Unit 5: Software Safety (5 hours)

Software safety refers to the process of ensuring that software systems operate without causing unacceptable risk of harm to people, equipment, or the environment. It focuses on identifying, analyzing, and mitigating potential hazards that could result from software failures or unintended behavior, especially in critical systems such as those used in aviation, healthcare, automotive, military, and nuclear industries.

Explanation:

Software safety is not just about preventing bugs or crashes—it's about making sure that even if something goes wrong, the system doesn't lead to dangerous outcomes. For example:

- In a **medical device**, software safety ensures that a drug pump doesn't overdose a patient if a sensor fails.
- In a **car's braking system**, safety measures ensure the brakes still function properly even if part of the software malfunctions.

Key elements of software safety include:

- **Hazard analysis**: Identifying what could go wrong and what the consequences would be.
- **Risk assessment**: Determining how likely and severe those consequences are.
- **Design for safety**: Building the software to avoid or control hazards.
- **Verification and validation**: Making sure the software meets safety requirements.
- **Compliance**: Following safety standards like ISO 26262 (automotive), DO-178C (aviation), or IEC 62304 (medical).

Safety Critical Sytems

A **safety-critical system** is a system whose failure or malfunction may result in one or more of the following:

- **Death or serious injury** to people
- **Loss or severe damage** to equipment or property
- **Environmental harm**

These systems are often used in domains like **aerospace, defense, nuclear power, automotive, railways, and medical devices**. Because of the potential consequences, safety-critical systems require extremely high reliability, rigorous testing, formal verification methods, and compliance with strict safety standards.

Detailed Explanation of Safety-Critical Systems

1. Examples of Safety-Critical Systems:

Domain	Safety-Critical System Example	Risk If It Fails
Aviation	Flight control software	Plane crash
Automotive	Airbag deployment system	Injury or death in accident
Medical	Infusion pump software	Incorrect medication dosage
Nuclear	Reactor cooling control system	Nuclear meltdown

2. Key Characteristics:

- **High assurance level**: Must work as intended under all conditions.
- **Fail-safe design**: System must revert to a safe state if failure occurs.
- **Redundancy**: Multiple systems may perform the same task to ensure safety.

- **Verification & validation**: Must go through thorough analysis, simulation, and real-world testing.
- **Regulatory compliance**: Must follow industry-specific standards (e.g., DO-178C, ISO 26262, IEC 61508).

Typical Safety Process:

1. **Hazard identification**
2. **Risk assessment**
3. **Requirements specification**
4. **Design with safety in mind**
5. **Implementation using best practices**
6. **Verification and validation**
7. **Certification and audit**

Numerical Example

Problem Statement:
A medical ventilator has a software component that monitors oxygen levels. If oxygen levels drop below a threshold and no alarm is raised within **2 seconds**, the system is considered to have failed. Assume the following:

- The failure rate of the oxygen sensor is $\lambda = 2\times10^{-6}$ failures/hour.
- The failure rate of the software alarm module is $\lambda = 1\times10^{-6}$ failures/hour.
- The system is **series-connected**, meaning both must work for the system to work.
- Evaluate the **Mean Time between Failures (MTBF)** for the **combined safety-critical function**.
- Also, calculate the **probability of failure on demand (PFD)** for a **1-hour operation**, assuming no repairs during the hour.

Before solving this question lets understand what PFD is.

What is PFD?

PFD *stands for* ***Probability of Failure on Demand****. It measures the* ***likelihood that a safety system will fail to perform its intended function when it's needed****—typically* ***on demand****, rather than during continuous operation.*

In simpler terms:

If something goes wrong and you need the system to respond (like triggering an alarm or shutting down a machine), PFD tells you how likely it is that the system ***won't*** *work correctly at that critical moment.*

Where is PFD Used?

PFD is mainly used in ***low-demand safety systems****, such as:*

- *Emergency shutdown systems*
- *Fire suppression systems*
- *Alarm systems in medical devices*
- *Backup systems in industrial processes*

How is PFD Calculated?

For systems with ***constant failure rates*** *and* ***no repair during the time interval****, a basic approximation is:*

$$PFD \approx \lambda \times t$$

Where:

- *λ is the* ***failure rate*** *(failures per hour)*
- *t is the* ***time interval*** *during which the system might be needed (hours)*

This approximation is valid when $\lambda \times t \ll 1$, which is usually the case for safety-critical systems with ***very low failure rates****.*

Example Calculation (from earlier):

*You have a system with a **combined failure rate**:*

$\lambda_{total} = 3\times10^{-6}$ failures/hour

*You want to know the PFD over a **1-hour period**:*

$PFD = \lambda_{total} \times t = 3 \times 10^{-6} \times 1 = 3\times 10^{-6}$

*So, the **probability that the system fails when needed in 1 hour** is **0.000003**, or **0.0003%**.*

Now let's solve the previous question.

