Cornell University
Computing and Information Science

CS 5150 Software Engineering

Reliability

William Y. Arms

Failures and Faults

Fault (bug):

Programming or design error whereby the delivered system does not conform to specification (e.g., coding error, protocol error)

Failure:

Software does not deliver the service expected by the user (e.g., mistake in requirements, confusing user interface)

Failure of Requirements

An actual example

The head of an organization is not paid his salary because it is greater than the maximum allowed by the program. (Requirements problem.)

Bugs and Features

That's not a bug. That's a feature!

Users will often report that a program behaves in a manner that they consider wrong, even though it is behaving as intended.

That’s not a bug. That’s a failure!

The decision whether this needs to be changed should be made by the client not by the developers.

Terminology

Fault avoidance

Build systems with the objective of creating fault-free (bug-free) software.

Fault detection (testing and verification)

Detect faults (bugs) before the system is put into operation or when discovered after release.

Fault tolerance

Build systems that continue to operate when problems (bugs, overloads, bad data, etc.) occur.

Dependable and Reliable Systems: A Case Study

A passenger ship with 1,509 persons on board grounded on a shoal near Nantucket Island, Massachusetts. At the time the vessel was about 17 miles from where the officers thought they were. The vessel was en route from Bermuda, to Boston.

Case Study: Analysis

From the report of the National Transportation Safety Board:

- The ship was steered by an autopilot that relied on position information from the Global Positioning System (GPS).
- If the GPS could not obtain a position from satellites, it provided an estimated position based on Dead Reckoning (distance and direction traveled from a known point).
- The GPS failed one hour after leaving Bermuda.
- The crew failed to see the warning message on the display (or to check the instruments).
- 34 hours and 600 miles later, the Dead Reckoning error was 17 miles.

Case Study: Software Lessons

All the software worked as specified (no bugs), but ...

- After the GPS software was specified, the **requirements** changed (stand alone system now part of integrated system).
- The manufacturers of the autopilot and GPS adopted different **design** philosophies about the communication of mode changes.
- The autopilot was not **programmed** to recognize valid/invalid status bits in messages from the GPS.
- The warnings provided by the **user interface** were not sufficiently conspicuous to alert the crew.
- The officers had not been properly **trained** on this equipment.

Reliable software needs all parts of the software development process to be carried out well.

Key Factors for Reliable Software

- Organization **culture** that expects quality. This comes from the management and the senior technical staff.
- Precise, unambiguous agreement on **requirements**.
- Design and implementation that **hides complexity** (e.g., structured design, object-oriented programming).
- **Programming style** that emphasizes simplicity, readability, and avoidance of dangerous constructs.
- **Software tools** that restrict or detect errors (e.g., strongly typed languages, source control systems, debuggers).
- Systematic **verification** at all stages of development, including requirements, system architecture, program design, implementation, and user testing.
- Particular attention to **changes** and maintenance.

Building Reliable Software: Organizational Culture

Good organizations create good systems:

- Managers and senior technical staff must lead by example.
- Acceptance of the group's style of work (e.g., meetings, preparation, support for juniors).
- Visibility.
- Completion of a task before moving to the next (e.g., documentation, comments in code).

Example: A library consortium

Building Reliable Software: Quality Management Processes

Assumption:

Good software is impossible without good processes

The importance of routine:

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

Software standards (*coding standards, naming conventions, etc.*)

Regular builds of complete system (*often daily*)

Internal and external documentation

Reporting procedures

This routine is important for both heavyweight and lightweight development processes.

Building Reliable Software: Quality Management Processes

When time is short...

Pay extra attention to the early stages of the process: feasibility, requirements, design.

If mistakes are made in the requirements process, there will be little time to fix them later.

Experience shows that taking extra time on the early stages will usually reduce the total time to release.

Building Reliable Software: Communication with the Client

A system is no use if it does not meet the client's needs

- The client must understand and review the agreed requirements in detail.
- It is not sufficient to present the client with a specification document and ask him/her to sign off.
- Appropriate members of the client's staff must review relevant areas of the design (including operations, training materials, system administration).
- The acceptance tests must belong to the client.

