CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 2: Quality - Dependability

Gregory Gay
DIT635 - January 24, 2020

Today's Goals

- Discuss software quality in more detail.
- Introduce Scenarios
  - High-level "test cases" used to assess quality.
- Discuss Dependability
  - How we build evidence that the system is good enough to release.
  - Encompasses correctness, reliability, safety, and robustness
  - How we can measure and assess reliability

Software Quality

- We all want **high-quality** software.
- We don’t all agree on the definition of quality.
- Quality encompasses both **what** the system does and **how** it does it.
  - How *quickly* it runs.
  - How *secure* it is.
  - How *available* its services are.
  - How easy it is to *modify*.
- Quality is hard to measure and assess objectively.

Quality Attributes

- Quality attributes describe desired properties of the system under development.
- Developers must prioritize quality attributes and design a system that meets chosen thresholds.
- Most relevant for this course: **dependability**
  - The ability of the system to *consistently* offer correct functionality, even under *unforeseen* or *unsafe* conditions.

Quality Attributes

- Performance
  - The ability of a system to meet timing requirements. When events occur, the system must respond quickly.
- Security
  - The ability of a system to protect information from unauthorized access while providing service to authorized users.
- Scalability
  - The ability to “grow” the system to process an increasing number of concurrent requests.

Quality Attributes

- Availability
  - The ability to carry out a task when needed, to minimize “downtime”, and to recover from failures.
- Usability
  - The ability of the software to enable users to perform desired tasks and provide support to users.
  - How easy is it to use the system, learn its features, adapt the system to meet user needs, and increase confidence and satisfaction in system use?

Scenarios

Scenarios

- A **scenario** is a well-defined description of an interaction between an external entity and the system.
  - It defines the event that triggers the scenario, the interaction initiated by the external entity, and the response required of the system.
  - Similar to use cases or user stories, but examines both quality and functionality.
  - A sort of “high-level” test case of the system.

Scenarios

Capture a range of requirements:

- A set of interactions with users to which a system must respond.
- Processing in response to timed events.
- Peak load situations that could occur.
- Regulator demands.
- Failure response.
- A change that a maintainer might make.
- Any situation that the design must handle.

Scenario Types

- **Functional Scenarios** define how the system responds to external stimuli.
  - Users initiate transactions, AIR call or data sent through an interface, timed events.
- **System Quality Scenarios** define how the system should react in order to exhibit quality properties.
  - Ability to be modified to provide new functionality, to cope with peak load, to protect critical information.

Scenario Usage

- Provide input to architecture definition.
  - Help flesh out and find missing requirements.
- Evaluate the architecture
  - Force description of execution paths through system
  - Find missing/incompatible interfaces.
- Communicate with stakeholders
  - Concrete, easy to understand.
- Drive the testing process
  - Help prioritize testing efforts.

Functional Scenarios

Functional Scenario Format

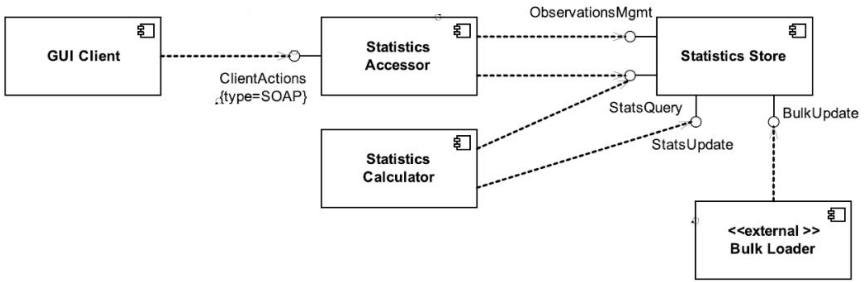
- **Overview**
  - Brief description of what the scenario illustrates.
- **System State**
  - State of system before the scenario starts.
- **System Environment**
  - Significant observations about the environment that the system is running in.
- **External Stimulus**
  - The event that sets off the scenario.
- **Required System Response**
  - How should the system respond?

