Chapter 11 – Reliability Engineering

Topics covered

- ✧ Availability and reliability
- ✧ Reliability requirements
- ✧ Fault-tolerant architectures
- ✧ Programming for reliability
- ✧ Reliability measurement

Software reliability

- ✧ Dependable software is expected
- ✧ However, some system failures are accepted.
- ✧ Software systems have high reliability requirements
  - E.g., critical software systems
- ✧ Software engineering techniques for reliability requirements.
  - E.g., medical systems and aerospace systems

Faults, errors and failures

Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
System fault	A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.

Faults and failures

- ✧ Failures
  - Results of system errors resulted from faults in the system
- ✧ However, faults do not necessarily result in system errors
  - Transient and ‘corrected’ before an error arises.
  - Never be executed.
- ✧ Errors do not necessarily lead to system failures
  - Corrected by detection and recovery
  - Protected by protection facilities.

Fault management

- ✧ Fault avoidance
  - Avoid human errors to minimize system faults.
  - Organize development processes to detect and repair faults.
- ✧ Fault detection
  - Verification and validation techniques to remove faults.
- ✧ Fault tolerance
  - Design systems that faults do not cause failures.

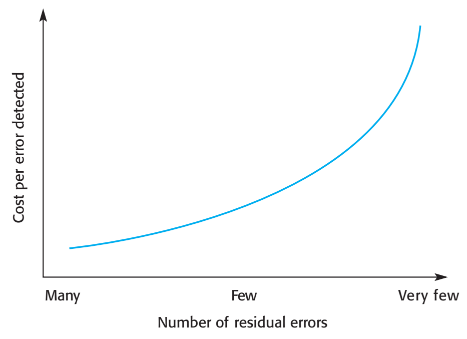
Reliability achievement

✧ Fault avoidance
- Development technique to minimise the possibility of mistakes or reveal mistakes.

✧ Fault detection and removal
- Verification and validation techniques to increase the probability of correcting errors.

✧ Fault tolerance
- Run-time techniques to ensure that faults do not cause errors.

The increasing costs of residual fault removal



Availability and reliability

Availability and reliability

✧ Reliability
- The probability of failure-free system operation.

✧ Availability
- The probability that a system conducts requested services at a point in time.

Reliability and specifications

✧ Reliability
- Defined formally w.r.t. a system specification
- A deviation from a specification.

✧ Incomplete or incorrect specifications
- A system following specifications may ‘fail’.

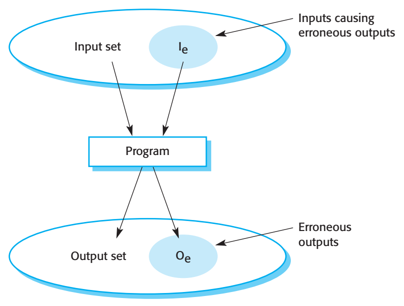
✧ Unfamiliar with specifications
- Unaware how the system is supposed to behave.

✧ Perceptions of reliability

Perceptions of reliability

✧ Not always reflect the user’s reliability perception
- The assumptions about environments for a system are incorrect
  - Different usage of a system between in an office environment and in a university environment.
- The consequences of system failures affects the perception of reliability.

A system as an input/output mapping

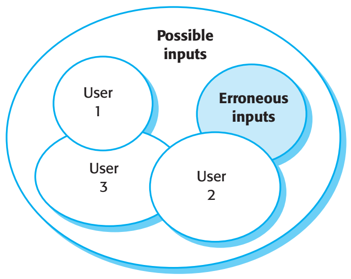


Availability perception

✧ Expressed as a percentage of the time
- Available to conduct services.

✧ Two factors not considered:
- The number of users affected by unavailable systems.
- The length of system failed or unavailable period.

Software usage patterns



Reliability in use

✧ Reliability not improved by X% by removing faults with X%

✧ Program defects rarely executed
- Not encountered by users.
- Not affect the perceived reliability.

