L1 - Course Introduction and Overview
L2 - Project Management
L3 - Requirements

**L4 - Acceptance Testing and Quality Factors**
L5 – UML
L6 - Design Patterns
L7 - System Design and Architecture
L8 - Testing Theory
L9 - Testing in Practice
L10 - Inspection
L11 - Software Life Cycles and Configuration Management
L12 - Software Quality Management
L13 - Course Summary, Exam examples, Questions
A Software Life-cycle Model

Which part will we talk about today?

- Requirements
- System Design (Architecture, High-level Design)
- Module Design (Program Design, Detailed Design)
- Implementation of Units (classes, procedures, functions)
- Module Testing (Integration testing of units)
- System Testing (Integration testing of modules)
- Acceptance Test (Release testing)
- Maintainence

Validate Requirements, Verify Specification
- Verify System Design
- Verify Module Design
- Verify Implementation

Verify System Design
- Project Management
- Software Quality Assurance (SQA), Supporting Tools, Education

Part I
Software quality factors

Part II
Usability

Part III
Reliability

Part IV
Critical systems
Agenda - What will you learn today?

Part I – Acceptance testing

Part II – Software quality factors

Part III – Critical systems

Part I
Software quality factors

Part II
Usability

Part III
Reliability

Part IV
Critical systems

Kristian Sandahl
krisa@ida.liu.se
Part I

Acceptance testing
Requirements - short revisit

Many sources

Elicitation → Analysis → Specification → Validation

Requirements
System
Result
Decision

Written SRS
Detailed use-cases
Detailed models
Formal analysis
Simulation
Negotiation
Conceptual model
Ideas
Notes

Part I
Software quality factors

Part II
Usability

Part III
Reliability

Part IV
Critical systems
Validation of requirements

Before design and coding
- Inspections
- Cross-referencing
- Interviews
- Checklists
- Scenarios
- Proofs
- Model validation
- Simulation
- Prototyping
Acceptance testing

- Purpose
- Benchmarking
- Pilot testing
  - alpha test
  - beta test
- Installation testing
- Parallel testing
Acceptance testing - documentation

IEEE-std 829

- Test plan
  - Identify *test items*
  - Plan resources, tasks, schedule
- Test design specifications
  - Test techniques to be used
  - Analysis methods
  - How to handle groups of tests
- Test case specifications
  2.1. Test case identifier
  2.2. Objective eg priority
  2.3. Inputs
  2.4. Outcome(s)
  2.5. Environmental needs
  2.6. Special procedural requirements – pre- and post processing
  2.7. Intercase dependencies – order of execution
Acceptance testing - documentation

- Test procedure specifications
  - Order of each step for all participants

- Test logs
  - Results
  - Changes to plans

- Test anomaly report
  - Things worth investigating
  - Known impact
  - Recommendation

- Test summary reports

IEEE-std 829
Part II
Software quality factors

“Reliability”

Part I
Software quality factors

Part II
Usability

Part III
Reliability

Part IV
Critical systems
Perspectives of quality

- Transcendent – something we learn to recognize
- Product-based – measurable variable
- Usage-based – in the eyes of the beholder
- Manufacturing-based – conformance to requirements
- Value-based – market sets the value

Many opinions ⇒ Statistical techniques

Part I
Software quality factors

Part II
Usability

Part III
Reliability

Part IV
Critical systems
Quality factors

- Correctness
- Reliability
- Efficiency
- Usability
- Integrity
- Maintainability
- Flexibility
- Testability
- Security

- Portability
- Reusability
- Interoperability
- Survivability
- Safety
- Manageability
- Supportability
- Replaceability
- Functionality

Price?
Quantification of Quality Factors

Quality factors must be quantified to:

- State precise requirements
- Validate the results
- An important class of non-functional requirements
- Original:
  - The system’s usability must be optimal
- Better:
  - The users shall consult the help function no more than once a week on average
Testing the usability requirements

AND STARTING TODAY, ALL PASSWORDS MUST CONTAIN LETTERS, NUMBERS, DOODLES, SIGN LANGUAGE AND SQUIRREL NOISES.
Usability

- Relevance
- Efficiency
- Attitude
- Learnability
- Metrics:
Usability testing

- Think-aloud test
- Observation test
- Automatic logging
- Heuristic evaluation 50% hit rate
- User review with expert users
- Avoid trade-union representatives and IT-experts

Normal users
Testing process

- Input: artefact, test tasks (realistic, closed)
- Participants: test user, facilitator, log keeper
- Explain purpose
- Background and daily matters
- Task
- Observe
- If necessary: give hints
- Debriefing
- Output: List of problems
Usability problem severity classes

- Missing function
- Task failure
- Annoying
- Medium problem
- Minor problem
User stories in eXtreme Programming

- The customer is always available
- The customer writes user stories, ex:
  - The user is presented with information of all phones connected
  - The user changes parameters for a phone
  - The user activates a dialup connection
  - The user inactivates a dialup connection
- Embrace change
- Make frequent releases – one per week
Reliability