Functional Scenario Format

- External stimulus should describe both the **actor** making a request and **action**.
  - Actor: the user, environmental stimulus such as a failure or timer, or external system.
  - Action: the request, event, or activity.
- Required system response should describe both how the system responds and a **response measure**.
  - Success or failure criterion for the response.

Example: Statistics Processing

- Raw data is loaded into a database.
- Derived statistics are calculated automatically based on the data.



- Statisticians view the data and make reports.
- Clients access statistics and make deductions that are checked manually.

Functional Scenario: Incremental Statistics Update

- **Overview:** How the system deals with a change to the existing base data.
- **System State:** Summary statistics already exist for the sales quarter that the incremental statistics refer to. The system's databases have enough space to cope with the processing required for this update.
- **System Environment:** The deployment environment is operating normally, without problems.
- **External Stimulus:** Update to sales transactions for the previous quarter arrives via the Bulk Data Load external interface.
- **Required System Response:** The incoming data should automatically trigger background statistical processing to update the summary statistics for the affected quarter to reflect the updated sales transaction data. The old summary statistics should stay available until the new ones are ready.

Quality Scenarios

Quality Scenario Format

- Overview
  - Brief description of what the scenario illustrates.
- System State
  - Aspects of the state that affect quality
  - (i.e., information stored in the system)
- System Environment
  - Significant observations about the environment that the system is running in.

Quality Scenario Format

- External Stimulus
  - Environmental factors that initiate the scenario.
  - (i.e., infrastructure changes or failures, security attacks, etc.)
- Required System Response
  - How should it respond (from a quantifiable point of view)?
  - (i.e., how should it handle a defined increase in requests)?

Example - Failure in Summary Database Instance

- **Overview:** How system behaves when database writes fail.
- **System state:** N/A
- **System environment:** The deployment environment is working correctly.
- **External Stimulus:** While writing summary statistics to the database, the system receives an exception indicating that the write failed (e.g., the database is full).
- **Required system behavior:** The system should immediately stop processing the statistics set it is working on and leave any work in progress behind. The system should log a fatal message to the operational console monitoring system and shut down.

Examples - Daily Data Update Increases in Size

- **Overview:** How the system's end-of-day processing behaves when regular data volumes are suddenly greatly exceeded.
- **System state:** The system has summary statistics in its database for data that has been processed, and the system's processing elements are lightly loaded at the current rate of system load.
- **System environment:** The deployment environment is working correctly, and data is arriving at a steady rate of 1,000 to 1,500 items per hour.
- **External Stimulus:** The data update rate on a particular day suddenly increases to 4,000 items per hour.
- **Required system behavior:** When the end-of-day processing starts, the system should process that day's data set for a period until the processing time exceeds a system-configurable limit. At that point, the system should stop processing the data set, discard work in process, leave the previous set of summary statistics in place, and log a diagnostic message (including cause and action taken) to the operational console monitoring system.

Performance Scenarios

- **Overview:** Description of the scenario.
- **System/environment state:** The system can be in various operational modes, such as normal, emergency, peak load, or overload.
- **External Stimulus:** Stimuli arrive from external or internal sources. The stimuli are event arrivals. The arrival pattern can be periodic, stochastic, or sporadic, characterized by numeric parameters.
- **Required system behavior:** The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode).
- **Response measure:** The response measures are the time it takes to process the arriving events (latency or a deadline), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate).

Performance Scenarios

- For real-time systems (i.e., embedded devices), measurements are absolute.
  - Look at worst-case scenario.
- For non-real-time systems, measurements should be probabilistic.
  - 95% of the time, the response should be N.
  - 99% of the time, the response should be M.