✧ Users' operation patterns to avoid system features.

✧ Software systems with known faults
- Considered reliable systems by users.

Reliability requirements

System reliability requirements

✧ Functional reliability requirements
- Define system and software functions
- Avoid, detect or tolerate faults
- Not lead to system failure.

✧ Software reliability requirements
- Cope with hardware failure or operator error.

✧ Non-functional reliability requirements
  - ✧ A measurable system attribute specified quantitatively.
  - ✧ E.g., the number of failures and the available time.

Reliability metrics

- ✧ Units of measurement of system reliability.
- ✧ Counting the number of operational failures and the period length that the system has been operational.
- ✧ Assess the reliability (e.g., critical systems)
  - Long-term measurement techniques
- ✧ Metrics
  - Probability of failure on demand
  - Rate of occurrence of failures

Probability of failure on demand (POFOD)

- ✧ The probability of the system failure when a service request is made.
  - Useful when demands for service are relatively infrequent.
- ✧ Implement appropriate protection systems
  - Demand services occasionally.
  - Serious consequence due to failed services.
- ✧ Develop for safety-critical systems
  - E.g., emergency shutdown system in a chemical plant.

Rate of fault occurrence (ROCOF)

- ✧ System failure occurrence rate
- ✧ ROCOF of 0.002
  - 2 failures are likely in each 1000 operational time units
- ✧ Reliable systems needed
  - Systems perform a number of similar requests in a short time
  - E.g., credit card processing system.
- ✧ Reciprocal of ROCOF is Mean time to Failure (MTTF)
  - Systems with long transactions
  - System processing takes a long time. MTTF is longer than expected transaction length.

Availability

- ✧ The time that a software system is available
  - Repair and restart time considered
- ✧ Availability of 0.001
  - Software is available for 1 out of 1000 time units.
- ✧ Continuously running systems
  - E.g., railway signalling systems.

Availability specification

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1,440 minutes), the system will be unavailable for 144 minutes. (1440 ∗ 10%)
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period. Roughly, one minute per week.

Non-functional reliability requirements

- ✧ Non-functional reliability requirements
  - Reliability specifications using one of the reliability metrics (POFOD, ROCOF or AVAIL).
- ✧ Used for many years in safety-critical systems
  - Uncommon for business critical systems.
- ✧ Need precise measurement about reliability and availability expectations.

Benefits of reliability specification

- ✧ Help to clarify stakeholders' needs.
- ✧ Provide a measurement basis for system tests.
- ✧ Improve the reliability by different design strategies.
- ✧ Evidence of including required reliability implementations.

Specifying reliability requirements

- ✧ Availability and reliability requirements for different types of failure.
  - Low probability of high-cost failures
- ✧ Availability and reliability requirements for different types of system service.
  - Tolerate failures in less critical services.
- ✧ A high level of reliability required.
  - Other mechanisms for reliable system services.

ATM reliability specification

- ✧ Key concerns
  - ATMs conduct services as requested
  - Record customer transactions
  - ATM systems are available when required.
- ✧ Database transaction mechanisms make a correction of transaction problems

ATM availability specification

- ✧ System services
  - The customer account database service;
  - ‘withdraw cash’, ‘provide account information’, etc.
- ✧ Specify a high level of availability in database service.
  - Database availability: 0.9999, between 7 am and 11pm.
  - A downtime of less than 1 minute per week.

ATM availability specification

- ✧ Key reliability issues depends on mechanical reliability.
- ✧ A lower level of software availability is acceptable.
- ✧ The overall availability
  - Specify availability with 0.999
  - A machine might be unavailable for between 1 and 2 minutes each day.

Insulin pump reliability specification

- ✧ Probability of failure (POFOD) metric.
- ✧ Transient failures
  - Repaired by user actions, such as, recalibration of the machine.
  - Low POFOD is acceptable. A failure occurs in every 500 demands.
- ✧ Permanent failures
  - Re-installed by the manufacturer.
  - Occur no more than once per year.
  - POFOD < 0.00002.

