

# **CS 501: Software Engineering**

## **Fall 2000**



### **Lecture 21**

Dependable Systems I  
Reliability

# Administration



## Assignment 3

- Report due tomorrow at 5 p.m.  
*group design with individual parts*
- Presentations Wednesday through Friday  
*every group member must present during the semester*

# Software Reliability



Failure: Software does not deliver the **service expected by the user** (e.g., mistake in requirements)

Fault: Programming or design error whereby the delivered system does not **conform to specification**

Reliability: Probability of a failure occurring in operational use.

Perceived reliability: Depends upon:

- user behavior

- set of inputs

- pain of failure

# Reliability Metrics



- Probability of failure on demand
- Rate of failure occurrence (failure intensity)
- Mean time between failures
- Availability (up time)
- Mean time to repair
- Distribution of failures

Hypothetical example: *Cars are safer than airplane in accidents (failures) per hour, but less safe in failures per mile.*

# Reliability Metrics for Distributed Systems



*Traditional metrics are hard to apply in multi-component systems:*

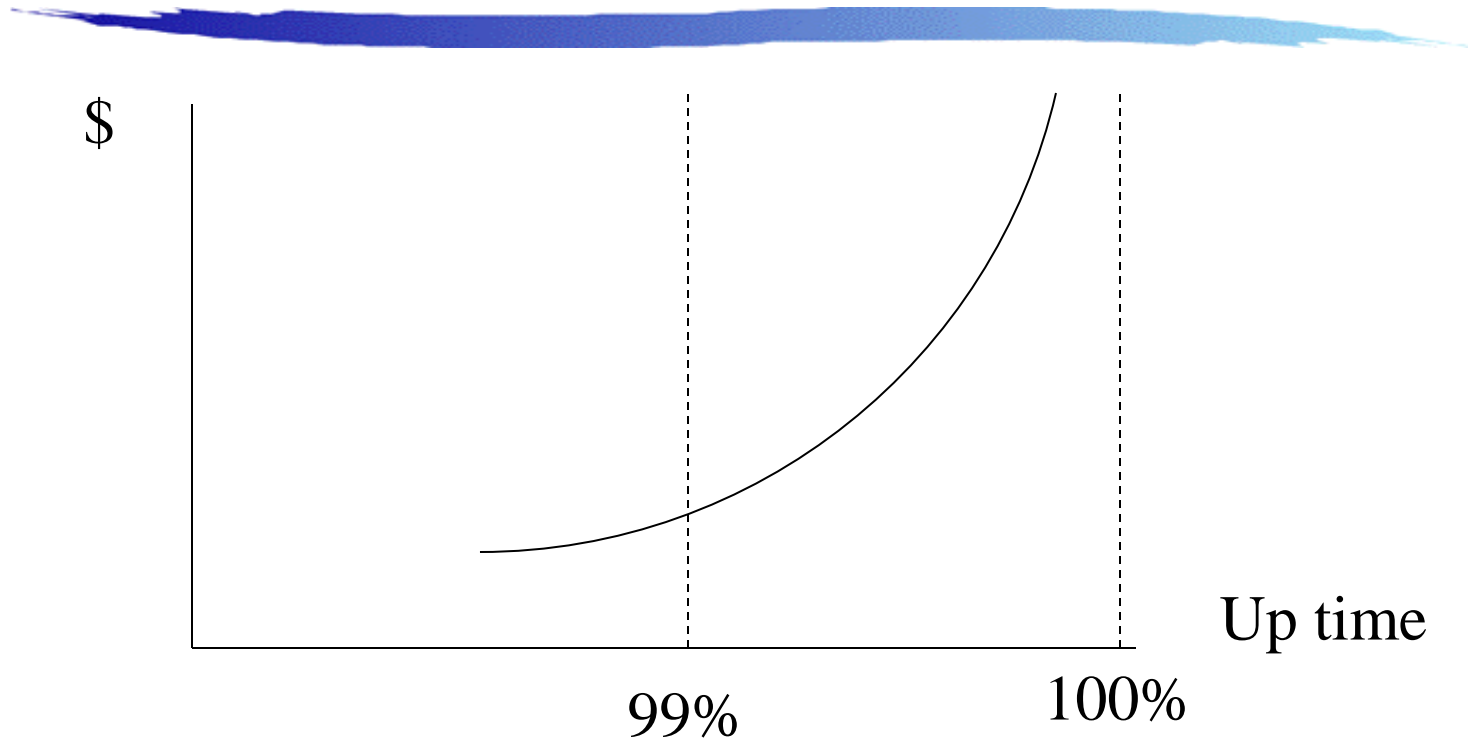
- In a big network, at a given moment something will be giving trouble, but very few users will see it.
- A system that has excellent average reliability may give terrible service to certain users.
- There are so many components that system administrators rely on automatic reporting systems to identify problem areas.