Example Performance Scenario

- **Overview:** Check system responsiveness for adding items to shopping cart under normal operating conditions.
- **System/environment state:** Normal load is defined as deployment environment with no failures and less than 20 customer requests per second. System is communicating over good internet connection to acceptable client (see glossary for expected internet / client specifications).
- **External Stimulus:** Customer adds product to shopping cart.
- **Required system behavior:** Web page refreshes. Icon on right side of web page displays last item added to cart. If item is out of stock, cart icon has exclamation point overlay on top of cart icon.
- **Response measure:** In 95% of requests, web page is loaded and displayed to user within 1 second. In 99.9% of requests, web page is loaded and displayed to user within 5 seconds.

Availability Scenarios

- **Overview:** Description of the scenario.
- **System/environment state:** The state of the system when the fault or failure occurs may also affect the desired system response. If the system has already failed and is not in normal mode, it may be desirable to shut it down. If this is the first failure, degradation of response time or functions may be preferred.
- **External Stimulus:** Differentiate between internal and external origins of failure because desired system response may be different. Stimuli is an *omission* (a component fails to respond to an input), a *crash* (component repeatedly suffers omission faults), *timing* (a component responds but the response is early or late) or *response* (a component responds with an incorrect value).

Availability Scenarios

- **Required system behavior:** There are a number of possible reactions to a failure. Fault must be detected and isolated before any other response is possible. After the fault is detected, the system must recover from it. Actions include logging the failure, notifying selected users or other systems, taking actions to limit the damage caused by the fault, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair.
- **Response measure:** Can specify an availability percentage, or it can specify a time to detect the fault, time to repair the fault, times or time intervals where system must be available, or duration for which the system must be available.

Example Availability Scenario

- **Overview:** One of the client-facing web servers fails during transmission of client page update.
- **System/environment state:** System is working correctly under normal load. Customer has generated a “add item to shopping cart” post, which was routed to web server <X> in transaction pool.
- **External Stimulus:** Web server <X> crashes during response generation.

Example Availability Scenario

- **Required system behavior:** Response page may be corrupted on client browser. Load balancer component no longer receives heartbeat message from web server and so removes it from the pool of available servers after 2s of missed messages, or upon next request sent to the server. Load balancer will remove the server from the pool of available servers. From client's perspective, a page reload will be automatically routed to alternate server by load balancer and page will be correctly displayed.
- **Response measure:** Upon client-side page refresh, client state and display contains state after last transaction. Time for re-routed refresh is equivalent to "standard" refresh (<1 second 95% of the time).

“Good” Scenarios

- Credible
  - Describes a realistic scenario.
- Valuable
  - Can be directly used during architectural definition.
- Specific
  - Addresses a single, concrete situation.
- Precise
  - Intended user of scenario should be clear about the described situation and response.
- Comprehensible
  - Writing should be unambiguous and free of jargon.

Effective Scenario Use

- **Identify a focused scenario set**
  - Too many scenarios can be distracting.
  - Prioritize no more than 15-20.
- **Use distinct scenarios**
  - Avoid having multiple scenarios centered around near-identical events. They are redundant.
  - Consider demands placed on the system.
- **Use scenarios early**
  - Most impactful early in development to focus design activities on most important aspects of the system.

Effective Scenario Use

- Include system quality scenarios!
  - Great potential for investigating, validating, and understanding quality properties.
  - You will augment stakeholder-provided scenarios to consider quality.
- Include failure scenarios!
  - Consider important failure cases and use scenarios to address them.
- Involve stakeholders closely
  - Stakeholders may reveal scenarios you didn’t consider and have differing priorities than you do.

Let’s take a break!

Dependability

When is Software Ready for Release?

- Can we can argue that we've done enough?
- Provide evidence that the system is *dependable*.
- The goal of dependability is to establish four things about the system:
  - That it is **correct**.
  - That it is **reliable**.
  - That it is **safe**.
  - That is is **robust**.

Correctness

- A program is **correct** if it is consistent with its specifications.
  - A program cannot be 30% correct. It is either correct or not correct.
  - A program can easily be shown to be correct with respect to a bad specification. However, it is often impossible to prove correctness with a good, detailed specification.
  - Correctness is a goal to aim for, but is rarely provably achieved.