Functional reliability requirements

- ✧ Checking requirements
  - Identify incorrect data before it leads to a failure.
- ✧ Recovery requirements
  - Help the system recover from a failure.
- ✧ Redundancy requirements
  - Specify redundant system features.
- ✧ Process requirements
  - Specify software development processes.

Examples of functional reliability requirements

RR1: A pre-defined range for all operator inputs shall be defined and the system shall check that all operator inputs fall within this pre-defined range. (Checking)

RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)

RR3: N-version programming shall be used to implement the braking control system. (Redundancy)

RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Fault-tolerant architectures

Fault tolerance

- ✧ Fault tolerant in critical situations.
- ✧ Fault tolerance required
  - High availability requirements
  - Failure costs are very high.
- ✧ Fault tolerance
  - Able to continue in operation despite software failures.
- ✧ Fault tolerant required against incorrect validation or specification errors, although a system is proved to conform to its specification

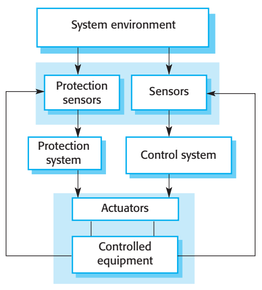
Fault-tolerant system architectures

- ✧ Fault-tolerant systems architectures
  - Fault tolerance is essential based on redundancy and diversity.
- ✧ Examples of situations for dependable architectures:
  - Flight control systems for safety of passengers
  - Reactor systems for a chemical or nuclear emergency
  - Telecommunication systemsfor 24/7 availability.

Protection systems

- ✧ A specialized system
  - Associated with other control system.
  - Take emergency action to deal with failures.
  - E.g., System to stop a train or system to shut down a reactor
- ✧ Monitor the controlled system and the environment.
- ✧ Take emergency action to shut down the system and avoid a catastrophe.

Protection system architecture



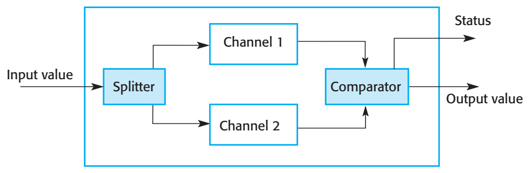
Protection system functionality

- ✧ Protection systems for redundancy
  - Control capabilities to replicate in the control software.
- ✧ Diverse protection systems
  - Different technology used in the control software.
- ✧ Need to expend in validation and dependability assurance.
- ✧ A low probability of failure for the protection system.

Self-monitoring architectures

- ✧ Multi-channel architectures
  - System monitoring its own operations
  - Take action if inconsistencies are discovered
- ✧ The same computation is carried out on each channel
  - Compare the results
  - Producing identical results assumes correct system operation
  - A failure exception is reported when different results arise.

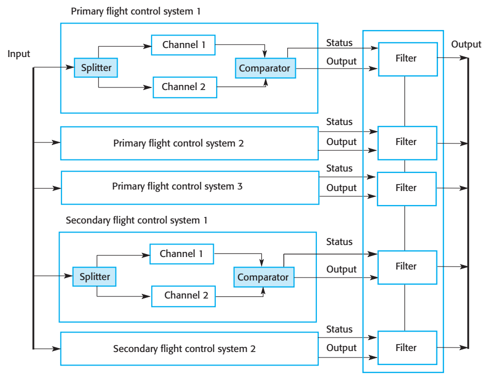
Self-monitoring architecture



Self-monitoring systems

- ✧ Diverse hardware systems on each channel
  - Prevent common failures producing the same results.
- ✧ Diverse software applications in each channel
  - Prevent same errors affecting each channel.
- ✧ Several self-checking systems in parallel.
  - High-availability required.
  - E.g., Airbus aircraft's flight control systems.