# User Perception of Reliability



1. A personal computer that crashes frequently v. a machine that is out of service for two days.
2. A database system that crashes frequently but comes back quickly with no loss of data v. a system that fails once in three years but data has to be restored from backup.
3. A system that does not fail but has unpredictable periods when it runs very slowly.

# Cost of Improved Reliability



Will you spend your money on new functionality or improved reliability?

# Specification of System Reliability



Example: ATM card reader

Failure class	Example	Metric
Permanent non-corrupting	System fails to operate with any card -- reboot	1 per 1,000 days
Transient non-corrupting	System can not read an undamaged card	1 in 1,000 transactions
Corrupting	A pattern of transactions corrupts database	Never

# Principles for Dependable Systems



The human mind can encompass only limited complexity:

- => Comprehensibility
- => Simplicity
- => Partitioning of complexity

# Principles for Dependable Systems



High-quality has to be built-in

- => Each stage of development must be done well
- => Testing and correction does not lead to quality
- => Changes should be incorporated into the structure

# Quality Management Processes



## Assumption:

Good processes lead to good software

## The importance of routine:

Standard terminology (*requirements, specification, design*, etc.)

Software standards (naming conventions, etc.)

Internal and external documentation

Reporting procedures

# Quality Management Processes



## Change management:

Source code management and version control

Tracking of change requests and bug reports

Procedures for changing requirements specifications, designs and other documentation

Release control

# Design and Code Reviews



- Colleagues review each other's work:
  - can be applied to any stage of software development
  - can be formal or informal
- The developer provides colleagues with:
  - documentation (e.g., specification or design), or code listing
  - talks through the work while answering questions
- Most effective when developer and reviewers prepare well

# Benefits of Design and Code Reviews



## Benefits:

- Extra eyes spot mistakes, suggest improvements
- Colleagues share expertise; helps with training
- An occasion to tidy loose ends
- Incompatibilities between modules can be identified
- Helps scheduling and management control

## Fundamental requirements:

- Senior team members must show leadership
- Must be helpful, not threatening

# Process (Plan) Reviews



## Objectives:

- To review progress against plan (formal or informal)
- To adjust plan (schedule, team assignments, functionality, etc.)

## Impact on quality:

Good quality systems usually result from plans that are demanding but realistic

*Good people like to be stretched and to work hard, but must not be pressed beyond their capabilities.*

# Statistical Testing



- Determine the operational profile of the software
- Select or generate a profile of test data
- Apply test data to system, record failure patterns
- Compute statistical values of metrics under test conditions

# Statistical Testing



## Advantages:

- Can test with very large numbers of transactions
- Can test with extreme cases (high loads, restarts, disruptions)
- Can repeat after system modifications

## Disadvantages:

- Uncertainty in operational profile (unlikely inputs)
- Expensive
- Can never prove high reliability

# Example: Dartmouth Time Sharing (1980)



*A central computer serves the entire campus. Any failure is serious.*

## Step 1. Gather data on every failure

- 10 years of data in a simple data base
- Every failure analyzed:
  - hardware
  - software (default)
  - environment (e.g., power, air conditioning)
  - human (e.g., operator error)

# Example: Dartmouth Time Sharing (1980)



## Step 2. Analyze the data.

- Weekly, monthly, and annual statistics
  - Number of failures and interruptions
  - Mean time to repair
- Graphs of trends by component, e.g.,
  - Failure rates of disk drives
  - Hardware failures after power failures
  - Crashes caused by software bugs in each module

# Example: Dartmouth Time Sharing (1980)



Step 3. Invest resources where benefit will be maximum, e.g.,

- Orderly shut down after power failure
- Priority order for software improvements
- Changed procedures for operators
- Replacement hardware

# Factors for Fault Free Software



- Precise, unambiguous specification
- Organization culture that expects quality
- Approach to software design and implementation that hides complexity (e.g., structured design, object-oriented programming)
- Use of software tools that restrict or detect errors (e.g., strongly typed languages, source control systems, debuggers)
- Programming style that emphasizes simplicity, readability, and avoidance of dangerous constructs
- Incremental validation

# Error Avoidance



## Risky programming constructs

- Pointers
- Dynamic memory allocation
- Floating-point numbers
- Parallelism
- Recursion
- Interrupts

*All are valuable in certain circumstances, but should be used with discretion*

# Defensive Programming



*Murphy's Law:* If anything can go wrong, it will.

## *Defensive Programming:*

- Redundant code is incorporated to check system state after modifications
- Implicit assumptions are tested explicitly

# Defensive Programming Examples



- Use *boolean* variable not *integer*
- Test  $i \leq n$  not  $i == n$
- Assertion checking
- Build debugging code into program with a switch to display values at interfaces
- Error checking codes in data, e.g., checksum or hash

# Some Notable Bugs



- Built-in function in Fortran compiler ( $e^0 = 0$ )
- Japanese microcode for Honeywell DPS virtual memory
- The microfilm plotter with the missing byte (1:1023)
- The Sun 3 page fault that IBM paid to fix
- Left handed rotation in the graphics package

*Good people work around problems.*

*The best people track them down and fix them!*