Reliability

- A statistical approximation of correctness.
- Reliability is a measure of the likelihood of correct behavior from some period of observed behavior.
  - Time period, number of system executions
  - Measured relative to a specification and a usage profile (expected pattern of interaction).
    - Reliability is dependent on how the system is interacted with by a user.

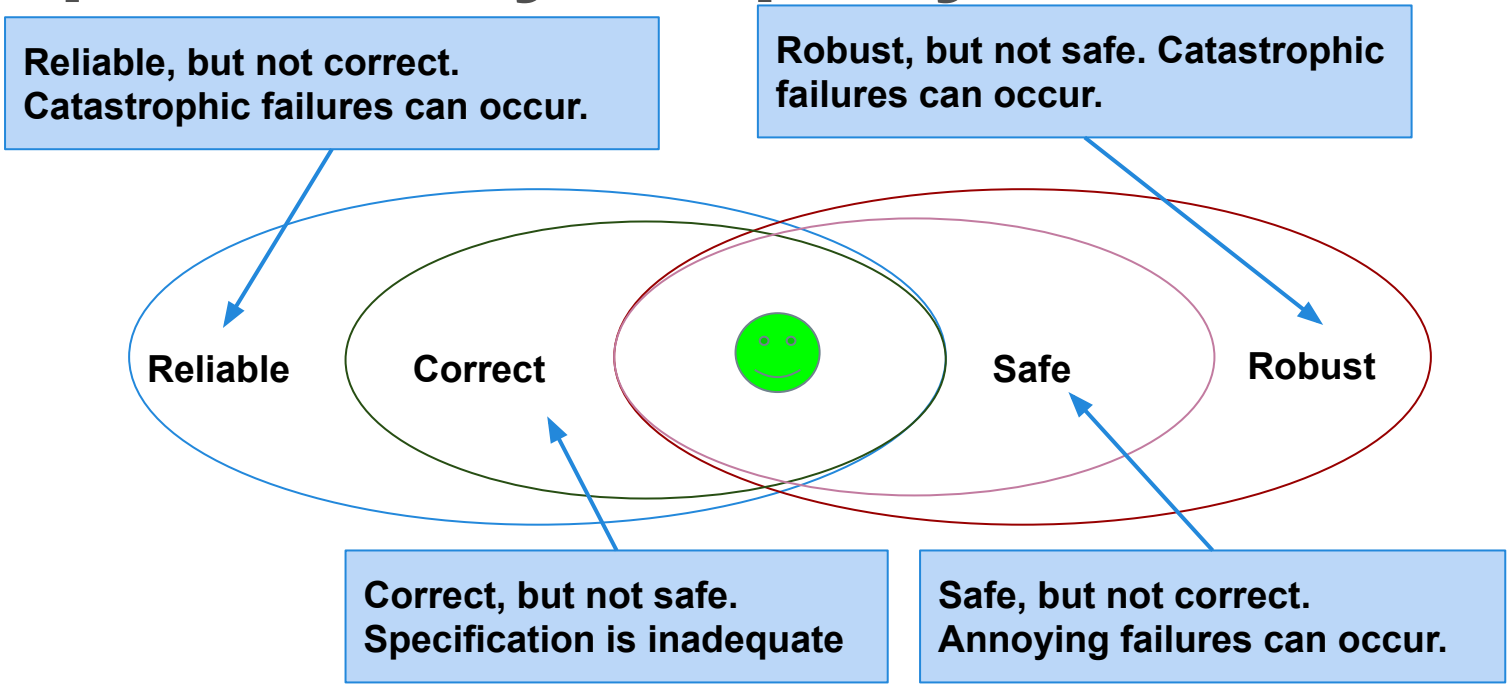
Safety

- Two flaws with correctness/reliability:
  - Success is relative to the strength of the specification.
  - Severity of a failure is not considered. Some failures are worse than others.
- **Safety** is the ability of the software to avoid *hazards*.
  - Hazard = any undesirable situation.
  - Relies on a specification of hazards.
    - But is only concerned with avoiding hazards, not other aspects of correctness.