Airbus flight control system architecture


Primary flight control system 1
Input
Splitter
Channel 1
Channel 2
Comparator
Status
Output
Filter
Output
Primary flight control system 2
Primary flight control system 3
Secondary flight control system 1
Secondary flight control system 2

Airbus architecture discussion

✧ The Airbus FCS has 5 separate computers
- Each system is able to perform the control software.

✧ Extensive diverse systems between primary and secondary systems:
- Different processors
- Different chipsets from different manufacturers.
- Different complexity -- only critical functionality in secondary.
- Different programming languages by different teams.

N-version programming

✧ Multiple versions of a program execute computations.
- Odd number of computers involved, e.g., three versions.

✧ The results are compared using a voting system.

✧ The correct result is determine by the majority result.

✧ The notion of triple-modular redundancy, as used in hardware systems.

Hardware fault tolerance

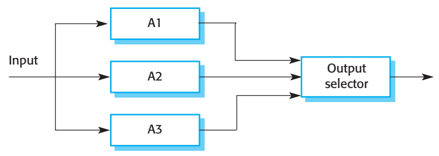
✧ Triple-modular redundancy (TMR).

✧ Three replicated identical components
- Receive the same input and their outputs are compared.

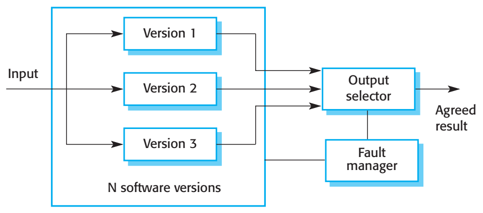
✧ One different output
- Ignored based on the assumption of component failure.

✧ Most faults caused by component failures
- A low probability of simultaneous component failure.

Triple modular redundancy (TMR)



N-version programming



N-version programming

✧ The different versions of a system
- Designed and implemented by different teams.

✧ Assuming a low probability of making same mistakes
- Different algorithms used

✧ Empirical evidence
- Commonly misinterpret specifications
- Use same algorithms in different systems.

Software diversity

- ✧ Fault tolerance depend on software diversity
- ✧ Assume that different implementations fail differently
  - Independent implementations
  - Uncommon errors
- ✧ Strategies
  - Different programming languages
  - Different design methods and tools
  - Explicit specification of different algorithms

Problems with design diversity

- ✧ Tend to solve problems using same methods
- ✧ Characteristic errors
  - Different teams making same mistakes. Making mistakes in same parts.
  - Specification errors propagated to all implementations

Specification dependency

- ✧ Software redundancy susceptible to specification errors.
  - Incorrect specification causes system failures.
- ✧ Complex software specifications
  - Hard to perform validation and verification.
- ✧ Developing separate software specifications.

Improvements in practice

- ✧ Multi-version programming leads to significant improvements in reliability and availability.
  - Diversity and independence
- ✧ In practice, observed improvements are much less significant
  - Reliability improvements of between 5 and 9 times.
- ✧ Considerable costs to develop multi-versions of systems.

Programming for reliability

Dependable programming

- ✧ Standard programming practices
  - Reduce program fault introduction rate.
- ✧ Support fault avoidance, detection and tolerance

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

```
ProcessBuilder pb =
 new ProcessBuilder("external-program");
Timer t = new Timer();
Process p = pb.start();

TimerTask killer =
  new TimeoutProcessKiller(p);
t.schedule(killer, 5000);
```

```
class TimeoutProcessKiller extends TimerTask {
  Process p;
  public TimeoutProcessKiller(Process p) {
    this.p = p;
  }
  @Override
  public void run() { p.destroy(); }
}
```

(1) Limit the visibility of information in a program

- ✧ Limited access to data for their implementation.
- ✧ Reduce possibilities of accidental corruption of program state by other components
- ✧ Control visibility by using abstract data types
  - Private data representation.
  - Limited access to data through predefined operations.