Solution:

Step 1: Combined failure rate (λtotal):
Since both components must work (series system), total failure rate:

$\lambda_{total} = \lambda_{sensor} + \lambda_{alarm} = 2\times10^{-6} + 1\times10^{-6} = 3\times10^{-6}$ failure / hour

Step 2: MTBF (Mean Time between Failures):

MTBF=1/λtotal=1/3×$^{10-6}$≈333,333 hours

Step 3: Probability of Failure on Demand (PFD) for 1 hour:

Assuming constant failure rate and short interval:

PFD $\approx \lambda_{total} \times t = 3\times10_{-6} \times 1 = 3 \times10^{-6}$

So, there's a **0.0003% chance** the system fails during a 1-hour period.

Safety Requirements and Hazard Analysis

Safety requirements are essential elements in the design, operation, and maintenance of systems, equipment, workplaces, and processes to ensure the protection of people, property, and the environment.

Definition of Safety Requirements

Safety requirements are **specifications or conditions** that a system, process, or product must meet to prevent accidents, minimize risk, and comply with laws or standards. These requirements ensure hazards are identified and mitigated through engineering, administrative, or procedural controls.

Purpose of Safety Requirements

- **Prevent harm to personnel** (injuries, illnesses, fatalities)
- **Protect the environment** (avoid contamination, pollution)
- **Safeguard equipment and assets**
- **Ensure compliance** with local, national, or international laws and regulations
- **Promote safe working conditions** and public confidence

Detailed Overview of **Software Safety Requirements**

1. Definition

Software Safety Requirements are specifications that ensure the software will function correctly and predictably in all situations that could impact safety. These requirements are intended to **prevent software-induced hazards**, **detect potential failures**, and **mitigate the impact** of software faults.

2. Importance of Software Safety Requirements

- **Prevent system-level hazards** caused by software errors
- Ensure **compliance** with safety standards (e.g., ISO 26262 for automotive, DO-178C for aviation)
- Facilitate **verification and validation (V&V)** of safety-critical software
- Reduce **operational risk** by ensuring reliable system behavior under fault conditions

3. Sources of Software Safety Requirements

Software safety requirements are typically derived from:

- **System-level hazard analysis** (e.g., FMEA, FTA, HAZOP)
- **Functional safety standards** (e.g., IEC 61508, ISO 26262, IEC 62304)
- **Safety integrity levels (SILs)**, **Automotive Safety Integrity Levels (ASIL)**, or other risk classifications
- **Interface requirements** with hardware and human operators

4. Types of Software Safety Requirements

A. Functional Safety Requirements

Describe specific behavior needed to maintain safety:

- "The software shall shut down the motor if the temperature exceeds 100°C."
- "The software shall trigger an alarm if pressure drops below 20 psi."

B. **Non-Functional Safety Requirements**

Address how the software performs under certain conditions:

- **Reliability**: "The software shall have an MTBF (Mean Time Between Failures) of at least 10,000 hours."

- **Response Time**: “The software shall respond to emergency input within 100 ms.”
- **Fault Tolerance**: “The software shall continue safe operation in the presence of a sensor failure.”

C. Design and Architectural Requirements

Ensure the software structure promotes safety:

- Use of **redundant computation paths**
- Separation of safety-critical code from non-critical code
- Use of **watchdogs** and **sanity checks**

D. **Process Requirements**

Specify how software should be developed and tested:

- Code must follow a **strict coding standard** (e.g., MISRA C)
- Safety-critical functions must be subject to **formal verification**
- All safety requirements must be **traced** from system-level hazards

5. Examples of Software Safety Requirements

Hazard	**Software Safety Requirement**
Patient overdose from infusion pump	"The software shall limit the maximum infusion rate to 25 ml/hr."
Autonomous car steering failure	"The software shall initiate a safe stop if sensor input is lost for more than 500 ms."
Industrial robot collision	"The software shall stop robot movement if a human is detected within 1 meter."

