Software Reliability Engineering
Objectives

- Look at some details on Software Reliability Engineering (SRE)
  - Steps in the SRE process
  - Setting reliability objectives
  - Using operational profiles to guide effort
  - Interpreting reliability trend graphs
Reliability Focus

- “Testing can only prove the presence of errors, not their absence.”
  
  Dijkstra

- So, focus on reliability, not defects
  - Correctness
Software Reliability Engineering

- Software Reliability Engineering (SRE) addresses the measurement, modeling, and improvement of software reliability
- Use quantitative information to choose the most cost-effective software reliability strategies for your situation
Reliability Engineering Practices

- Define reliability objectives
- Use operational profiles to guide test execution
- Track failures during system tests
- Use reliability growth curves to track quality of product
- Release when quality of product meets reliability objectives
Establish Reliability Objectives

- Predict software reliability growth
- Trade-offs between time, reliability, cost, performance, etc.
- When to stop testing – release decision
- How much post-release support to plan for

Develop Operational Profiles (OPs)

Plan Tests Matched to OPs

Use Test Results to Drive Decisions

Engineer “Just Right” Reliability

Determine Achieved Reliability and OPs

(Fielded System)
Reliability and Failure Intensity

- Failure intensity: Number of failures per hour of operation
- Reliability is the inverse of failure intensity (FI)

![Graph showing the relationship between failure intensity and reliability over time.](image)
Defining Reliability Objectives

- Set quantitative targets for level of reliability that make business sense

<table>
<thead>
<tr>
<th>Impact of a failure</th>
<th>FI Objective</th>
<th>MTBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>100’s deaths, &gt;=$10^9 cost</td>
<td>10^-9</td>
<td>114,000yrs</td>
</tr>
<tr>
<td>1-2 deaths, around $10^6 cost</td>
<td>10^-6</td>
<td>114 yrs</td>
</tr>
<tr>
<td>$1,000 cost</td>
<td>10^-3</td>
<td>6 weeks</td>
</tr>
<tr>
<td>$100 cost</td>
<td>10^-2</td>
<td>100 h</td>
</tr>
<tr>
<td>$10 cost</td>
<td>10^-1</td>
<td>10 h</td>
</tr>
<tr>
<td>$1 cost</td>
<td>1</td>
<td>1 h</td>
</tr>
</tbody>
</table>

From John D. Musa
Operational Profiles Guide Effort

- Guide software development priorities and quality effort by what the user will use the most often
  - Pareto principle: 20% of the software’s functionality or “size” may satisfy 80% of the user’s needs
  - Operational profiles expose most frequently used product features
## Operational Profile

- Sample application: Word Processor

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
<th>Approx. Relative Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open file</td>
<td>1/session (5 session/day)</td>
<td>0.001</td>
</tr>
<tr>
<td>Close file</td>
<td>1/session</td>
<td>0.001</td>
</tr>
<tr>
<td>Save file</td>
<td>25/session</td>
<td>0.04</td>
</tr>
<tr>
<td>Insert text</td>
<td>1000/session</td>
<td>1.0</td>
</tr>
<tr>
<td>Cut-and-paste</td>
<td>6/session</td>
<td>0.006</td>
</tr>
<tr>
<td>Check spelling</td>
<td>1000/session</td>
<td>1.0</td>
</tr>
<tr>
<td>Repaginate</td>
<td>100/session</td>
<td>0.1</td>
</tr>
<tr>
<td>Upgrade software</td>
<td>1/ 6 months</td>
<td>0.000001</td>
</tr>
</tbody>
</table>
Testing Based on Operational Profiles

- Done during black-box system testing
- Mix of test cases that match operational profile
- If possible, create automated test harness to execute test cases
  - Need to run large numbers of test cases with randomized parameters for statistical validity
- Execute test cases in randomized order, with selection patterns matching frequencies in operational profile
  - Simulating actual pattern of usage
Studying Patterns in the Trends of Reliability Growth
Reliability Metric

- Estimated failure intensity
  - (Reliability = 1 / failure intensity)
  - Use reliability tracking and analysis tools to show actual (to date) and predicted (future) estimates of how failure intensity varies over time

- The curve is referred to as the “reliability growth curve”
  - Note that the product being tested varies over time, with fixes and new code
  - In-process feedback on how quality is changing over time
Code Integration/Build Patterns

