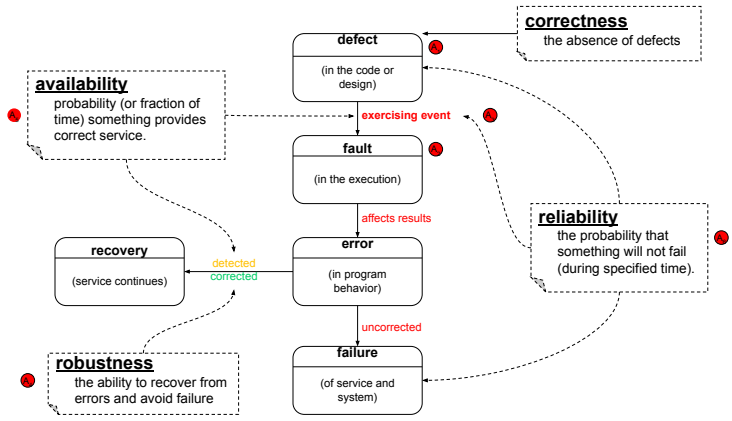
Software Robustness

- terminology
  - phases of failures, measuring goodness
- developing robust software
  - philosophy and general approach
  - planning for error management
    - enumeration, prioritization
    - detection, diagnosis, containment
    - handling, reporting
- Failure Mode Analysis exercise

Taxonomy of not-working



why Software fails so often

1. we implement functionality incorrectly
   - often, because we don't understand correctness
   - we make mistakes without realizing they are wrong
2. we make optimistic assumptions
   - valid inputs, good data, successful requests
   - allowing errors to cascade to greater failures

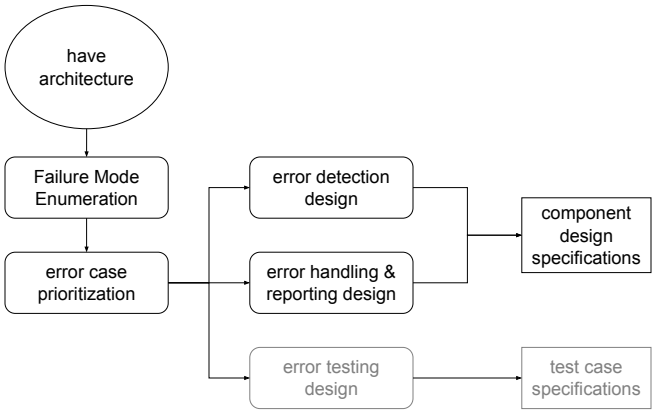
solution:

1. completely define correctness
2. explicitly state all assumptions
3. review these along with the design
4. use them to guide the implementation

why Software fails so badly

- error handling is seldom well specified
  - requirements seldom enumerate error cases
  - specifications seldom describe handling
- architecture often ignores error handling
  - errors are difficult to diagnose & contain
- error handling is seldom well tested
  - we test against the specifications ☺
  - we can't reliably cause most errors
- bottom line – because we have let it do so

Designing for Robustness



Failure Mode Enumeration

- enumerate all likely external errors
  - services: resources, access, ...
  - hardware: transient and persistent
  - data: bad configuration, corrupt data, ...
  - communication errors: protocol, link, node, …
- and general classes of internal errors
  - resulting from failures of our own components (may be easier to focus on symptoms here)
- include problems reported to support
  - involving similar products

Error Prioritization

- assess likelihood (common, occasional, rare)
  - based on prior experience w/such services
  - based on estimated transaction rates
  - based on intrinsic risk (novelty, complexity)
- assess impact (serious, moderate, minor)
  - serious: loss of service or data
  - moderate: a few brief and isolated failures
  - minor: user can easily work around problem
- rank all of the errors on a priority ladder
  - from <common,serious> to <rare,minor>