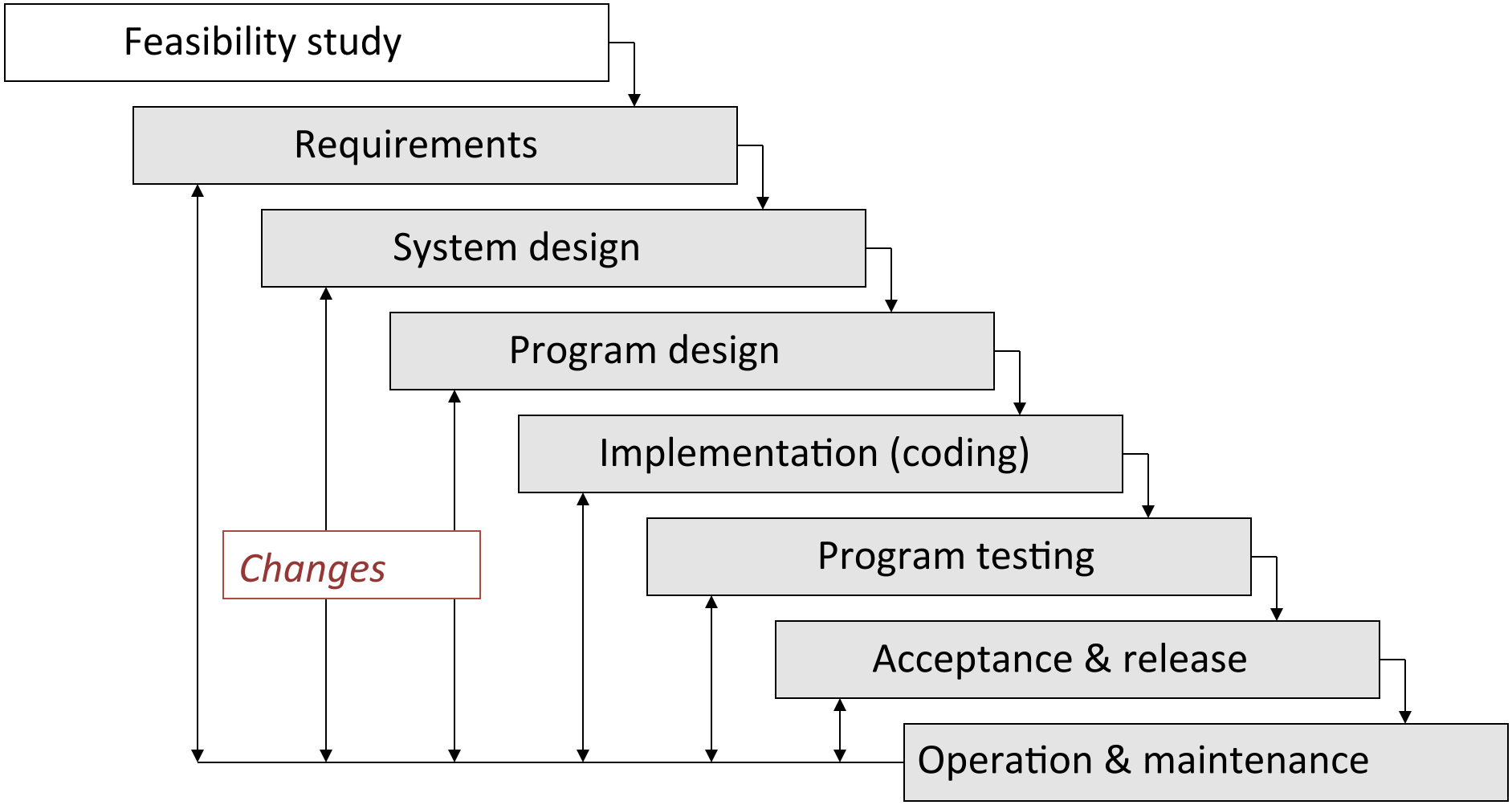
Building Reliable Software: Complexity

The human mind can encompass only limited complexity:

- Comprehensibility
- Simplicity
- Partitioning of complexity

A simple component is easier to get right than a complex one.

Building Reliable Software: Changes



Building Reliable Software: Change

Changes can easily introduce problems

Change management

- Source code management and version control
- Tracking of change requests and bug reports
- Procedures for changing requirements specifications, designs and other documentation
- Regression testing (discussed later)
- Release control

When adding new functions or fixing bugs it is easy to write patches that violate the systems architecture or overall program design. This should be avoided as much as possible. Be prepared to modify the architecture to keep a high quality system.

Building Reliable Software: Fault Tolerance

Aim:

A system that continues to operate when problems occur.

Examples:

- Invalid input data (e.g., in a data processing application)
- Overload (e.g., in a networked system)
- Hardware failure (e.g., in a control system)

General Approach:

- Failure detection
- Damage assessment
- Fault recovery
- Fault repair

Fault Tolerance: Recovery

Backward recovery

- Record system state at specific events (**checkpoints**). After failure, recreate state at last checkpoint.
- Combine checkpoints with system **log** (**audit trail** of transactions) that allows transactions from last checkpoint to be repeated automatically.

Recovery software is difficult to test

Example

After an entire network is hit by lightning, the restart crashes because of overload. (Problem of incremental growth.)

