Chapter 13 Notes

Dependability Engineering is concerned with the techniques that are used to enhance the dependability of both critical and non-critical systems. These techniques support 3 complementary approaches that are used in developing dependable software:

1. Fault Avoidance
   1. SW Design and Implementation process should use approaches to SW development that help avoid design and programming errors and so minimize the number of faults that are likely to arise when the system is executing
2. Fault Detection and Correction
   1. Verification and Validation processes are design to discover and remove faults in a program, before it is deployed for operational use.
3. Fault Tolerance
   1. The system is designed so that faults or unexpected system behavior during execution are detected at run-time and are managed in such a way that system failure does not occur. Simple approaches may include run-time checking

The cost of finding and removing the remaining faults in a software system rises exponentially. As a result, SW companies accept that they SW will always contain some residual faults.

Some companies believe that it is cheaper to let errors expose themselves than to try to find them

**Redundancy and Diversity (Section 13.1)**

* Redundancy and Diversity are fundamental strategies for enhancing the dependability of any type of system.
* Redundancy = that spare capacity is included in a system that can be used if part of that system fails
  + Example from life
    - Keeping spare light bulbs
    - Keeping many locks
* Diversity = that redundant components of the system are of different types, thus increasing the chances that they will not fail in exactly the same way
  + Example from lift
    - Have many different locks that require different keys to unlock
* Systems that include redundant components have these components provide the same functionality as other system components. These are switched into the system if the primary component fails. If these components are diverse, than a fault will not affect all of the diverse components.
* Diversity and Redundancy make system more complex and usually harder to understand.
  + Added functionality must also be added to the system to detect component failure and to switch control to alternative components
  + This introduces more bugs from programmers

**Dependable Processes (Section 13.2)**

* These are processes that are designed to produce dependable software.
* Usually 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:
  + Defined
    - A process that has a defined process model that is used to drive the software production process. There must be data collected during the process that demonstrates that all of the necessary steps in the process model have been enacted.
  + Repeatable
    - A process that does not rely on individual interpretation and judgment. It can be repeated across projects and with different team members.
* Dependable processes make use or redundancy and diversity to achieve reliability. They include other activities
  + 1) Requirements Reviews
    - Check that the requirements are, as far as possible, complete and consistent
  + 2) Requirements Management
    - 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
    - A mathematical model of the software is created and analyzed. It forces a very detailed analysis of the system requirements.
  + 4) System Modeling
    - SW Design is explicitly documented as a set of graphical models, and the links between the requirements and these models are explicitly documented.
  + 5) Design and Program Inspections
    - Different description of the system are inspected and checked by different people. Inspections are often driven by checklists of common design and programming errors
  + 6) Static Analysis
    - Automated checks are carried out on the source code of the program. The look for anomalies that could indicate programming errors or omissions.
  + 7) Test Planning and Management
    - 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.
  + Also there must be well-define Quality Management and Change Management Processes
    - Quality Management -> Creates a set of process and product standards and defines inspections
    - Change Management -> 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.
* Attributes of Dependable Processes
  + Documentable
    - Should have a process model that sets out the activities in the process and the documentation that is to be produced during these activities
  + Standardized
    - Standard covering SW production and documentation should be available
  + Auditable
    - Should be understandable by people apart from process participants who can check that process standards are being followed and make suggestion for process improvement
  + Diverse
    - Should include Redundant and Diverse verification and validation activities
  + Robust
    - Should be able to recover from failures of individual process activities