- The probability that the software executes with no failures during a specified time interval
  - Approximation: $\text{MTTF}/(1+\text{MTTF})$
  - Example:
    - http://www.ida.liu.se/~TDDC93/timetable/failure-based.xls
Failure intensity

- Easier to manage: Failure intensity, [failures / hours of execution time]
- Another approximation: $\lambda = (1-R)/t$
- Example
- Define target failure intensity
- Develop operational profile
- Plan tests
- Execute test
- Apply data to decisions

**Part I**
Software quality factors

**Part II**
Usability

**Part III**
Reliability

**Part IV**
Critical systems

Software reliability testing
## Failure intensity guideline

<table>
<thead>
<tr>
<th>Impact</th>
<th>Failure intensity</th>
<th>Time btwn failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hundreds of deaths, $10^9$ cost</td>
<td>$10^{-9}$</td>
<td>114 000 years</td>
</tr>
<tr>
<td>1-2 deaths, $10^6$ cost</td>
<td>$10^{-6}$</td>
<td>114 years</td>
</tr>
<tr>
<td>$1000$ 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>
Develop operational profiles

- Find operations and their relative probability
  - An operation is a major system logical task of short duration which returns control to the system when complete and whose processing is substantially different from other operations.
- Operational modes (e.g. day, night)
- Representation (tabular, graphical)
- Ignore non-critical with $p < 1-R_{\text{target}}$
- How do you find and assess probabilities of operations?
Prepare for test

- Feature test – check that each operation work properly
  - Load test – mimic filed usage
  - Tests are executed in runs comprising:
    - Operation,
    - input state,
    - direct and indirect variables
  - A test case is defined as a run with named direct variables and values
  - A test case is independent of the operational mode
  - Thus a test case can generate multiple runs
  - Problem: how to select test cases?
Select test cases

- Determine the total number of test cases.
- Sanity check: \#new operations + 100
- Distribute test cases proportionally to operations
- Assign at least one case per operation
- Accelerate rare, critical operation
- Maximise variable distance in input space
- Run test cases
- When a failure is found, log the time, correct the fault (don’t count twice)
- Decisions made according to reliability growth criteria
- Don’t count multiple failures due to single fault
- Run performance tests simultaneously
Part III

Critical systems
Critical systems

- When failed a critical system can cause injury of people, damage the environment or loose immense sum of money.
- Criticality is often expressed in terms of:
  - Reliability
  - Availability
  - Maintainability
  - Safety
  - Security
- Critical systems makes expensive methods worthwhile and needs experience

What horror stories (true or false) do you know of?
Hazard, Incident, Accident

- Hazard – System state or condition that may lead to an incident.
- Incident – Potentially dangerous system behaviour
- Accident – Unplanned sequence of events that lead to human death, severe injury or big loss of resources
Create safety requirements checklists

- Encapsulation of experience
  - Avoids errors by omission
  - Keep the list short, and there is only small costs

Examples:
- Safe start-up
- Initialise variables
- Manual override
- Off-line behaviour
- Time-out
- Unexpected input

- Possible sensor data
- Output values produced
- Value ranges
- Reasonableness of output
- Timing delays
- Overload behaviour
- Alarm management
- Data ageing
- Exception handling
- Reversible commands
- Fail-safe states
- Error detection and recovery
Involve external reviewers in the validation process

- External people lacks commitment to the product and the company
- Wisely selected, the union of competence can cover a vast area
- High costs in time and money
- The inspection process can limit the contribution of members
- Nominal groups perform best
- A meeting can be needed to gain personal commitment
Identify and analyse hazards

- A panel of experts can anticipate hazards
- Experiment with a suitable process in non-critical applications
- Spend time on training
- Use brainstorm-like mode of communication
- Costly

- Analyse the system from different viewpoints, for example:
  - potential victims
  - events in the system environment
  - behaviour of sub-systems
  - properties of hardware platform
  - properties of special software
Define safety requirements from hazard analysis

- Natural continuation of analysis
  - Specify:
    - avoidance
    - tolerance
    - detection
    - recovery
    - minimising damage
    - handling sub-sequent hazards
  - Often a matter of severity analysis and prioritisation
Cross-check operational and functional requirements against safety requirements

- Complements the hazard analysis
- Check the do’s against the don’t’s
- Create a traceability matrix Safety requirements × Functional requirements
- Determine the relationship
- Handle the potential hazards
Specify systems using formal specification

- Problems seem to be finding the most appropriate formalism and training
- Experiment, and hire competent people!
Collect incident experience

- An incident database helps to:
  - training
  - create and refine checklists
- The key to success is the retrieval functions
- A problem is that elements on very different levels of abstraction contribute to incidents
Learn from incident experience

Use incident data to:

- refine checklists
- refine incident description
- refine severity classes
- create general requirements
- reuse requirements
Establish an organisational safety culture

- Clear message from management
- Problem reporting channels
- Clear responsibility
- Egoless style of working
- Incentive and reward system
- Competence development
Summary - What have we learned today?

- Software quality is multi-faceted often in the eyes of the beholder.
- Usability can and shall be measured during a prototyping development project.
- Reliability can be assured through statistical usage testing.
- Critical systems development needs experience.