6. Life Cycle of Software Safety Requirements

1. **Hazard Analysis & Risk Assessment**
   → Identify what could go wrong.
2. **Requirements Derivation**
   → Define what the software must do (or not do) to prevent those hazards.
3. **Specification**
   → Write the requirements in clear, unambiguous, testable terms.
4. **Design & Implementation**
   → Architect the system to enforce safety behavior.
5. **Verification & Validation**
   → Prove through tests, reviews, and analysis that the software meets safety requirements.
6. **Maintenance & Change Management**
   → Ensure updates don't violate original safety intent (via regression testing, impact analysis).

7. Tools and Standards

Domain	**Standard**	**Tool Example**
Automotive	ISO 26262	Vector CAST, Polyspace
Aviation	DO-178C	SCADE, LDRA
Industrial	IEC 61508	Simulink, CodeSonar
Medical Devices	IEC 62304	IBM Rational DOORS, TortoiseSVN

8. Challenges in Implementing Software Safety Requirements

- **Ambiguity**: Poorly defined requirements lead to misinterpretation.
- **Complexity**: As systems become more complex, tracking safety logic becomes harder.
- **Change management**: Updates to software must be re-evaluated for safety.
- **Emergent behavior**: Interaction of multiple safe components might still lead to unsafe outcomes.

9. Best Practices

- Use **model-based design** and **simulation** early in development.
- Maintain strict **traceability** from system-level hazards to software requirements and test cases.
- Apply **formal methods** for high-assurance systems (e.g., theorem proving, static analysis).
- Conduct regular **safety audits and reviews**.

Hazard Analysis

Hazard analysis is a critical component of **software dependability** and **safety management**. It involves identifying potential sources of danger (hazards) that could arise from the malfunction or misuse of a software system, and determining how these hazards can be eliminated or controlled to ensure system safety and reliability.

What is Hazard Analysis?

Hazard analysis is the **systematic process of identifying, evaluating, and documenting hazards**—conditions or behaviors that could lead to accidents or undesired consequences. In the context of software systems, this means finding where and how **software behavior could lead to unsafe conditions**, especially in **safety-critical systems**.

Importance in Software Dependability and Safety Management

1. Improves Software Dependability

- Ensures **correct and predictable** software behavior under normal and abnormal conditions.
- Enables the design of **fail-safe**, **fault-tolerant**, and **robust systems**.

- Reduces **unexpected behavior** from software faults or unexpected interactions.

2. Core of Safety Management

- Forms the foundation for **safety requirements**, **mitigation strategies**, and **system design**.
- Provides traceability from **hazards** to **software requirements**, tests, and validation activities.
- Essential for **regulatory compliance** in industries like aerospace (DO-178C), automotive (ISO 26262), and medical (IEC 62304).

Key Concepts in Hazard Analysis

Term	Definition
Hazard	A potential source of harm or adverse effect
Risk	The combination of the likelihood and severity of a hazard
Cause	The initiating event or condition leading to a hazard
Mitigation	An action or control to reduce the risk associated with a hazard
Accident	An actual event where a hazard results in damage or injury

Types of Hazard Analysis Techniques

1. Preliminary Hazard Analysis (PHA)

- Conducted early in development.
- Identifies general system-level hazards.
- Helps in defining **high-level safety requirements**.

2. Failure Modes and Effects Analysis (FMEA)

- Analyzes possible **failure modes** of system components (including software).
- Determines the effect of each failure and its severity.
- Calculates **Risk Priority Number (RPN)** for prioritization.

3. Fault Tree Analysis (FTA)

- Top-down method that starts with a hazard and works backward to find causes.
- Uses logic gates to show how combinations of faults could lead to a hazard.

4. Hazard and Operability Study (HAZOP)

- Structured, qualitative method.
- Typically used in process industries.
- Explores deviations from normal operation using guidewords.

5. Software Hazard Analysis (SHA)

- Focuses specifically on software interactions.
- Identifies how software logic or failures can cause or contribute to hazards.

Process of Hazard Analysis

Step 1: System Understanding

- Define the system boundaries, interfaces, and intended functions.
- Identify all software components and their interactions with hardware and users.

Step 2: Hazard Identification

- List potential hazardous conditions related to software (e.g., incorrect output, timing errors, data corruption).
- Use checklists, past incidents, expert judgment, or modeling tools.

Step 3: Hazard Evaluation

- Assess each hazard in terms of:
  - **Severity** (how serious the outcome is)
  - **Likelihood** (how often it could happen)
- Tools: risk matrices, scoring systems

Step 4: Derivation of Safety Requirements

- Translate hazards into **software safety requirements**.
  - E.g., “The software shall shut off power if temperature exceeds threshold for more than 2 seconds.”

