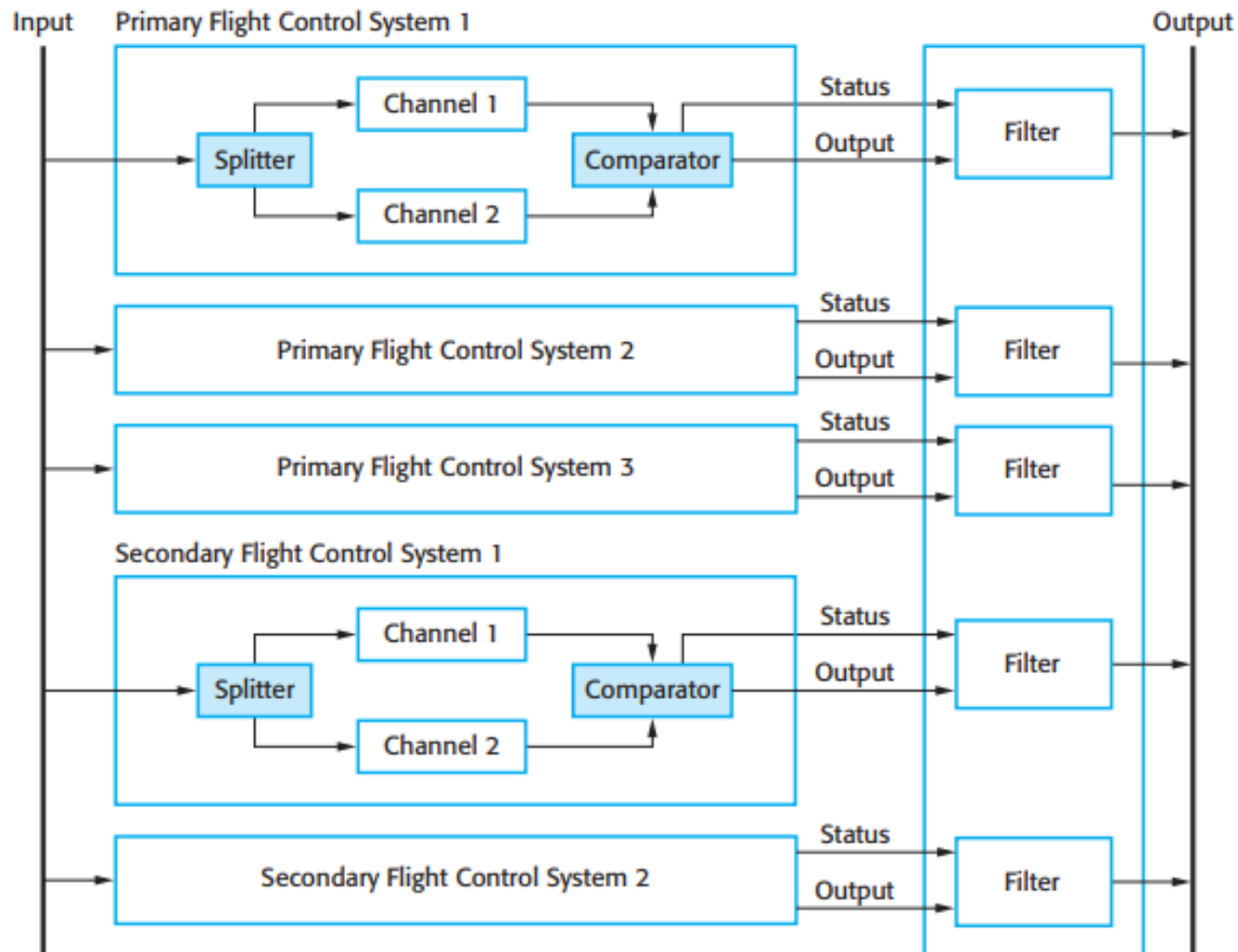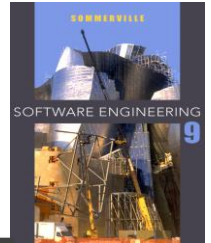
Airbus architecture discussion
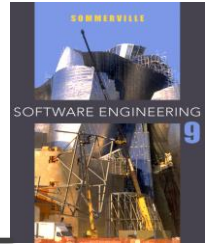
✧ The Airbus FCS has 5 separate computers, any one of which can run the control software.

✧ Extensive use has been made of diversity

- Primary systems use a different processor from the secondary systems.

- Primary and secondary systems use chipsets from different manufacturers.

- Software in secondary systems is less complex than in primary system – provides only critical functionality.

- Software in each channel is developed in different programming languages by different teams.

- Different programming languages used in primary and secondary systems.

Software diversity

✧ Approaches to software fault tolerance depend on software diversity where it is assumed that different implementations of the same software specification will fail in different ways.

✧ It is assumed that implementations are (a) independent and (b) do not include common errors.

✧ Strategies to achieve diversity

  ▪ Different programming languages

  ▪ Different design methods and tools

  ▪ Explicit specification of different algorithms

Problems with design diversity
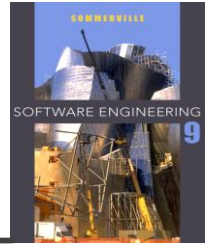
✧ Teams are not culturally diverse so they tend to tackle problems in the same way.

✧ Characteristic errors

- Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place;

- Specification errors;

- If there is an error in the specification then this is reflected in all implementations;

- This can be addressed to some extent by using multiple specification representations.

Good practice guidelines for dependable programming
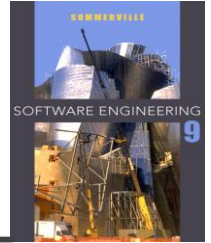
Dependable programming guidelines

1. Limit the visibility of information in a program
2. Check all inputs for validity
3. Provide a handler for all exceptions
4. Minimize the use of error-prone constructs
5. Provide restart capabilities
6. Check array bounds
7. Include timeouts when calling external components
8. Name all constants that represent real-world values

Control the visibility of information in a program

✧ Program components should only be allowed access to data that they need for their implementation.

✧ This means that accidental corruption of parts of the program state by these components is impossible.

✧ You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().

Check all inputs for validity

✧ All program take inputs from their environment and make assumptions about these inputs.

✧ However, program specifications rarely define what to do if an input is not consistent with these assumptions.

✧ Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.

✧ Consequently, you should always check inputs before processing against the assumptions made about these inputs.

Validity checks

✧ Range checks

- Check that the input falls within a known range.

✧ Size checks

- Check that the input does not exceed some maximum size e.g. 40 characters for a name.

✧ Representation checks

- Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.

✧ Reasonableness checks

- Use information about the input to check if it is reasonable rather than an extreme value.

Provide a handler for all exceptions

✧ A program exception is an error or some unexpected event such as a power failure.

✧ Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.

Exception handling

Code Section

Normal Flow of Control

Exception Detected

Normal Exit

Exception Processing

Exception Handling Code
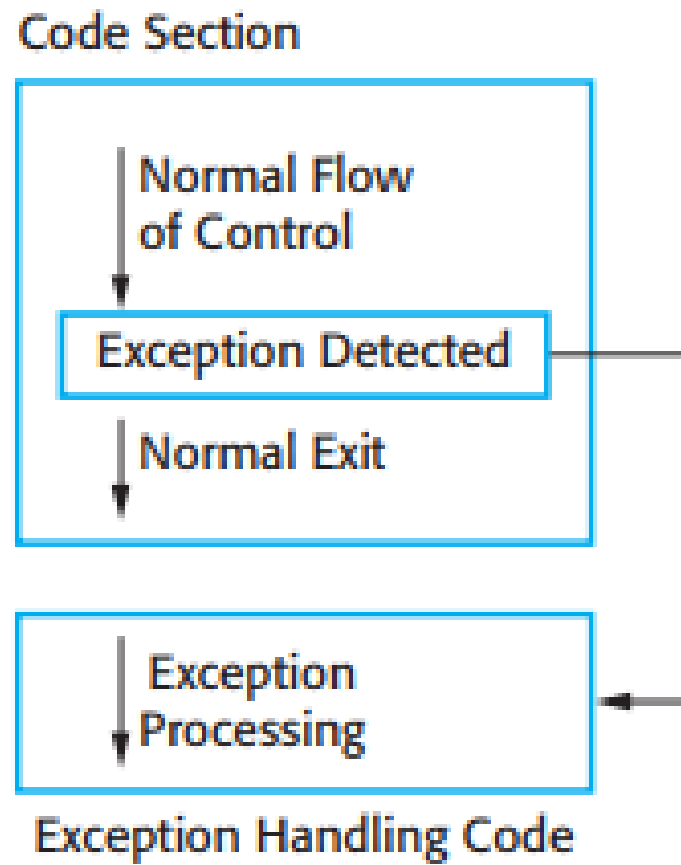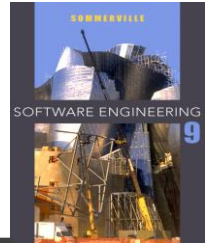
Exception handling

✧ Three possible exception handling strategies

  ▪ Signal to a calling component that an exception has occurred and provide information about the type of exception.

  ▪ Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.

  ▪ Pass control to a run-time support system to handle the exception.

✧ Exception handling is a mechanism to provide some fault tolerance

Minimize the use of error-prone constructs

✧ Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system

✧ This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don't check for mistakes when they could do so.

✧ Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.

Error-prone constructs
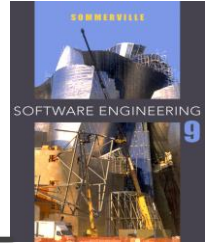
✧ Parallelism

- Can result in subtle timing errors because of unforeseen interaction between parallel processes.

✧ Recursion

- Errors in recursion can cause memory overflow as the program stack fills up.

✧ Interrupts

- Interrupts can cause a critical operation to be terminated and make a program difficult to understand.

✧ Inheritance

- Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.

Provide restart capabilities

✧ For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.

✧ Restart depends on the type of system

- Keep copies of forms so that users don't have to fill them in again if there is a problem

- Save state periodically and restart from the saved state

Check array bounds

✧ In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.

✧ This leads to the well-known 'bounded buffer' vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.

✧ If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

Include timeouts when calling external components

✧ In a distributed system, failure of a remote computer can be 'silent' so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.

✧ To avoid this, you should always include timeouts on all calls to external components.

✧ After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

Name all constants that represent real-world values

✧ Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name

✧ You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.

✧ This means that when these 'constants' change (for sure, they are not really constant), then you only have to make the change in one place in your program.

Key points
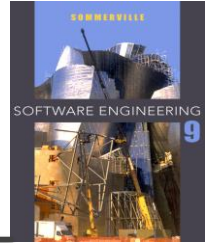
✧ Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.

✧ Dependable programming relies on the inclusion of redundancy in a program to check the validity of inputs and the values of program variables.

✧ Some programming constructs and techniques, such as goto statements, pointers, recursion, inheritance and floating-point numbers, are inherently error-prone. You should try to avoid these constructs when developing dependable systems.