Building Reliable Software: Adapting Small Teams to Large Projects

Small teams and **small projects** have advantages for reliability:

- Small group communication cuts need for intermediate documentation, yet reduces misunderstanding.
- Small projects are easier to test and make reliable.
- Small projects have shorter development cycles. Mistakes in requirements are less likely and less expensive to fix.
- When one project is completed it is easier to plan for the next.

Improved reliability is one of the reasons that incremental development (e.g., Agile) has become popular over the past few years.

Reliability Metrics

Reliability

Probability of a failure occurring in operational use.

Traditional measures for online systems

- Mean time between failures
- Availability (up time)
- Mean time to repair

Market measures

- Complaints
- Customer retention

Reliability Metrics for Distributed Systems

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

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

Metrics: User Perception of Reliability

Perceived reliability depends upon:

- user behavior
- set of inputs
- pain of failure

User perception is influenced by the distribution of failures

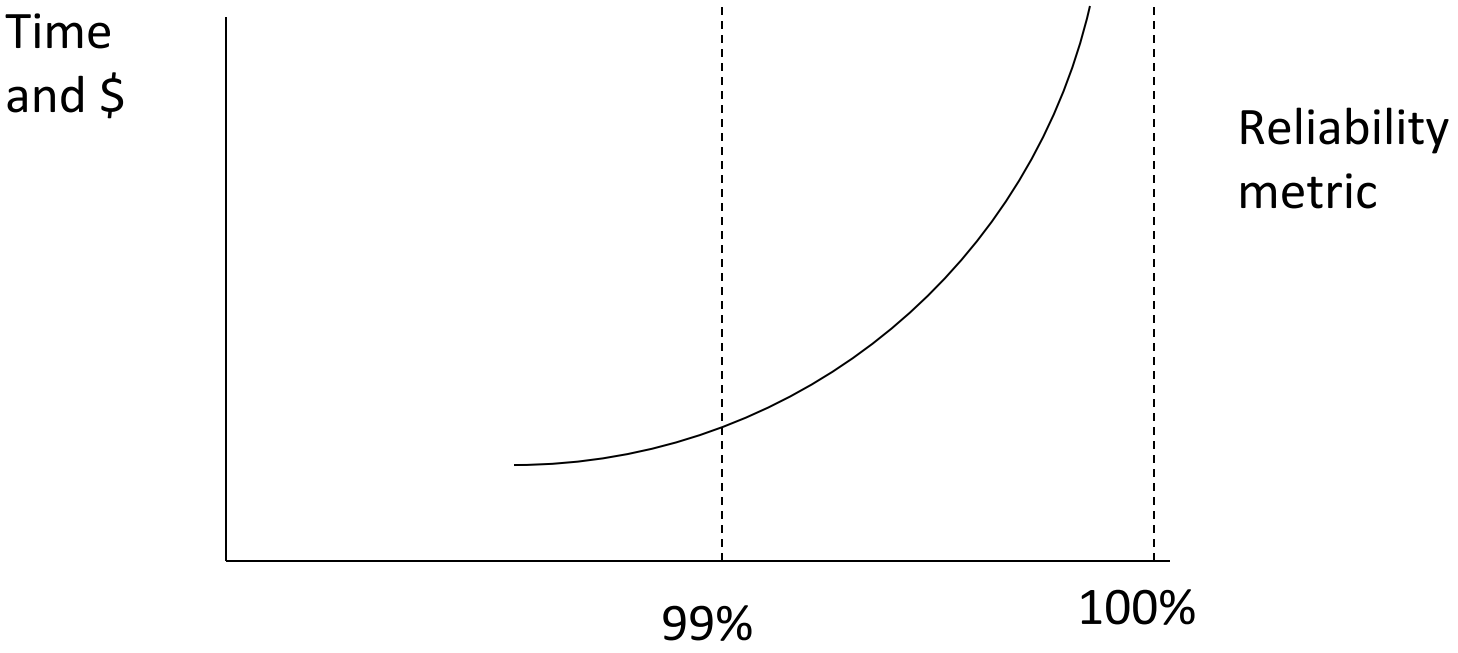
- A personal computer that crashes frequently, or a machine that is out of service for two days every few years
- A database system that crashes frequently but comes back quickly with no loss of data, or a system that fails once in three years but data has to be restored from backup.
- A system that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

Reliability Metrics for Requirements

Example: ATM card reader

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

Metrics: Cost of Improved Reliability



Example.

Many supercomputers average 10 hours productive work per day. How do you spend your money to improve reliability?

Example: Central Computing System

A central computer system (e.g., a server farm) is vital to an entire organization (e.g., an Internet shopping site). Any failure is serious.

Step 1: Gather data on every failure

- Create a database that records every failure
- Analyze every failure:

  hardware
  software (default)
  environment (e.g., power, air conditioning)
  human (e.g., operator error)

Example: Central Computing System

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 component
  Categories of human error

Example: Central Computing System

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

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