**Dependable System Architectures (Section 13.3)**

* You need to design system architecture for dependability, especially when fault tolerance is required. This means that the architecture has to be designed to include redundant components and mechanisms that allow control to be switched from one component to another.
* Simple Example of a dependable architecture
  + Replicated servers -> where 2 or more servers carry out the same task. Requests for processing are channeled through a server management component that routes each request to a particular server. In the vent of server failure, the faulty server is switched out of the system. Unprocessed requests are resubmitted to other servers for processing.
* **Protection Systems (Section 13.3.1)**
  + Is a specialized system that is associated with some other system. This is usually a control system for some process, such as a chemical manufacturing process.
  + These systems independently monitor their environment and, if the sensors indicate a problem that the controlled system is not dealing with, then the protection system is activated to shut down the process or equipment. These systems have 2 sets of sensors with one serving as a backup.
  + Protection systems only include the critical functionality that is required to move the system from a potentially unsafe state to a safe state (system shutdown).
  + Advantages
    - It can be much simpler then the software that is controlling the protected process. The only function of the protection system is to monitor operation and to ensure that the system is brought to a safe state in the event of an emergency.
* **Self-Monitoring Architectures (Section 13.3.2)**
  + Is a system architecture in which the system is designed to monitor its own operation and to take some action if a problem is detected
    - This is achieved by carrying out computations on separate channels and comparing the outputs of these computations. If the outputs are different, then a failure is assumed. When this occurs, the system will normally raise a failure exception on the status output line, which will lead to control being transferred to another system
  + To be effective in detecting hardware and software faults, these system have to be designed so that:
    - Hardware used in each channel is diverse. This reduces the probability of common processor design faults affecting the computations.
    - Software used in each channel is diverse. Otherwise, the same software error could arise at the same time on each channel
  + On its own, this architecture may be used in situation where it is important for computations to be correct, but where availability is not essential.
    - In situations where high availability is required, you have to use several self-checking systems in parallel. You need switching unit that detects faults and selects a result from one of the systems, where both channels are producing a consistent response.
* **N-version Programming (Section 13.3.3)**
  + Self-Monitoring systems are examples of systems in which multi-version programming is used to provide software redundancy and diversity
  + The notion of multi-version programming has been derivced from hardware systems where the notion of Triple Modular Redundancy (TMR) has been used for many years to build systems that tolerate hardware failures
  + TMR
    - Hardware unit is replicated 3 or more time.
    - Output from each unit is passed to an output comparator that is usually implemented as a voting system. This system compares all of its inputs and, if two or more units are the same, then that value is output. If one of the unites fails and does not produce the same output as the other unities, its output is ignored
    - A fault manager may try to repair the faulty unit automatically but if this is impossible the system is automatically reconfigured to take the unit out of service. The system then continues to function with 2 units
    - This approach to fault tolerance relies on most hardware failures being the result of component failure rather than design faults.
  + Another approach is to use N-diverse versions of a software system
    - Using a common specification, the same software system is implemented by a number of teams. These versions are executed on separate computers. Their outputs are compared using a voting system, and inconsistent outputs that are not produced in time are rejected. At least 3 versions of the system should be available so that 2 version should be consistent in the event of a single failure
* **Software Diversity (Section 13.3.4)**
  + All of the fault-tolerant architectures rely on SW diversity to achieve fault tolerance. This is based on the assumption that diverse implementations of the same specification are independent. This requires the software to be written by different teams who should not communicate during the development process.
  + The company may include explicit diversity policies that are intended to maximize the differences between the system versions
    - By including requirements that different design models should be used (one team uses object oriented design and another uses function-oriented design)
    - By stipulating that the implementations are to be written in different programming languages.
    - By requiring the use of different tools and development environments for the system
    - By explicitly requiring different algorithms to be used in some parts of the implementation
  + Each development team should work with a detailed system specification (called V-Spec ) that has been derived from the system requirements specification.
  + Reasons why achieving complete channel independence is impossible
    - 1) Members of different teams are often from the same cultural background and may have been educated using the same approach and textbooks
    - 2) If the requirements are incorrect or they are based on misunderstandings about the environment of the system, then these mistakes will be reflected in each implementation of the system
    - 3) In a critical system, the V-Spec is a detailed document based on the system’s requirements, which provides full details to the team on how the system behave. There cannot be scope for interpretation by the SW developers. If there are errors in this document, then these will be present to all of the development teams and implemented in all version of the system
  + Ways to reduce the possibility of common specification errors
    - Develop detailed specifications for the system independently, and to define the specifications in different languages (formal language, state-based language, natural language, etc.)