Robustness

- Correctness and reliability are contingent on normal operating conditions.
- Software that is “correct” may still fail when the assumptions of its design are violated. *How* it fails matters.
- Software that “gracefully” fails is **robust**.
  - Consider events that could cause system failure.
  - Decide on an appropriate counter-measure to ensure graceful degradation of services.

Dependability Property Relations



Measuring Dependability

- Finding all faults is nearly impossible, and always expensive.
- We can *always* test more.
- Must establish criteria for when the system is dependable *enough* to release.
  - Correctness hard to prove conclusively.
  - Robustness/Safety important, but not enough.
  - **Reliability** is the basis for arguing dependability.

Analyzing Reliability

What is Reliability?

- Reliability is the probability of failure-free operation for a specified time in a specified environment for a given purpose.
- This means different things depending on the system and the users of that system.
- Informally, reliability is a measure of how well users think the system provides the services they require.

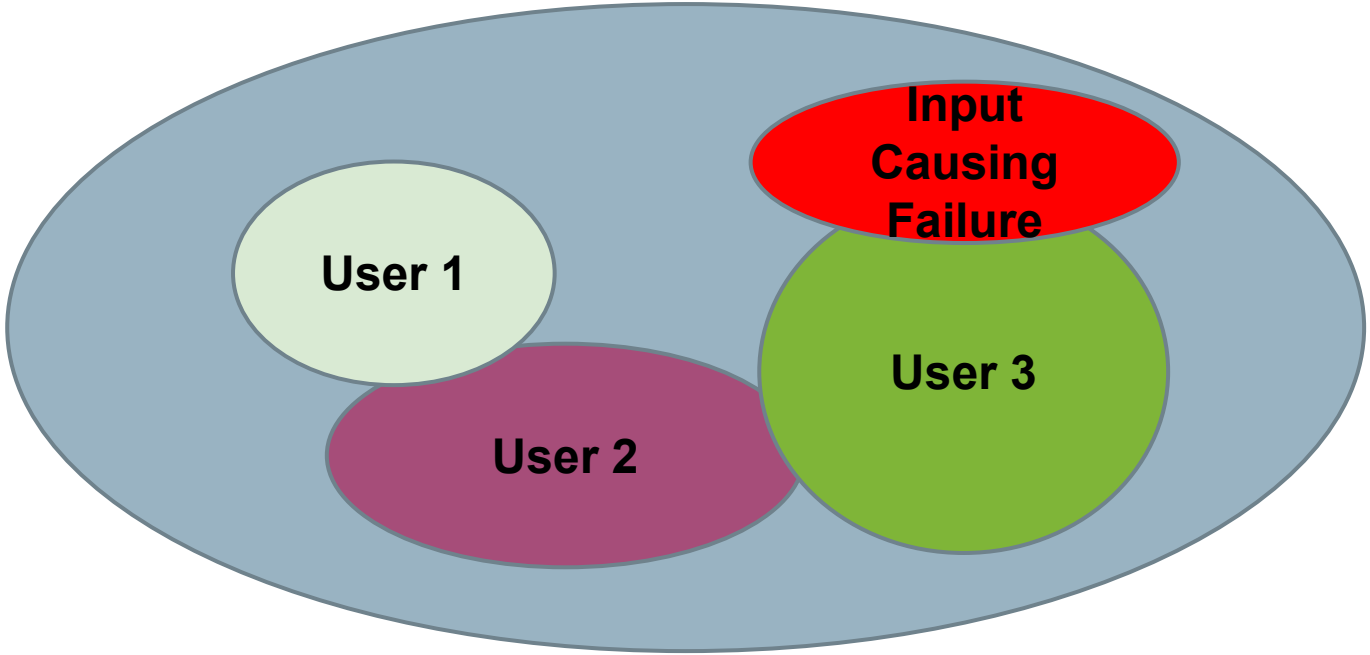
Reliability is Measurable

- Reliability can be defined and measured.
- Reliability requirements can be specified:
  - Non-functional requirements can define the number of failures that are acceptable during normal use of the system, or the time in which the system is allowed to be unavailable for use.
  - Functional requirements can define how the software avoids, detects, and tolerates faults to ensure they don't lead to failures.
  - **Scenarios should be written for both.**