- Most large projects have periodic builds
  - Development team integrates a new chunk of code into the product and delivers to test team
- Test team does black box system testing
  - Identifies defects (failures) and reports them to development team
- Track pattern of defects found during system testing to see how reliability varies as development progresses
  - Defects found should decrease over time as defects are removed, but each new chunk of code adds more defects
- Pattern of reliability growth curve tells us about the code being added, and whether the product code is becoming more stable
- Pattern can also be used to statistically predict how much more testing will be needed before desired reliability target reached
  - Useful predictions only after most of the code is integrated and failure rates trend downward
Tracking Failures During Testing

- Enter data about when failures occurred during system testing into reliability tool such as CASRE (Computer-Aided Software Reliability Engineering tool) or Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS)
  - Plots graph of failure intensity vs. development/test time

In concept, a nice smooth curve of reliability growth

From netserver.cerc.wvu.edu/numsse/Fall2003/691D/lec3.ppt
Reliability Over Time

Hardware “Bathtub” Model

Software Model

DACS Software Reliability Source Book
Predicting with a Software Reliability Growth Model

Actual data points based on software validation testing

Failure intensity goal required to ship software

Predicted software reliability after shipment to customers

[Rakitin]
A More Realistic Curve During Development

Many Statistical Models of Reliability Growth

The Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) contains a collection of several reliability models, including:

- The Littlewood-Veral Bayesian model
- The Musa execution time model
- The geometric model
- The nonhomogeneous Poisson model for execution time data
- The Musa logarithmic Poisson execution time model
- The generalized Poisson model for interval data
- The nonhomogeneous Poisson model for interval data
- The Brooks-Motley discrete software reliability model
- The Schneidewind maximum likelihood model
- The Yamada S-shaped reliability growth model

Get SMERFS at http://www.slingcode.com/smerfs/
Model Comparison Using SMERFS^3

[Dolores R. Wallace Practical Software Reliability Modeling]
Interpreting Reliability Growth Curves

- Spikes are normally associated with new code being added
- Larger volumes of code or more unreliable code causes bigger spikes
  - The curve itself tells us about the stability of the code base over time
- If small code changes/additions cause a big spike, the code is really poor quality or impacts many other modules heavily
- The code base is stabilizing when curve trends significantly downward
  - Release (ideally) only when curve drops below target failure intensity objective ... indicates right time to stop testing
  - Can statistically predict how much more test effort needed before target failure intensity objective needed
Limitations of Reliability Curves

- Operational profiles are often “best guesses,” especially for new software products
- The reliability models are empirical and only approximations
- Failure intensity objectives should really be different for different criticality levels of different kinds of failures
  - Results in loss of statistical validity!
- Automating test execution is challenging (particularly building verifiers) and costly
  - But it does save a lot over the long run
  - More worthwhile when reliability needs are high
- Hard to read much from the growth curves till later stages of system testing … very late in the development cycle
Reliability Certification

- Another use for reliability engineering is to determine the reliability of a received or acquired software product: Certification of Acceptability
  - For example, you are evaluating web servers for your company website – reliability is a major criterion
- Build a test suite representative of your likely usage
  - Put up some pages, maybe including forms
  - Create test suite that generates traffic
  - Log failures such as not loading, wrong data received, server time out
  - Track failure patterns over time
- Evaluate multiple products or new releases using test suite, to determine reliability
  - Avoids major problems and delays with poor vendor software
- Note that this applies the analysis to a fixed code base
  - Fewer problems with statistical validity
Example Certification Curve

Run the product and track the time of occurrence of each failure

- Failure #1 Decision: Don’t know enough yet, so continue running
- Failure #2 Decision: Don’t know enough yet, so continue running
- Failure #3 Decision: Came far enough later (in MTBF sense) that the product is certified acceptable
- Had failures #3-7 happened as shown by the x’s, then the failures are occurring too frequently -- Reject

Summary

- Software Reliability Engineering is a scientific (statistical) approach to reliability
- Vast improvement over common current practice
  - “Keep testing until all our test cases run and we feel reasonably confident”
- Avoids under-engineering as well as over-engineering (“zero defects”)
- When done well, Software Reliability Engineering adds ~1% to project cost
  - Musa’s numbers: ~10% for medium-sized projects if you include cost of automated testing
  - Note that as the number of builds and releases increases, automated testing more than pays for itself