Error Detection

- to handle an error we must first detect it
  - in most cases, by adding error detection code
    - checking return codes, testing pre-conditions
  - some errors may be quite difficult to detect
- general principles of error detection
  - close as possible to errors first appearance
  - general checks that find many problems
  - simple w/high probability of correctness
  - low overhead – this stuff executes often

Difficult Error Detection

- some errors may be hard to detect
  - symptoms are very subtle (not total failures)
  - may involve relationships among transactions
- some may be hard to discriminate
  - the symptoms could indicate many problems
  - each of which requires very different handling
- this may indicate problems w/architecture
  - if likely/severe errors are hard to detect, consider design changes that would make them easier to detect and/or isolate

Diagnosis and Containment

- diagnosis
  - source: where did the error originate
  - impact: recoverable, degraded, failed
  - persistence: transient, chronic, permanent
  - much diagnosis is done at design time
- containment
  - which components have actually failed
  - which components have been affected
  - minimize impact of the error
  - much containment is dictated by architecture

Exercise

- Get together in your teams
- Consider an at-risk component in your project
- Perform a Failure Mode Analysis
  - enumerate likely errors that can happen in system
    - more interesting ones than “bad user input”
    - perhaps requests other services
  - rank them by likelihood and impact
  - describe how you will recognize each case
    - symptoms and/or test to confirm
    - source, scope, impact, containment
  - describe how to handle the problem
  - identify and analyze 2-3 good problems

Exercise Discussion

1. What kinds of problems did you come up with, and how did you sub-class them?
2. How did you estimate likelihood and severity?
3. How hard was it to choose the most important problem?
4. How did you detect and deal with the problem?
5. Did this cause you to reconsider any aspects of your architecture?

For Next Lecture

- McConnell: chapter 23 - Debugging
- Kampe: Forensic Debugging
- Kampe: Root Cause Analysis
- Wiki: Defect Tracking
- Black: Writing a Bug Report
- Kampe: Severity and Priority
- Github Issues … a simple bug tracking system
  - labels (e.g. bug, feature)
  - milestones (when is it needed)
  - assignee (who is supposed to deal with it)

Supplementary Slides

Don't trust other components

- explicit return code checking
  - always check return codes from all requests
- assertion checking parameters & returns
  - sanity check input parameters
  - sanity check results returned from requests
- detect network/IPC no-response errors
  - choose a time that is clearly too long to wait
  - set timer at start of request, cancel at end
  - if timer goes off, something is wrong w/server

Don't trust yourself

- in-line consistency assertions
  - sanity check results of our own computations
  - find errors before we return them to others
- periodic consistency audits
  - scan and sanity check persistent resources
  - find errors we missed when they happened
- exception handling (for very serious errors)
  - catch, report, and handle all exceptions
  - these may happen long after original error

Don't trust your error detection

- external health monitoring
  - independent program, maybe another system
  - sending periodic test transactions
  - checking responsiveness/reasonableness
- provides end-to-end testing
  - network links, front-end, system, application
- very general health monitoring
  - will find a very wide range of failures
- requires 3rd party error report mechanism

handling well-contained errors

- can we correct the error and continue?
  - error may not stop us from fulfilling request
  - retry request, perhaps from a new server
- can we inform requestor and continue?
  - return an error code identifying the problem
  - let your caller decide how to handle problem
- can we continue with reduced capability?
  - fail requests needing compromised resources
  - continue fulfilling other types of requests

handling poorly-contained errors

- assume entire component compromised
  - may be unsafe to continue providing service
  - this is a correctness/robustness trade-off
- options
  - refuse all further requests to this component
  - shut down this component
  - shut down the entire system
- architectural fault containment domains
  - limit potential spread of the damage
  - obvious units of shut-down/restart

Error reporting

- what to report, and to whom
  - explain what went wrong to the user
  - log incidents that require corrective action
  - record information to help establish trends
  - record information for post-mortem analysis
- reporting mechanisms
  - detailed error codes (for software)
  - error messages/dialogs (for users)
  - system error log (for support)
  - automated reporting and diagnosis services