Improving Reliability

- Reliability is improved when software faults that occur in the most frequently-used parts of the software are removed.
  - Removing X% of the faults will not necessarily lead to an X% improvement in reliability.
    - In a study, removing 60% of the faults actually led to a 3% reliability improvement.
- Removing faults with serious consequences is the top priority.

Reliability Perception



Software Reliability

- Cannot be defined objectively for all situations.
  - Reliability measurements quoted out of context are meaningless.
- Requires operational profile for its definition.
  - A profile of the expected pattern of software usage.
- Must consider fault consequences.
  - Not all faults are equally serious.
  - System is perceived as unreliable if there are more serious faults.

How to Measure Reliability

- Measuring reliability is normal when building hardware, but hardware metrics often aren't suitable for software.
  - Based on component failures and the need to repair or replace a component once it has failed.
  - In hardware, the design is assumed to be correct.
- Software failures are always design failures.
  - Often, the system is available even though a failure has occurred.

Availability

- The availability of a system reflects its ability to deliver services when available (uptime/total time).
  - Takes repair and restart time into account.
  - Does not tend to take incorrect computations (partial failures) into account.
- Availability of 0.9999 means the system is available 99.99% of the time.
  - 0.9 = down for 144 minutes a day, 0.99 = down for 14.4 minutes, 0.999 = down for 84 seconds, 0.9999 = down for 8.4 seconds.

Probability of Failure on Demand (POFOD)

- The likelihood that a service request will result in a system failure (failures/requests over a period).
- POFOD = 0.001 means that 1 out of 1000 service requests result in a failure.
- Should be used in situations where a failure on request is serious.
  - Independent of the frequency of requests.
  - 1/1000 failure rate sounds risky, but if one failure per lifetime, it is good.

Rate of Occurrence of Fault (ROCOF)

- Frequency of the occurrence of unexpected behavior.
  - Probable number of failures over a period of time or number of system executions.
- ROCOF of 0.02 means that 2 failures are likely per 100 time units.
- Most appropriate metric when requests are made on a regular basis (such as a shop).

Mean Time Between Failures (MTBF)

- Measures the average length of time between observed failures.
  - Requires the timestamp of each failure and the timestamp of when the system resumed service.
- MTBF of 500 means that the time between failures is, on average, 500 time units (or requests).
- For systems with long user sessions, you want to require a long MTBF.

Data Needed for Measurements

To assess reliability, data must be captured from users' sessions with the system:

- Measure the number of failures per a given number of requests (used for POFOD).
- Measure the number of failures, plus total elapsed time or request number (ROCOF).
- Requires the timestamp of each failure and the timestamp of when service is resumed (MTBF).
- Measure the time to restart after a failure (availability).

Reliability Examples

- Provide software with 10000 requests.
  - Wrong result on 35 requests, crash on 5 requests.
  - What is the POFOD?
- 40 / 10000 = 0.0004
- Run the software for 144 hours
  - (6 million requests). Software failed on 6 requests.
  - What is the ROCOF? The POFOD?
- ROCOF = 6/144 = 1/24 = 0.04
- POFOD = 6/6000000 = (10^{-6})

Reliability Examples

- You advertise a piece of software with a ROCOF of 0.001 failures per hour.
  - However, it takes 3 hours (on average) to get the system up again after a failure.
  - What is the availability per year?