Step 5: Mitigation and Control

- Apply strategies like:
  - **Redundancy** (e.g., backup systems)
  - **Software constraints** (e.g., input validation, range checking)
  - **Alarms and alerts**
  - **Safe states** (e.g., software shuts down gracefully)

Step 6: Documentation and Traceability

- Record all hazards, causes, consequences, and mitigation measures.
- Link each hazard to corresponding software requirements and tests.

Example: Software Hazard Analysis in a Medical Infusion Pump

Hazard	**Cause**	**Consequence**	**Mitigation(relieve)**
Overdose	Software miscalculates infusion rate	Harm to patient or death	Limit rate in software, double-check logic, alert if rate exceeds threshold
No delivery	Timer malfunction	Treatment interruption	Watchdog timer, alert on failure, backup delivery mode

Hazard	Cause	Consequence	Mitigation(relieve)
Wrong drug	Incorrect UI input	Administration of wrong medication	Confirmation dialog, barcode scanning, double-entry verification

Role in the Software Safety Lifecycle

Hazard analysis influences every phase of the **software safety lifecycle**:

Phase	Role of Hazard Analysis
Requirements	Derives safety-related requirements
Design	Guides architectural choices (redundancy, fault isolation)
Implementation	Ensures safe coding practices and defensive programming
Verification	Validates that safety mitigations work correctly
Maintenance	Assesses impact of changes on existing hazards

Conclusion

Hazard analysis is **vital** for ensuring the **dependability** and **safety** of software systems, especially where human lives or critical infrastructure are involved. By systematically identifying and managing hazards, it allows developers and engineers to:

- Proactively design out risks
- Build safer software systems
- Comply with international safety standards
- Increase user and regulatory trust

Safety Engineering Processes: Design, Implementation, Testing

Software is the set of instructions in the form of programs to govern the computer system and to process the hardware components. To produce a software product the set of activities is used. This set is called a software process.

What are Software Processes?

Software processes in software engineering refer to the methods and techniques used to develop and maintain software. Some examples of software processes include:

- **Waterfall**: a linear, sequential approach to software development, with distinct phases such as requirements gathering, design, implementation, testing, and maintenance.
- **Agile**: a flexible, iterative approach to software development, with an emphasis on rapid prototyping and continuous delivery. Based on Software Manifesto, a set of principles for software development that prioritize individuals and interactions, working software, customer collaboration and responding to change.
- **Scrum**: a popular Agile methodology that emphasizes teamwork, iterative development, and a flexible, adaptive approach to planning and management.
- **DevOps**: a set of practices that aims to improve collaboration and communication between development and operations teams, with an emphasis on automating the software delivery process.

SAFETY ENGINEERING PROCESSES

1. Design Phase

The design phase aims to create an architecture and components that minimize risk and support fault detection, containment, and recovery.

Key Concepts:

- **Hazard Analysis**
  - Identify potential hazards that the software might introduce or fail to mitigate.
  - Techniques: FMEA (Failure Modes and Effects Analysis), HAZOP, Fault Tree Analysis (FTA).
- **Safety Requirements Specification**
  - Define functional and non-functional safety requirements.

    - Include constraints (e.g., real-time deadlines, input validation) and safe states.
- **Design for Safety**
    - Use **safety patterns** and **defensive design**.
    - Apply **redundancy** (hardware/software), **diversity** (e.g., N-version programming), and **partitioning** (to isolate faults).
- **Formal Methods**
    - Mathematically prove that a design satisfies safety properties.
    - Examples: model checking, theorem proving (e.g., using TLA+, Z).
- **Safe State Design**
    - Ensure the system enters a safe state upon failure (fail-safe or fail-operational).
- **Separation of Concerns**
    - Use architectural styles like layered architecture to isolate safety-critical components from non-critical ones.

2. Implementation Phase

This phase turns design into code, with a focus on preserving safety goals and minimizing the introduction of errors.

Key Concepts:

- **Safe Programming Languages and Practices**
    - Use type-safe, memory-safe languages (e.g., Ada, SPARK, MISRA C).
    - Avoid unsafe constructs (e.g., pointer arithmetic in C).
- **Coding Standards**
    - Enforce guidelines that reduce fault likelihood (e.g., MISRA, CERT).
- **Static Analysis**
    - Analyze source code without executing it to detect bugs, vulnerabilities, or violations of coding standards.