(2) Check all inputs for validity

- ✧ Taking inputs from their environment based on assumptions about the inputs.
- ✧ Program specifications rarely defined
  - Inconsistent inputs with the assumptions.
- ✧ Unpredictable program behavior
  - Unusual inputs
  - Threats to the security of the system.
- ✧ Check program inputs before processing
  - Considering the assumptions about the inputs.

Validity checks

- ✧ Range checks
  - Check the input's range.
- ✧ Size checks
  - Check the input's maximum or minimum size.
- ✧ Representation checks
  - Check the input's expression for the representation
  - E.g. names do not include numerals.
- ✧ Reasonableness checks
  - Check the input's logical information.
  - E.g., reasonable rather than an extreme value.

Example

```
  try {
    int a[]= {1, 2, 3, 4};
    for (int i = 1; i <= SIZE; i++) {
      System.out.println ("a[" + i + "]=" + a[i] + "\n");
    }
  }
  catch (Exception e) {
    System.out.println ("error = " + e);
  }
  catch (ArrayIndexOutOfBoundsException e) {
    System.out.println ("ArrayIndexOutOfBoundsException");
  }
```

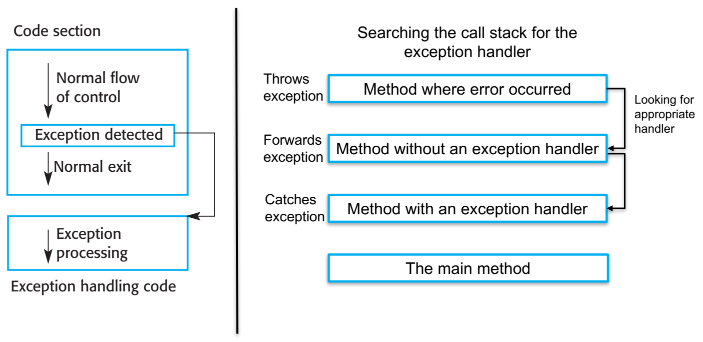
```
A.java: error: exception ArrayIndexOutOfBoundsException has already been caught
           catch (ArrayIndexOutOfBoundsException e)

^ 1 error
```

(3) Provide a handler for all exceptions

- ✧ An error or some unexpected event
  - E.g., a power failure.
- ✧ Exception handling constructs
  - Responding and handling exception events
  - Change the program execution flow
- ✧ Using normal control constructs to handle exceptions?
  - A number of additional statements
  - Significant overhead
  - Tedious and error-prone

Exception handling



Code section
Normal flow of control
Exception detected
Normal exit
Exception processing
Exception handling code
Searching the call stack for the exception handler
Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Catches exception
Method with an exception handler
The main method

Exception handling

✧ Three possible exception handling strategies
- Report exceptions and provide the related information.
- Conduct alternative processes
  - The related information required to recover from the problem.
- Pass control to a run-time support system.

✧ A mechanism to provide fault tolerance
- Recovering from fault-caused errors
- Eliminating faults

(4) Minimize the use of error-prone constructs

✧ Human error of misunderstanding or losing track of the relationships between the different parts of the system

✧ Error-prone constructs in programming languages
- Inherently complex

✧ Avoid or minimize the use of error-prone constructs.

Error-prone constructs

✧ Unconditional branch (goto) statements

✧ Floating-point numbers
- Inherently imprecise. The imprecision may lead to invalid comparisons.
- E.g., 7.00000000 (6.99999999 or 7.00000001)

✧ Pointers
- Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.

✧ Dynamic memory allocation
- Run-time allocation can cause memory overflow.

Common Mistakes in Dynamic Memory Allocation

✧ No matter how much we try, it is very difficult to free all dynamically allocated memory. Even if we can do that, it is often not safe from exceptions.