**Dependable Programming (Section 13.4)**

* Guidelines
  + 1) Control the visibility of Information in a Program
    - Program components should only be allowed access to data they need for their implementation…other data should be hidden and inaccessible from them to ensure they don’t get corrupted
  + 2) Check all Inputs for Validity
    - Do Range Checks
      * Make sure data falls within a valid range
    - Do Size Checks
      * Ex. Checking string is 8 characters long for bank account number
    - Do Representation Checks
      * Make sure people’s names don’t include numbers, emails are made up of 2 parts separated by a @ sign
    - Do Reasonableness Checks
      * Ensure data fits an expected pattern
      * Ex. Electric usage data for a month should be similar to data from the same month last year
  + 3) Provide a Handler for All Exception
    - Exceptions maybe caused by hardware or software conditions. By default, the system handles exceptions by reporting the error and exiting the system. Therefore, to ensure the system doesn’t fail, you should define an exception handler for all possible exceptions and make sure that all exceptions are detected and explicitly handled.
    - Handlers usually do one or more of the following:
      * Signal to a high-level component that an exception occurred and provide info to that component about the type of exception.
        + Use this when one component calls another component and need to know if the called component executed successfully
      * Carry out some alternative processing to that which was originally intended. Handler takes some actions to recover from the problem.
      * Pass control to a run-time support system that handles the execution. This is often the dealt when faults occur in a program. The usual action of this system is to halt processing.
  + 4) Minimize the Use of Error-Prone Constructs (Programming Language stuff)
    - 1) Unconditional Branch (GOTO) Statements
      * These lead to spaghetti code that is tangled and difficult to understand and debug
    - 2) Floating Point Numbers
      * This are inherently imprecise and comparisons may not be correct when they should be
      * Use fixed-length decimal values instead (e.g., 5 decimal places)
    - 3) Pointers
      * These can point to wrong memory
      * Makes bound checking harder to implement
    - 4) Dynamic Memory Allocation
      * Danger with this is that the memory may not by properly de-allocated and may result in a memory leak
    - 5) Parallelism
      * There may be subtle timing dependencies
    - 6) Recursion
      * May be concise but their logic can be difficult to follow
      * Makes debugging more difficult
      * Can waster the entire stack space
    - 7) Interrupts
      * These force control to transfer to a section of code irrespective of the code currently executing
      * The danger is that an interrupt may cause a critical operation to be terminated
    - 8) Inheritance
      * Problem is that code for an object is not all in the same place and thus, makes it more difficult to understand the object
      * Can result in binding run-time issues
    - 9) Aliasing
      * Occurs when one variable has more than one name to refer to it
      * Makes it easy for programmers to miss statements that change the entity when they have several names to consider
    - 10) Unbounded Arrays
      * Make accessing memory outside of an array possible
    - 11) Default input processing
      * This may accept invalid data
  + 5) Provide Restart Capabilities
    - You should provide the capability to keep copies of data that is collected or generated during processing and use these copies to restart the program instead of restarting from scratch. Thee copies are sometimes called checkpoints
    - If an exception occurs and it is impossible to continue normal operation, you can handle the exception using backward error recovery
  + 6) Check Array Bounds
    - Not checking array bounds leads to security vulnerabilities
  + 7) Include Timeouts when Calling External Components
    - If component A calls component B and B does NOT respond, then A cannot proceed indefinitely
      * To avoid this use timeouts…if time passes that exceeds the timeout then it is assumed that B failed so now A can continue
  + 8) Name all constants that Represent Real-World Values
    - Always include a section in your program in which you name all real-world constant values that are used
    - 2 Advantages
      * Less likely to make mistakes and use the wrong value
      * When a value changes, you do not have to look through the whole program to discover where you have used that value