- **Fault Containment & Error Handling**
  - Implement robust exception handling.
  - Ensure the software can gracefully degrade or switch to redundant systems.
- **Modularization and Encapsulation**
  - Isolate faults to prevent propagation.
- **Traceability**
  - Maintain traceability from requirements → design → code → tests.

3. Testing and Verification Phase

The goal is to validate that the software meets its safety requirements and behaves correctly under all (even faulted) conditions.

Key Concepts:

- **Verification vs. Validation**
  - *Verification*: Are we building the product right?
  - *Validation*: Are we building the right product?
- **Safety Testing Techniques**
  - **Unit Testing**: Isolated testing of components.
  - **Integration Testing**: Ensure safe interactions between modules.
  - **System Testing**: Ensure overall system behavior under normal and abnormal conditions.
  - **Fault Injection Testing**: Introduce faults to test fault tolerance.
  - **Stress and Load Testing**: Check behavior under high demand.
  - **Boundary Testing**: Examine edge cases where failures are likely.
- **Coverage Analysis**
  - **Code coverage** (e.g., statement, branch, path).
  - **Requirements coverage**: Are all safety requirements tested?

- **Model-Based Testing**
  - Derive test cases from formal models of system behavior.
- **Regression Testing**
  - Ensure new changes don't compromise safety.
- **Safety Audits and Reviews**
  - Conduct independent reviews of code, documentation, and test results.
- **Certification Testing**
  - Meet industry-specific safety standards (e.g., DO-178C for avionics, ISO 26262 for automotive, IEC 61508 for industrial control).

Safety Cases and Their Role in Certification

In the context of **software dependability**, especially in **safety-critical systems**, **safety cases** play a vital role in demonstrating that software is acceptably safe and supporting its **certification** by regulatory bodies. Here's a breakdown of their role and how they support certification:

What is a Safety Case?

A **safety case** is a structured argument, supported by evidence, intended to justify that a system is safe for a given application in a given environment.

It typically includes:

1. **Claims**: Statements about the system's safety (e.g., "System X will not cause harm under Y conditions").
2. **Arguments**: The reasoning that links the evidence to the claims (e.g., logical decomposition, fault-tree analysis).
3. **Evidence**: Artifacts supporting the claims (e.g., test results, formal verification, inspections, hazard analyses).

This structure is often visualized using tools like **Goal Structuring Notation (GSN)**

Role of Safety Cases in Certification

1. **Demonstrate Compliance**:
   Safety cases provide the evidence needed to show compliance with relevant **safety standards** (e.g., ISO 26262 for automotive, DO-178C for aviation, IEC 61508 for industrial systems).
2. **Support Regulatory Review**:
   Certification bodies (like the FAA, TÜV, or national rail authorities) review safety cases to assess whether the system meets safety requirements. A well-structured case makes this easier.
3. **Encourage Rigorous Development**:
   The process of building a safety case enforces discipline in **hazard identification**, **risk assessment**, and **mitigation planning** throughout the software development lifecycle.
4. **Manage Complexity**:
   In large systems, safety cases help structure complex safety arguments and trace them back to individual components or subsystems.
5. **Enable Change Management**:
   When systems are updated, the safety case can be modified to show that safety is still maintained. This is critical for maintaining certification during product evolution.

Connection to Software Dependability

Software dependability encompasses attributes like **reliability**, **safety**, **security**, and **maintainability**. Safety cases focus specifically on **safety**, but their structured approach and reliance on evidence-based arguments contribute to broader dependability by:

- Ensuring rigorous verification and validation.
- Promoting traceability from requirements to implementation.
- Identifying and mitigating failure modes.

Example: Automotive Software

In **automotive systems** governed by **ISO 26262**, the safety case might include:

- Functional safety requirements.
- Software architecture adhering to ASIL(Automotive Safety Integrity Level).
- Evidence from unit, integration, and system testing.
- Analysis of safety mechanisms (e.g., watchdogs, memory protection).
- Results of failure mode and effects analysis (FMEA).

The safety case is submitted during the **confirmation review** process and is essential for **certification** and market approval.

Summary

Role	**Description**
Justification	Provides structured argument that system is acceptably safe
Compliance	Supports certification to safety standards
Traceability	Connects safety requirements to implementation and testing

Role	**Description**
Lifecycle Support	Helps manage safety through changes and updates
Audit Facilitation	Organizes evidence for external review and approval