✧ If an exception is thrown, the "a" object is never deleted.

```
void SomeMethod() {
  ClassA *a = new ClassA;

  // it can throw an exception
  foo();

  delete a;
}
```

✧ Detect memory leaks by Valgrind

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.

Common Mistake in Inheritance and Dynamic Binding

```
abstract class Base {
   int f = 0;
   public int m1() {
      return 0;
   }
}

class ClassA extends Base {
   int f = 1;

   public int m1() {
      return f;
   }
}

class ClassB extends Base {
   public int m1() {
      return f;
   }
}

class ClassC extends ClassB {
   int f = 2;

   public int m1() {
      return f;
   }
}
```

```
class Main {
   Base x1;

   void thread1() {
      x1 = new ClassA();
      System.out.println(x1.m1());
   }

   void thread2() {
      x1 = new ClassB();
      System.out.println(x1.m1());
   }

   void thread3() {
      x1 = new ClassC();
      System.out.println(x1.m1());
   }
}
```

Inheritance when combined with dynamic binding can cause timing problems at run-time.

Error-prone constructs

✧ Aliasing
- Using more than 1 name to refer to the same state variable.

✧ Unbounded arrays
- Buffer overflow failures can occur if no bound checking on arrays.

✧ Default input processing
- Occur irrespective of the input.
- The default action changes the program control flow.
- Malicious inputs trigger a program failure.

Common Errors in Default Input Processing

✧ Unexpectedly execute a call to method foo(int) with default input '0', instead of a call to method foo(int, int).
- Method overloading.
- Integer '0' is a default value for argument 'b' in the overloaded method foo(int).

```
void foo(int a, int b) {
   // the implementation goes here
}

void foo(int a) {
   foo(a, 0);
}
```

(5) Provide restart capabilities

✧ The system restart capability
- Preserve the program data or status.
- Long transactions or user interactions.

✧ Restart scenarios
- Web applications keeping copies of forms that users fill in before there is a problem
- Text editors restarting from the checkpoint saving periodically the program state or memory

(6) Check array bounds

✧ Address a memory location outside of the range of an array declaration.

✧ Bounded buffer vulnerability
- E.g., Writing executable code into memory by deliberately writing beyond the top element in an array.

✧ Bound checking for an array access
- Within the bounds of the array.

Common Errors in Bounded Buffer Vulnerability

✧ Writing data past the end of allocated memory can be detected by OS
- Generate a segmentation fault error that terminates the process.

```
char str[8] = "";              // 8-byte-long string buffer
unsigned short year = 2017;  // two-byte integer
```

var	str								year
value	[null string]								2017

```
strcpy(str, "boundless");    // boundless" 9 characters long
```

var	str								year
value	'b'	'o'	'u'	'n'	'd'	'l'	'e'	's'	????

Common Errors in Bounded Buffer Vulnerability

✧ Return a heap-allocated copy of the string with all uppercase letters.

✧ No bounds checking on its input.

✧ Overflow buf with the unbounded call to strcpy().

```
char *lccopy(const char *str) {
  char buf[BUFSIZE];
  char *p;

  strcpy(buf, str);
  for (p = buf; *p; p++) {
    if (isupper(*p)) {
      *p = tolower(*p);
    }
  }
  return strdup(buf);
}
```

(7) Include timeouts when calling external components

✧ May never receive services from failed systems
- No indication of a failure.
- Failure of a remote computer can be 'silent' In a distributed system.

✧ Set timeouts on all calls to external components.

✧ Assume failure and take actions to recover from errors
- After a defined time period without a response.

(8) Name all constants that represent real-world values

✧ Use constants reflecting real-world values names
- Do not use numeric values and always refer to them by name.

✧ Reduce mistakes for wrong values by using a name rather than a value.

✧ Changing constant values
- Easy to maintain and localize edit locations to make the change.

Constant Interface Pattern

✧ Use final class for Constants

✧ Declare public static final and static import all constants

```
public final class Constants {
   private Constants() {
      // restrict instantiation
   }

   public static final double PI               = 3.14159;
   public static final double PLANCK_CONSTANT = 6.62606896e-34;
}
```

```
import static math.Constants.PLANCK_CONSTANT;
import static math.Constants.PI;

class Calculations {
   public double getReducedPlanckConstant() {
      return PLANCK_CONSTANT / (2 * PI);
   }
}
```

Reliability measurement

Reliability measurement

✧ Collect data about system operation:

✧ The number of failures for system service requests
- Probability of failure on demand (POFOD)

✧ The time or the number of transactions between system failures plus the total elapsed time or total number of transactions
- Mean time to failure (MTTF)
- Rate of occurrence of fault (ROCOF)

✧ The repair or restart time after a system failure
- Availability
- The time between failures
- The time required to restore the system from failures

Reliability testing

✧ Reliability testing (Statistical testing)
  ▪ Assess whether a system reaches the required level of reliability.

✧ Not part of a defect testing process
  ▪ Datasets for defect testing do not include actual usage data.

✧ Data sets required to replicate actual inputs to be processed.

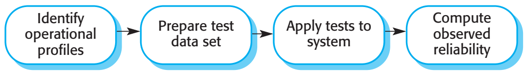
Statistical testing

✧ Testing software for reliability rather than fault detection.

✧ Measuring the number of errors
  ▪ Predict reliability of the software
  ▪ More errors, compared to the one in the specification.

✧ An acceptable level of reliability
  ▪ Test software systems and repair or improve them until systems reach the level of reliability.

Reliability measurement



Reliability measurement problems

✧ Operational profile uncertainty
  ▪ Inaccurate operational profile that does not reflect the real use of the system.

✧ High costs of test data generation
  ▪ Expensive costs to generate test datasets for the system.

✧ Statistical uncertainty
  ▪ A statistically significant number of failures for computation.
  ▪ Highly reliable systems rarely fail.

✧ Recognizing failure
  ▪ Conflicting interpretations of a specification about unobvious failures.

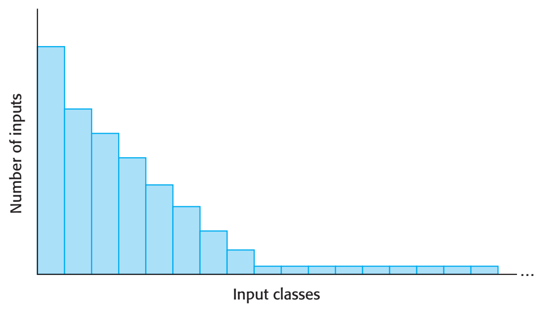
Operational profiles

✧ A set of test data
  ▪ Frequency matches the actual frequency from 'normal' usage of the system.
  ▪ The number of times the failure event occurred.

✧ A close match with actual usage
  ▪ The measured reliability
  ▪ Reflect the actual usage of the system.

✧ Generate from real data
  ▪ Collect from an existing system
  ▪ Assumption of the usage pattern of a system

An operational profile



Operational profile generation

- ✧ Automatic data generation, if possible.
  - Difficult for interactive systems.
  - Easy for 'normal' inputs
- ✧ Difficult to generate 'unlikely' inputs (anomalies) and test data for these anomalies.
- ✧ Unknown usage pattern of new systems.
- ✧ Changeable operational profiles
  - Non-static but dynamic
  - E.g., learn about new systems, changing usage patterns.

Key points

- ✧ Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment and by including fault tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- ✧ Reliability requirements can be defined quantitatively in the system requirements specification.
- ✧ Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF) and availability (AVAIL).

Key points

- ✧ Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.
- ✧ Dependable system architectures are system architectures that are designed for fault tolerance.
- ✧ There are a number of architectural styles that support fault tolerance including protection systems, self-monitoring architectures and N-version programming.

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 including redundancy in a program as checks on the validity of inputs and the values of program variables.
- ✧ Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.