- Failures per year:
  - approximately 8760 hours per year (24*365)
  - 0.001 * 8760 = 8.76 failures per year
- Availability
  - 8.76 * 3 = 26.28 hours of downtime per year.
  - Availability = 0.997 ((8760 - 26.28)/8760)

Activity - Availability

- Your customers want an availability of at least 99%, a POFOD of less than 0.1, and ROCOF of less than 2 failures per 8 hour work period.
- After testing your code for 7 full days, 972 requests were made. The product failed 64 times (37 system crashes, 27 bad calculations) and it took an average of 2 minutes to restart after each failure.
  - What is the availability, POFOD, and ROCOF?
  - Can we calculate MTBF?
  - Is the product ready to ship?
  - If not, why not?

Activity Solution

- What is the rate of fault occurrence?
  - 64/168 hours = 0.38/hour = 3.04/8 hour work day
- What is the POFOD?
  - 64/972 = 0.066
- What is the availability?
  - Was down for (37*2) = 74 minutes out of 168 hours = 74/10089 minutes = 0.7% of the time.
  - Availability is 0.993.

Activity Solution

- Can we calculate MTBF?
  - No - need timestamps. We know how long they were down (on average), but not when each crash occurred.
- Is the product ready to ship?
  - No. Availability/POFOD are good, but ROCOF is too low.
  - Suggestions for improvement?

Reliability Economics

- Raising reliability is expensive. It may be cheaper to accept unreliability and pay for failure costs.
- The balancing point depends on social and political factors and the system type.
  - A reputation for unreliable products may hurt more than the cost of improving reliability.
  - Cost of failure depends on risks of failure. For business systems, modest reliability may be fine.

Assessing Reliability

- Rather than using tests to trigger faults, we can use tests (or scenarios) to measure reliability.
  - Test inputs should match the predicted usage profile of a user.
  - By recording errors and other measurements, we can calculate ROCOF, POFOD, etc.
  - An acceptable level of reliability should be specified and the software tested until that level is reached.
  - Scenarios should be repeatedly executed.

Operational Profiles

- Reflects how the software is used.
- Consists of classes of input and the probability of their occurrence.
- Can be specified in advance if other systems exist that perform similar actions.
- For new systems, it is harder to specify.
  - Conduct beta testing to gather initial usage data.
  - Remember that usage changes over time.

Statistical Testing Procedure

- Form an operational profile.
- Construct test input that reflects the profile.
  - Using scenarios can help.
- Apply inputs and count the frequency and type of failures that occur, along with the time between failures.
- After observing a statistically significant number of failures, compute the reliability.

Statistical Testing Challenges

- Operation profile uncertainty
  - A profile based on other systems may not be valid for your system.
- High cost of test input generation
  - Large volume of inputs needed. Can be expensive.
- Statistical uncertainty
  - Need to generate enough failures to estimate reliability. This is hard when the system is already reliable.
  - Hard to estimate confidence in operational profile.

Key Points

- Defining and applying scenarios ensures that desired quality attributes are shown.
  - **Functional scenarios** define how the system responds to external stimuli.
  - **System quality scenarios** define how the system responds to environmental factors that affect quality properties.
  - Should include the initial system state and environment, external stimulus or environment changes, and the required system response (and how to measure it).

Key Points

- Dependability is one of the most important software characteristics.
  - Aim for correctness, reliability, safety, robustness.
  - Often assessed using reliability.
- Reliability depends on the pattern of usage of the software. Different users will interact differently.
  - Faulty software can be reliable for some users.

Key Points

- Reliability can be measured quantitatively.
  - ROCOF, POFOD, Availability, MTBF
- Statistical testing is used to estimate reliability.
  - Construct and execute scenarios (or concrete tests) multiple times.

Next Time

- Quality - Non-functional Attributes
  - Performance, Scalability, Availability, Security
- No exercise session today!

- **Form your teams!**
  - Deadline: Thursday, January 30
  - E-mail me (ggay@chalmers.se) with list of team members, e-mail addresses, and a team name.
    - Or e-mail if you want to be assigned to a team

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY