SWE.1-3 Software Engineering

Process Group

System Engineering vs. Software Engineering

System engineering focuses on the entire system, while software engineering specializes in the software components

1. Scope of Work

- **System Engineering:** Encompasses hardware, software, processes, and people.
- **Software Engineering:** Focuses exclusively on software development and lifecycle.
- **Overlap:** Software is a subsystem within the overall system.

2. Requirements and Design

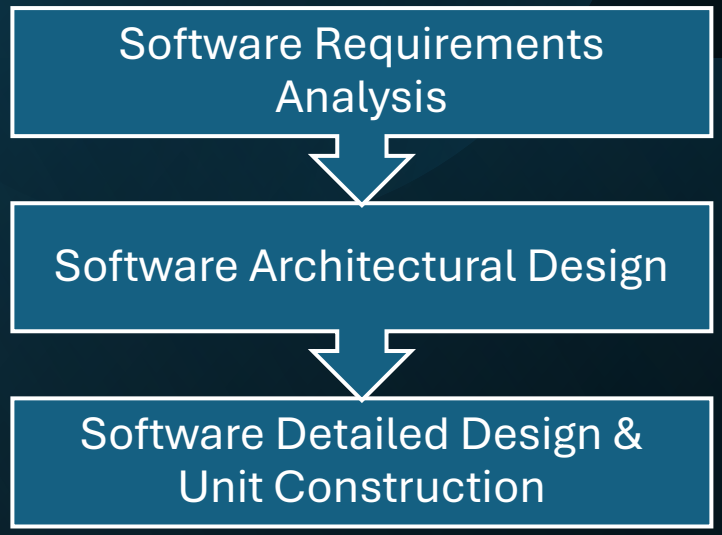
- **System Engineering:** Defines high-level requirements and system architecture.
- **Software Engineering:** Derives software-specific requirements and architecture.
- **Collaboration:** Software engineering relies on inputs from system engineering.

3. Verification and Validation

- **System Engineering:** Validates the complete system against stakeholder needs.
- **Software Engineering:** Verifies software against its requirements.
- **Dependency:** System validation includes the results of software verification.

Automotive **S**oftware **P**rocess **I**mprovement and **C**apability d**E**termination

Overview of the SW Engineering Process Group



SWE process group supports software engineering, from requirements analysis to verification at all levels

1. Purpose of SWE Processes

- Ensures traceable software development from requirements to implementation.
- Helps align software deliverables with system-level requirements.
- Improves quality and consistency in software engineering practices.

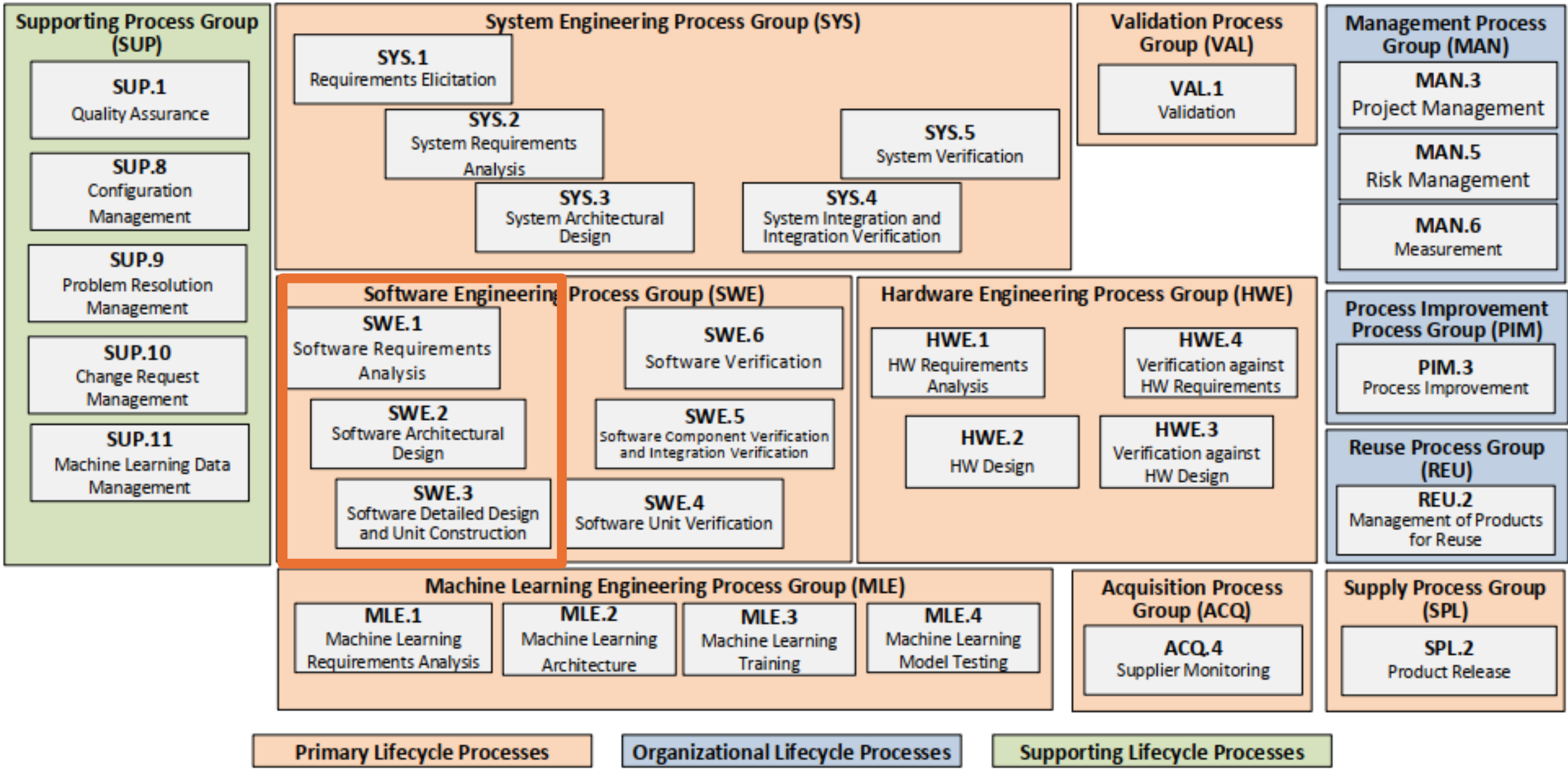
2. Integration with ASPICE Framework

- Establishes compliance with ASPICE process requirements.
- Links software development to system engineering processes.
- Facilitates continuous improvement through structured activities

3. Focus Areas of SWE

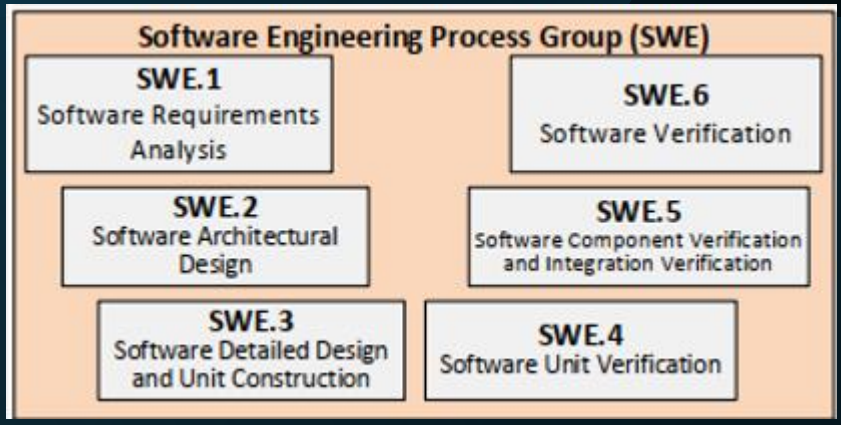
- SWE.1: Software Requirements Analysis to derive and refine software requirements.
- SWE.2: Software Architectural Design to define software components and interactions.
- SWE.3: Software Detailed Design & Unit Construction for implementing software units.
- SWE.4: Software Unit Verification focuses on verifying individual software units.
- SWE.5: Software Component and Integration Verification validates the integration of components.
- SWE.6: Software Verification ensures system-wide verification against requirements.

Automotive **S**oftware **P**rocess **I**mprovement and **C**apability d**E**termination



ASPICE – **A**utomotive **S**oftware **P**rocess **I**mprovement and **C**apability d**E**termination

Purpose and Scope of the SWE Process Group



SWE process group, emphasize its integration with the overall ASPICE lifecycle

1. Purpose of the SWE Process Group

- Establish a structured approach to software development.
- Ensure traceability from requirements to implementation and verification.
- Align software development processes with ASPICE compliance goals.

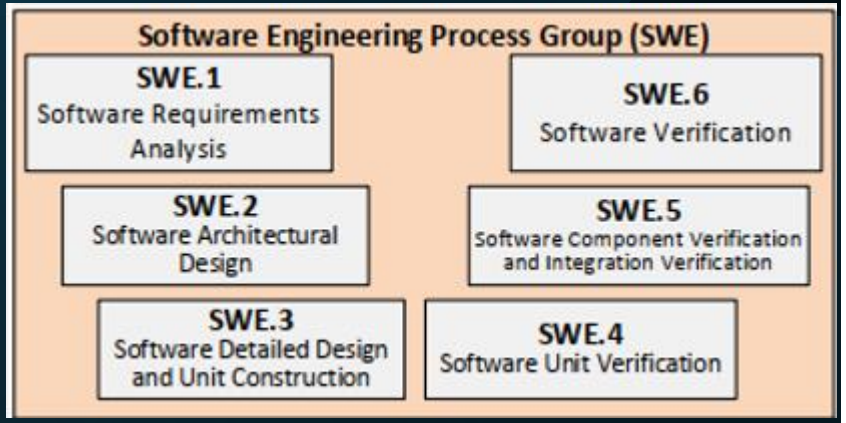
2. Scope of the SWE Processes

- Covers requirements analysis, architectural design, and detailed design.
- Includes unit, integration, and system-level verification activities.
- Applies to software development in safety-critical and non-safety-critical systems.

3. Connection to ASPICE Framework

- Supports system engineering through traceable software artifacts.
- Bridges the gap between system requirements and detailed implementation.
- Facilitates collaboration across disciplines and teams.

ASPICE SW Engineering Processes Overview



Relationships between SWE processes, illustrating how development outputs flow into verification activities

1. SWE Development Processes

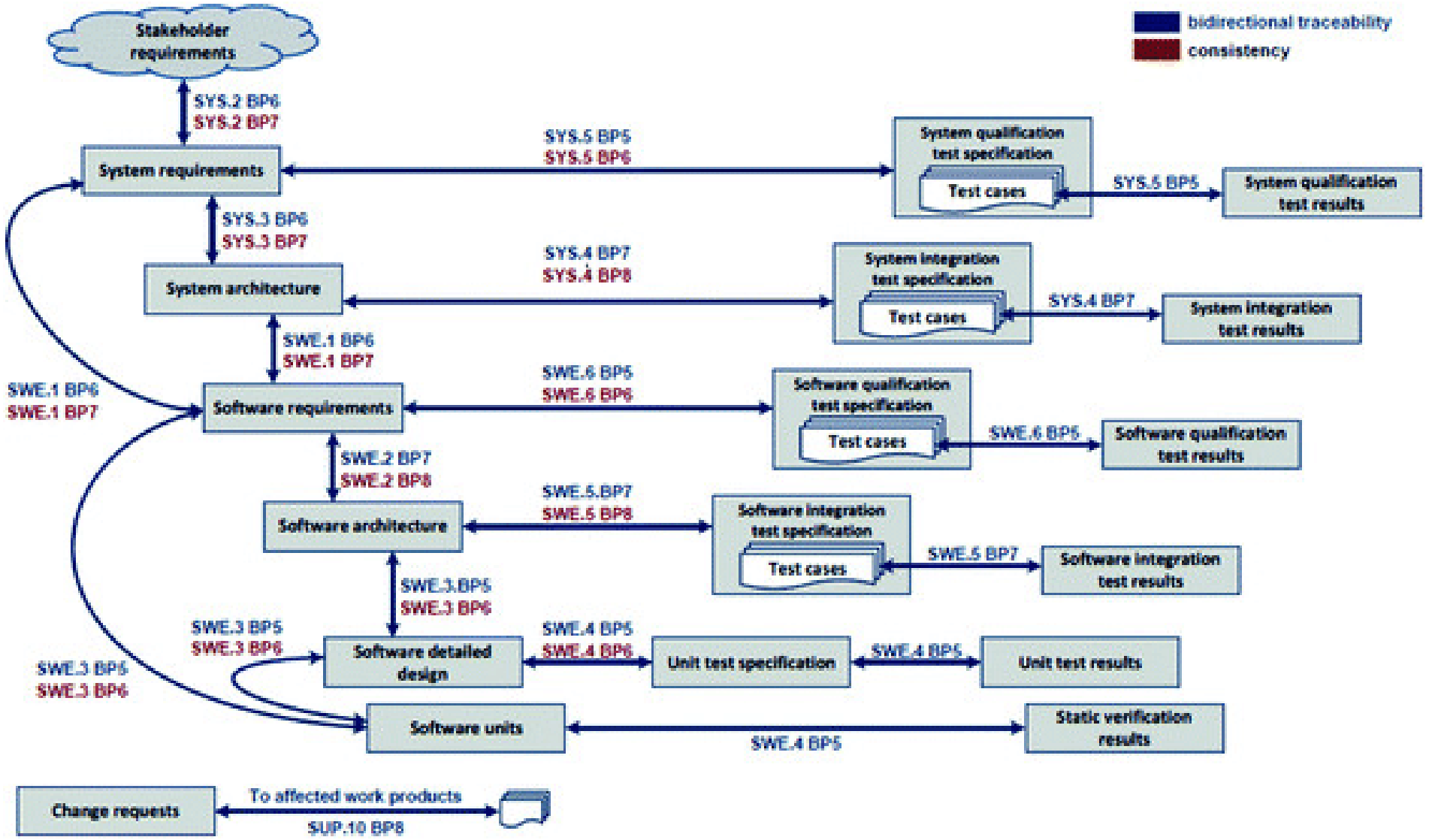
- SWE.1: Software Requirements Analysis focuses on defining and refining software requirements.
- SWE.2: Software Architectural Design defines software structure and interactions.
- SWE.3: Software Detailed Design & Unit Construction deals with detailed implementation.

2. SWE Verification Processes

- SWE.4: Software Unit Verification ensures individual software units meet design specifications.
- SWE.5: Software Component and Integration Verification validates the integration of components.
- SWE.6: Software Verification confirms compliance of the entire software system with requirements.

3. Interconnection and Traceability

- Each SWE process feeds into the next, ensuring traceability.
- Development processes provide inputs for corresponding verification activities.
- All SWE processes link back to system-level requirements and architecture.



ASPICE – **A**utomotive **S**oftware **P**rocess **I**mprovement and **C**apability d**E**termination

System Requirements vs. Software Requirements

System requirements define overall system functionality, while software requirements focus on software contributions to the system

1. Definition and Scope

- **System Requirements:** Describe overall system behavior, including hardware, software, and operational environment.
- **Software Requirements:** Detail what the software must do within the system.
- **Dependency:** Software requirements are derived from system requirements.

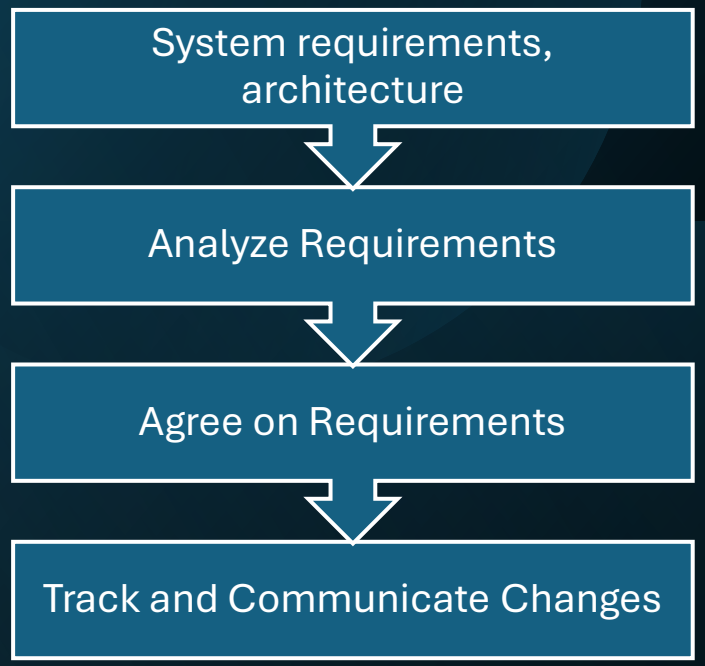
2. Examples of Requirements

- **System Requirement:** "The system shall monitor and display environmental conditions.“
- **Software Requirement:** "The software shall process sensor data to calculate temperature averages.“
- **Traceability:** Software requirements trace back to their corresponding system requirements

3. Non-Functional Focus

- **System Requirements:** Address overall system constraints (e.g., safety, reliability).
- **Software Requirements:** Focus on software-specific constraints (e.g., performance, usability).
- **Alignment:** Both contribute to meeting stakeholder needs and constraints.

SWE.1 Software Requirements Analysis Overview



SWE.1 transforms system-level inputs into structured and traceable software requirements

1. Purpose of Software Requirements Analysis

- Establish a structured and analyzed set of software requirements.
- Ensure software requirements are consistent with system requirements and architecture.
- Provide a foundation for software architectural design.

2. Process Scope

- Includes specification, prioritization, and structuring of software requirements.
- Analyzes requirements for correctness and feasibility.
- Evaluates the impact of software requirements on the operating environment.

3. Expected Outcomes

- Software requirements are specified, structured, and prioritized.
- Consistency and traceability are established between system and software requirements.
- Agreed-upon requirements are communicated to all stakeholders.

Process ID
SWE.1
Process name
Software Requirements Analysis
Process purpose
The purpose is to establish a structured and analyzed set of software requirements consistent with the system requirements, and the system architecture.
Process outcomes
1) Software requirements are specified. 2) Software requirements are structured and prioritized. 3) Software requirements are analyzed for correctness and technical feasibility. 4) The impact of software requirements on the operating environment is analyzed. 5) Consistency and bidirectional traceability are established between software requirements and system requirements. 6) Consistency and bidirectional traceability are established between software requirements and system architecture. 7) The software requirements are agreed and communicated to all affected parties.

SWE.1 Software Requirements Analysis	Outcome 1	Outcome 2	Outcome 3	Outcome 4	Outcome 5	Outcome 6	Outcome 7
Output Information Items							
17-00 Requirement	X	X					
17-54 Requirement Attribute		X					
15-51 Analysis Results			X	X			
13-51 Consistency Evidence					X	X	
13-52 Communication Evidence							X
Base Practices							
BP1: Specify software requirements	X						
BP2: Structure software requirements		X					
BP3: Analyze software requirements			X				
BP4: Analyze the impact on the operating environment				X			
BP5: Ensure consistency and establish bidirectional traceability					X	X	
BP6: Communicate agreed software requirements and impact on the operating environment							X

Base practices

SWE.1.BP1: Specify software requirements. Use the system requirements and the system architecture to identify and document the functional and non-functional requirements for the software according to defined characteristics for requirements.

NOTE 1: Characteristics of requirements are defined in standards such as ISO IEEE 29148, ISO 26262-8:2018, or the INCOSE Guide for Writing Requirements.

SWE.1.BP2: Structure software requirements. Structure and prioritize the software requirements.

Note 5: Examples for structuring criteria can be grouping (e.g. by functionality) or variants identification.

Note 6: Prioritization can be done according to project or stakeholder needs via e.g. definition of release scopes. Refer to SPL.2.BP1.

SWE.1.BP3: Analyze software requirements. Analyze the specified software requirements including their interdependencies to ensure correctness, technical feasibility, and to support project management regarding project estimates.

Note 7: See MAN.3.BP3 for project feasibility and MAN.3.BP5 for project estimates.

Note 8: Technical feasibility can be evaluated based on e.g. platform or product line, or by prototyping.

SWE.1.BP4: Analyze the impact on the operating environment. Analyze the impact that the software requirements will have on elements in the operating environment.

SWE.1.BP5: Ensure consistency and establish bidirectional traceability. Ensure consistency and establish bidirectional traceability etween software requirements and system architecture. Ensure consistency and establish bidirectional traceability between software requirements and system requirements.

Note 9: Redundant traceability is not intended.

Note 10: There may be non-functional system requirements that the software requirements do not trace to. Examples are process requirements or requirements related to later software product lifecycle phases such as incident handling. Such requirements are still subject to verification.

Note 11: Bidirectional traceability supports consistency, and facilitates impact analysis of change requests, and demonstration of verification coverage

Note 12: In case of software development only, the system requirements and system architecture refer to a given operating environment. In that case, consistency and bidirectional traceability can be ensured between stakeholder requirements and software requirements.

SWE.1.BP6: Communicate agreed software requirements and impact on the operating environment. Communicate the agreed software requirements, and the results of the analysis of impact on the operating environment, to all affected parties.

BP1: Specify Software Requirements

Define Software Requirements

Establish Requirement Characteristics

Document Requirements

System requirements and architecture serve as inputs for defining actionable software requirements

Base practices

SWE.1.BP1: Specify software requirements. Use the system requirements and the system architecture to identify and document the functional and non-functional requirements for the software according to defined characteristics for requirements.

NOTE 1: Characteristics of requirements are defined in standards such as ISO IEEE 29148, ISO 26262-8:2018, or the INCOSE Guide for Writing Requirements.

Note 2: Examples for defined characteristics of requirements shared by technical standards are verifiability (i.e. verification criteria being inherent in the requirements formulation), unambiguity/comprehensibility, freedom from design and implementation, and not contradicting any other requirement.

Note 3: In case of software-only development, the system requirements and the system architecture refer to a given operating environment. In that case, stakeholder requirements can be used as the basis for identifying the required functions and capabilities of the software.

Note 4: The hardware-software-interface (HSI) definition puts in context hardware and therefore is an interface decision at the system design level (see SYS.3). If such a HSI exists, then it may provide input to software requirements.

1. Define Software Requirements

- Derive software requirements from system requirements and architecture.
- Identify functional and non-functional requirements, including performance and security.
- Ensure requirements are verifiable, complete, and unambiguous.

2. Establish Requirement Characteristics

- Align software requirements with system-level constraints and capabilities.
- Address usability, performance, scalability, and interoperability considerations.
- Incorporate stakeholder feedback and resolve ambiguities.

3. Document Requirements

- Use a structured format to document requirements (e.g., DOORS, Confluence).
- Categorize requirements by type (functional, performance, safety).
- Maintain consistency across requirement documents.

Requirement ID	Requirements definition	Type
SW-REQ-001	**[The SW Component shall] authenticate users by validating their credentials against the SQL database within 2 seconds.**	Functional
SW-REQ-002	**[The SW Component shall] encrypt all HTTP request payloads using the AES-256 algorithm before sending them to the server.**	Non-Functional (Security)
SW-REQ-003	**[The SW Component shall] display a 'login failed' message in the GUI if the authentication API returns an error.**	Functional
SW-REQ-004	**[The SW Component shall] limit memory allocation to 500 MB during data caching operations under normal conditions.**	Non-Functional (Performance)
SW-REQ-005	**[The SW Component shall] handle up to 100 simultaneous API requests without exceeding a response time of 200 milliseconds.**	Non-Functional (Scalability)
SW-REQ-006	**[The SW Component shall] allow administrators to export user activity logs to a CSV file via the administration panel.**	Functional
SW-REQ-007	**[The SW Component shall] integrate with the Stripe API to process payment transactions within 3 seconds.**	Functional
SW-REQ-008	**[The SW Component shall] comply with WCAG 2.1 guidelines by providing screen reader support for all UI components.**	Non-Functional (Usability)

BP1 Examples: Requirements definition

BP2: Structure software Requirements

Requirements are prioritized, ensuring critical needs are addressed early while balancing effort and feasibility

SWE.1.BP2: Structure software requirements. Structure and prioritize the software requirements.

Note 5: Examples for structuring criteria can be grouping (e.g. by functionality) or variants identification.

Note 6: Prioritization can be done according to project or stakeholder needs via e.g. definition of release scopes. Refer to SPL.2.BP1.

1. Group Requirements by Criteria

- Organize requirements by functionality, system modules, or lifecycle phases.
- Use grouping to identify dependencies and overlaps.
- Maintain logical relationships between requirements for traceability.

2. Prioritize Requirements Based on Impact

- Assign priority levels based on business value, technical feasibility, and risk.
- Address critical requirements early in the project timeline.
- Reassess prioritization as new requirements emerge or constraints change.

3. Document Requirements Structure and Priority

- Maintain a requirements hierarchy (e.g., high-level to detailed requirements).
- Use tools like DOORS or Jira for dynamic organization.
- Ensure stakeholders approve and understand prioritization criteria.

Category	Requirement (MASTER Template)	Impact	Effort	Priority
Critical and Low-Effort Tasks	**[The system shall] provide user authentication functionality**	High	Low	1
Critical and Low-Effort Tasks	**[The system shall] display error messages for invalid inputs**	High	Low	2
High-Impact and High-Effort Work	**[The system shall] synchronize data in real-time across platforms**	High	High	3
High-Impact and High-Effort Work	**[The system shall] refactor the legacy database module for performance**	High	High	4
Low-Effort and Low-Impact Tasks	**[The system shall] display tooltips to improve UI usability**	Low	Low	5
Low-Effort and Low-Impact Tasks	**[The system shall] update the footer styling on all pages**	Low	Low	6
Optional and High-Effort Work	**[The system shall] provide an advanced analytics dashboard**	Low	High	7
Optional and High-Effort Work	**[The system shall] offer customizable themes for the user interface**	Low	High	8

BP2 Examples: Requirements Prioritization Matrix

BP3: Analyze Software Requirements

Process of analyzing requirements to ensure they are valid, feasible, and aligned with system needs

SWE.1.BP3: Analyze software requirements. Analyze the specified software requirements including their interdependencies to ensure correctness, technical feasibility, and to support project management regarding project estimates.

Note 7: See MAN.3.BP3 for project feasibility and MAN.3.BP5 for project estimates.

Note 8: Technical feasibility can be evaluated based on e.g. platform or product line, or by prototyping.

1. Validate Requirements Against Inputs

- Ensure alignment with system requirements and system architecture.
- Verify that requirements meet quality criteria: complete, clear, consistent, and unambiguous.
- Identify any gaps or conflicts in requirements.

2. Assess Feasibility and Constraints

- Evaluate technical feasibility, including computational and architectural constraints.
- Conduct prototyping or simulations for high-risk requirements.
- Address scalability, performance, and safety considerations.

3. Analyze Dependencies and Risks

- Define a process for managing requirement changes after agreement.
- Use Jira workflows to track approved changes and their impact.
- Communicate changes to all teams involved to avoid misalignment.

BP4: Analyze the Impact on the Operating Environment

Software requirements influence and are influenced by the operating environment, emphasizing compatibility and risk mitigation

SWE.1.BP4: Analyze the impact on the operating environment. Analyze the impact that the software requirements will have on elements in the operating environment.

1. Assess Environmental Compatibility 🌐

- Evaluate how software requirements align with hardware and network constraints.
- Identify compatibility issues with the existing system architecture.
- Ensure requirements consider the operating conditions (e.g., temperature, power usage).

2. Evaluate System-Wide Impacts ⚙️

- Analyze how software behavior affects other components in the system.
- Address issues such as shared resource usage, latency, and communication protocols.
- Propose adjustments to minimize adverse system-wide impacts.

3. Identify Constraints and Risks 🚦

- Highlight environmental constraints (e.g., memory limits, data throughput).
- Assess risks associated with external dependencies like third-party APIs or hardware.
- Develop mitigation strategies for identified risks.

BP5: Ensure Consistency and Traceability

Software requirements maintain traceability to system-level inputs and related project artifacts, ensuring consistency

SWE.1.BP5: Ensure consistency and establish bidirectional traceability. Ensure consistency and establish bidirectional traceability etween software requirements and system architecture. Ensure consistency and establish bidirectional traceability between software requirements and system requirements.

Note 9: Redundant traceability is not intended.

Note 10: There may be non-functional system requirements that the software requirements do not trace to. Examples are process requirements or requirements related to later software product lifecycle phases such as incident handling. Such requirements are still subject to verification.

Note 11: Bidirectional traceability supports consistency, and facilitates impact analysis of change requests, and demonstration of verification coverage

Note 12: In case of software development only, the system requirements and system architecture refer to a given operating environment. In that case, consistency and bidirectional traceability can be ensured between stakeholder requirements and software requirements.

1. Establish Traceability Links

- Link each requirement to its corresponding system requirement or architectural element.
- Maintain traceability across development phases, from requirements to testing.
- Use tools like DOORS, Jira, or Polarion to manage traceability matrices.

2. Verify Requirement Consistency

- Ensure requirements are consistent with each other and with system-level requirements.
- Address and resolve conflicts or overlaps between requirements.
- Regularly review requirements for alignment with project goals.

3. Maintain Updated Traceability

- Update traceability links as requirements evolve during the project lifecycle.Ensure traceability is maintained through version control and change management.Validate traceability as part of project reviews and audits.

BP6: Communicate Agreed Software Requirements and Impact on the Operating Environment

Communication of requirements, including operating environment constraints, to stakeholders.

> **SWE.1.BP6: Communicate agreed software requirements and impact on the operating environment.** Communicate the agreed software requirements, and the results of the analysis of impact on the operating environment, to all affected parties.

1. Develop a Communication Plan

- Share software requirements and environmental impacts through reports or presentations.
- Tailor communication to the audience, highlighting key constraints and conditions.
- Schedule regular reviews to align on updates and changes.

2. Engage Key Stakeholders

- Involve stakeholders in discussions on requirements and environmental compatibility.
- Use feedback to address ambiguities and improve alignment.
- Highlight risks or challenges in the operating environment.

3. Maintain Documentation Transparency

- Use platforms like Confluence to share and update requirements.
- Ensure traceability and version control for all changes.
- Provide summaries of requirements and their environmental impacts for clarity.

Challenges in SWE.1 - Software Requirements Analysis

Major challenges in requirements elicitation and provides key areas to focus on for overcoming them

1. Unclear or Incomplete Requirements

- System requirements lack clarity or details for deriving software requirements.
- Stakeholders provide inconsistent or conflicting inputs.
- Missing edge cases or non-functional requirements in the system specifications.

2. Alignment Issues with System Architecture

- Software requirements do not align with system-level architecture constraints.
- Misinterpretation of architectural inputs leads to incompatible software definitions.
- Lack of traceability between system and software requirements.

3. Managing Stakeholder Expectations

- Misalignment of priorities between stakeholders and the development team.
- Frequent changes in requirements disrupt analysis and prioritization processes.
- Ineffective communication leads to misunderstandings or missed goals.

System Architectural Design vs. Software Architectural Design

System architecture provides the overall structure, while software architecture defines the software's role and internal organization

1. Scope and Purpose 📐

- **System Architectural Design:** Defines the overall system structure, including hardware and software components.
- **Software Architectural** Design: Specifies the software's internal structure and interactions.
- **Dependency:** Software architecture is a subset derived from system architecture.

2. Focus Areas 🌐

- **System Architectural Design:** Addresses system-level constraints, interactions, and integration.
- **Software Architectural Design:** Focuses on software components, modules, and interfaces.
- **Overlap:** Software architecture must align with system-level interfaces and constraints.

3. Verification and Traceability ✅

- **System Architectural Design:** Ensures system-level consistency and stakeholder alignment.
- **Software Architectural Design:** Provides traceability to system requirements and inputs.
- **Integration:** System validation includes verification of software architecture.

SWE.2 - Software Architectural Design Overview

SWE.2 transforms software requirements into an actionable architectural design, ensuring alignment with system-level inputs

1. Purpose of Software Architectural Design

- Define the software architecture, including static and dynamic aspects.
- Ensure the architecture aligns with software requirements and system architecture.
- Provide a foundation for detailed design and unit construction.

2. Process Scope

- Specify software elements, their interactions, and interfaces.
- Analyze architectural decisions for feasibility and consistency.
- Address constraints such as performance, security, and scalability

3. Expected Outcomes

- A documented and analyzed software architecture.
- Traceability established between software architecture and requirements.
- Architectural decisions communicated to all stakeholders.

Process ID

SWE.2

Process name

Software Architectural Design

Process purpose

The purpose is to establish an analyzed software architecture consistent with the software requirements.

Process outcomes

1) A software architecture is designed including static and dynamic aspects.
2) The software architecture is analyzed against defined criteria.
3) Consistency and bidirectional traceability are established between software architecture and software requirements.
4) The software architecture is agreed and communicated to all affected parties.

SWE.2 Software Architectural Design	Outcome 1	Outcome 2	Outcome 3	Outcome 4
Output Information items				
04-04 Software Architecture	X			
13-51 Consistency Evidence			X	
13-52 Communication Evidence				X
15-51 Analysis Results		X		
05-01 Resource Consumption Targets		X		
Base Practices				
BP1: Specify static aspects of software architecture	X			
BP2: Specify dynamic aspects of software architecture	X			
BP3: Analyze software architecture		X		
BP4: Ensure consistency and establish bidirectional traceability			X	
BP5: Communicate agreed software architecture				X

Base practices

SWE.2.BP1: Specify static aspects of the software architecture. Specify and document the static aspects of the software architecture with respect to the functional and non-functional software requirements, including external interfaces and a defined set of components with their interfaces and relationships.

Note 1: The hardware-software-interface (HIS) definition puts in context the hardware design and therefore is an aspect of system design (SYS.3).

SWE.2.BP2: Specify dynamic aspects of the software architecture. Specify and document the dynamic aspects of the software architecture with respect to the functional and non-functional software requirements, including the behavior of the components and their interaction in different system modes, and concurrency aspects.

Note 2: Examples for concurrency aspects are application-relevant interrupt handling, preemptive processing, multi-threading.

Note 3: Examples for behavioral descriptions are natural language or semi-formal notation (e.g. SysML, UML).

SWE.2.BP3: Analyze software architecture. Analyze the software architecture regarding relevant technical design aspects and to support project management regarding project estimates. Document a rationale for the software architectural design decision.

Note 4: See MAN.3.BP3 for project feasibility and MAN.3.BP5 for project estimates.

Note 5: The analysis may include the suitability of pre-existing software components for the current application.

Note 6: Examples of methods suitable for analyzing technical aspects are prototypes, simulations, qualitative analyses.

Note 7: Examples of technical aspects are functionality, timings, and resource consumption (e.g. (ROM, RAM, external / internal EEPROM or Data Flash or CPU load).

Note 8: Design rationales can include arguments such as proven-in-use, reuse of a software framework or software product line, a make-or-buy decision, or found in an evolutionary way (e.g. set-based design).

SWE.2.BP4: Ensure consistency and establish bidirectional traceability. Ensure consistency and establish bidirectional traceability between the software architecture and the software requirements.

Note 9: There may be non-functional software requirements that the software architectural design does not trace to. Examples are development process requirements. Such requirements are still subject to verification.

Note 10: Bidirectional traceability supports consistency, and facilitates impact analysis of change requests, and demonstration of verification coverage

SWE.2.BP5: Communicate agreed software architecture. Communicate the agreed software architecture to all affected parties.

BP1: Specify Static Aspects of the Software Architecture

Static structure of the software architecture, highlighting relationships and key interfaces between components

Base practices

SWE.2.BP1: Specify static aspects of the software architecture. Specify and document the static aspects of the software architecture with respect to the functional and non-functional software requirements, including external interfaces and a defined set of components with their interfaces and relationships.

Note 1: The hardware-software-interface (HIS) definition puts in context the hardware design and therefore is an aspect of system design (SYS.3).

1. Define Software Components and Modules

- Identify major software components and their roles in the system.
- Specify modules, libraries, or layers used in the architecture.
- Ensure component definitions align with software requirements.

2. Model Static Structures

- Create static diagrams like class diagrams or block diagrams.
- Show relationships between components, such as inheritance or dependencies.
- Highlight interfaces and key data flows between components.

3. Align with System Architecture

- Verify that the static architecture is consistent with system-level architecture.
- Address system constraints, such as hardware and platform limitations.
- Ensure traceability between static software structures and system elements.

BP2: Specify Dynamic Aspects of the Software Architecture

Dynamic interactions of software components while a user-triggered event, ensuring alignment with system use cases

> **SWE.2.BP2: Specify dynamic aspects of the software architecture.** Specify and document the dynamic aspects of the software architecture with respect to the functional and non-functional software requirements, including the behavior of the components and their interaction in different system modes, and concurrency aspects.
>
> *Note 2: Examples for concurrency aspects are application-relevant interrupt handling, preemptive processing, multi-threading.*
>
> *Note 3: Examples for behavioral descriptions are natural language or semi-formal notation (e.g. SysML, UML).*

1. Define Component Interactions

- Specify interactions between software components under various scenarios.
- Include data exchange, control flow, and event triggers.
- Use runtime behavior to validate alignment with functional requirements.

2. Model Dynamic Behavior

- Create diagrams like sequence diagrams, state diagrams, or activity diagrams.
- Illustrate key processes such as message passing or system responses.
- Address scenarios like error handling and concurrency.

3. Align with Use Cases and Requirements

- Ensure dynamic behavior supports system use cases and software requirements.
- Validate that interactions handle both typical and edge cases.
- Update models as new scenarios or requirements emerge.

BP3: Analyze the Software Architecture

Analysis of architectural risks, highlighting areas requiring immediate attention and mitigation strategies

SWE.2.BP3: Analyze software architecture. Analyze the software architecture regarding relevant technical design aspects and to support project management regarding project estimates. Document a rationale for the software architectural design decision.

Note 4: See MAN.3.BP3 for project feasibility and MAN.3.BP5 for project estimates.

Note 5: The analysis may include the suitability of pre-existing software components for the current application.

Note 6: Examples of methods suitable for analyzing technical aspects are prototypes, simulations, qualitative analyses.

Note 7: Examples of technical aspects are functionality, timings, and resource consumption (e.g. (ROM, RAM, external / internal EEPROM or Data Flash or CPU load).

Note 8: Design rationales can include arguments such as proven-in-use, reuse of a software framework or software product line, a make-or-buy decision, or found in an evolutionary way (e.g. set-based design).

1. Assess Architectural Feasibility

- Verify that the architecture meets functional and non-functional requirements.
- Evaluate the feasibility of implementing the architecture within system constraints.
- Use tools or simulations to assess performance and scalability.

2. Identify and Mitigate Risks

- Detect architectural risks such as bottlenecks, single points of failure, or incompatibilities.
- Propose mitigation strategies to address identified risks.
- Include safety, security, and maintainability considerations in the analysis.

3. Ensure Architectural Consistency

- Verify consistency with system architecture and requirements.
- Ensure that architectural elements are compatible and align with project standards.
- Update the architecture to address findings from reviews or testing feedback.

BP4: Ensure Consistency and Traceability

Traceability links align software architecture with requirements and higher-level design artifacts.

SWE.2.BP4: Ensure consistency and establish bidirectional traceability. Ensure consistency and establish bidirectional traceability between the software architecture and the software requirements.

Note 9: There may be non-functional software requirements that the software architectural design does not trace to. Examples are development process requirements. Such requirements are still subject to verification.

Note 10: Bidirectional traceability supports consistency, and facilitates impact analysis of change requests, and demonstration of verification coverage

1. Establish Bidirectional Traceability

- Create traceability links between software architecture elements and softearevrequirements.
- Maintain links from system architecture to software components.
- Use tools like DOORS, Jira, or SysML to manage traceability matrices.

2. Verify Consistency Across Artifacts

- Ensure architectural diagrams, requirements, and models align with project standards.
- Check for gaps, overlaps, or conflicts between architectural elements.
- Validate alignment with system constraints and requirements.

3. Update and Validate Traceability

- Update traceability links as the architecture evolves.
- Perform periodic reviews to validate that traceability is intact.
- Address any inconsistencies discovered during project audits or testing.

BP5: Communicate agreed Software Architecture

Effective communication ensures stakeholders understand and align with the software architecture

SWE.2.BP5: Communicate agreed software architecture. Communicate the agreed software architecture to all affected parties.

1. Prepare Architectural Documentation

- Document static and dynamic aspects of the architecture.
- Use standardized formats like SysML or UML for clarity.
- Highlight key architectural decisions and justifications.

2. Engage Stakeholders

- Present the architecture to stakeholders using diagrams and models.
- Collect feedback to address ambiguities and improve alignment.
- Conduct regular reviews to ensure shared understanding.

3. Leverage Tools and Platforms

- Share architectural artifacts through repositories or collaborative platforms.
- Use tools like Confluence or architecture management software for accessibility.
- Ensure version control and traceability of architectural documents.

Challenges in SWE.2 - Software Architectural Design

Complex architectures often face issues of feasibility, scalability, and stakeholder alignment

1. Complexity in Architecture Design

- Difficulty in balancing simplicity with system requirements.
- Managing dependencies between components in large systems.
- Designing for modularity and maintainability.

2. Alignment with System Constraints

- Ensuring consistency with system-level architecture.
- Addressing hardware limitations, communication protocols, and legacy systems.
- Incorporating non-functional requirements like performance and security.

3. Stakeholder Communication and Feedback

- Translating technical architecture into understandable formats for stakeholders.
- Addressing conflicting stakeholder requirements or expectations.
- Managing feedback cycles without scope creep.

SWE.3 - Software Detailed Design and Unit Construction Overview

SWE.3 transforms architectural designs into implementable and testable software units that align with requirements and project goals

1. Purpose of Software Detailed Design and Unit Construction

- Develop detailed designs for software units based on architecture.
- Ensure units meet functional and non-functional requirements.
- Provide inputs for software implementation and testing.

2. Process Scope

- Define static and dynamic aspects of software unit design.
- Ensure design decisions are traceable to software architecture.
- Develop software units, including coding and unit-level documentation.

3. Expected Outcomes

- Documented and validated software unit designs.
- Implemented software units ready for verification.
- Traceability established between detailed design and requirements.

Process ID

SWE.3

Process name

Software Detailed Design and Unit Construction

Process purpose

The purpose is to establish a software detailed design the software architecture, and to construct software units consistent with the software detailed design.

Process outcomes

1) A detailed design is specified including static and dynamic aspects
2) Software units defined by the software detailed design are produced
3) Consistency and bidirectional traceability are established between software detailed design and software architecture; and consistency and bidirectional traceability are established between software units and software detailed design
4) The software detailed design is agreed and communicated to all affected parties

SWE.3 Software Detailed Design and Unit Construction	Outcome 1	Outcome 2	Outcome 3	Outcome 4
Output Information Items				
04-05 Software Detailed Design	X			
11-05 Software Unit	X	X		
13-51 Consistency Evidence			X	
13-52 Communication Evidence				X
Base Practices				
BP1: Specify the static aspects of the detailed design	X			
BP2: Specify the dynamic aspects of the detailed design	X			
BP3: Develop software units		X		
BP4: Ensure consistency and establish bidirectional traceability			X	
BP5: Communicate agreed software detailed design and software units				X

Base practices

SWE.3.BP1: Specify the static aspects of the detailed design. For each software component specify the behavior of its software units, their static structure and relationships, their interfaces including

- valid data value ranges for inputs and outputs (from the application domain perspective)
- physical or measurement units applicable to inputs and outputs (from the application domain perspective)

Note 1: The boundary of a software unit is independent from the software unit's representation in the source code, code file structure, or model-based implementation, respectively. It is rather driven by the semantics of the application domain perspective. Therefore, a software unit may be, at the code level, represented by a single subroutine or a set of subroutines.

Note 2: Examples of valid data value ranges with applicable physical units from the application domain perspective are '0..200 [m/s]', '0.. 3.8 [A]' or '1..100 [N]'. For mapping such application domain value ranges to programming language-level data types (such as unsigned Integer with a value range of 0..65535) refer to BP2.

Note 3: Examples of a measurement unit are '%' or '‰'.

Note 4: A counter is an example of a parameter, or a return value, to which neither a physical nor a measurement unit is applicable.

Note 5: The hardware-software-interface (HSI) definition puts in context the hardware design and therefore is an aspect of system design (SYS.3).

SWE.3.BP2: Specify the dynamic aspects of the detailed design. Specify and document the dynamic aspects of the detailed design with respect to the software architecture, including the interactions between relevant software units to fulfill the component's dynamic behavior.

Note 6: Examples for behavioral descriptions are natural language or semi-formal notation (e.g. SysML, UML).

SWE.3.BP3: Develop software units. Develop and document the software units consistent with the detailed design, and according to coding principles.

Note 7: Examples for coding principles at Capability Level 1 are not to use implicit type conversions, only one entry and one exit point in subroutines, and range checks (design-by-contract, defensive programming). Further examples see e.g. ISO 26262-6 clause 8.4.5 together with table 6.

SWE3.BP4: Ensure consistency and establish bidirectional traceability. Ensure consistency and establish bidirectional traceability between the software detailed design and the software architecture. Ensure consistency and establish bidirectional traceability between the developed software units and the software detailed design. Ensure consistency and establish traceability between the software detailed design and the software requirements.

Note 8: Redundancy should be avoided by establishing a combination of these approaches.

Note 9: Examples for tracing a software unit to a software requirement directly are communication matrices or basis software aspects such as list of diagnosis identifiers inherent in an Autosar configuration.

Note 10: Bidirectional traceability supports consistency, and facilitates impact analysis of change requests, and demonstration of verification coverage.

SWE.3.BP5: Communicate agreed software detailed design and software units. Communicate the agreed software detailed design and software units to all affected parties.

Static design establishes the framework for software units, ensuring consistency with architecture and requirements

BP1: Specify Static Aspects of the Detailed Design

Base practices

SWE.3.BP1: Specify the static aspects of the detailed design. For each software component specify the behavior of its software units, their static structure and relationships, their interfaces including

- valid data value ranges for inputs and outputs (from the application domain perspective)
- physical or measurement units applicable to inputs and outputs (from the application domain perspective)

Note 1: The boundary of a software unit is independent from the software unit's representation in the source code, code file structure, or model-based implementation, respectively. It is rather driven by the semantics of the application domain perspective. Therefore, a software unit may be, at the code level, represented by a single subroutine or a set of subroutines.

Note 2: Examples of valid data value ranges with applicable physical units from the application domain perspective are '0..200 [m/s]', '0.. 3.8 [A]' or '1..100 [N]'. For mapping such application domain value ranges to programming language-level data types (such as unsigned Integer with a value range of 0..65535) refer to BP2.

Note 3: Examples of a measurement unit are '%' or '‰'.

Note 4: A counter is an example of a parameter, or a return value, to which neither a physical nor a measurement unit is applicable.

Note 5: The hardware-software-interface (HSI) definition puts in context the hardware design and therefore is an aspect of system design (SYS.3).

1. Define Unit Structure

- Identify software unit components, including their responsibilities.
- Specify data structures and control interfaces.
- Ensure the design aligns with architectural inputs.

2. Model Static Relationships

- Create static diagrams like class diagrams to represent unit connections.
- Show relationships such as inheritance, composition, and dependencies.
- Highlight key interfaces and shared resources.

3. Ensure Consistency with Architecture

- Validate that static design aligns with architectural constraints.
- Address any gaps or conflicts with system-level architecture.
- Maintain traceability from static design to software requirements.

BP2: Specify Dynamic Aspects of the Software Architecture

Dynamic design ensures software units meet functional requirements through defined runtime interactions.

SWE.3.BP2: Specify the dynamic aspects of the detailed design. Specify and document the dynamic aspects of the detailed design with respect to the software architecture, including the interactions between relevant software units to fulfill the component's dynamic behavior.

Note 6: Examples for behavioral descriptions are natural language or semi-formal notation (e.g. SysML, UML).

1. Define Unit Behavior

- Specify the behavior of software units during runtime.
- Include scenarios like normal operation, error handling, and edge cases.
- Use clear descriptions to capture expected functionality.

2. Model Interactions Between Units

- Create sequence diagrams, state diagrams, or activity flowcharts.
- Represent data exchange, control flows, and event triggers.
- Address interactions with external systems or components.

3. Align with Functional Requirements

- Ensure dynamic behavior meets functional and non-functional requirements.
- Validate runtime scenarios against system use cases and architecture.
- Update dynamic models as requirements evolve.

BP3: Develop Software Units

Software unit development ensures implementation aligns with design, standards, and requirements

SWE.3.BP3: Develop software units. Develop and document the software units consistent with the detailed design, and according to coding principles.

Note 7: Examples for coding principles at Capability Level 1 are not to use implicit type conversions, only one entry and one exit point in subroutines, and range checks (design-by-contract, defensive programming). Further examples see e.g. ISO 26262-6 clause 8.4.5 together with table 6.

1. Implement Software Units

- Develop code based on detailed design specifications.
- Follow coding standards and best practices for readability and maintainability.
- Ensure unit-level functionality matches the design.

2. Document Unit Details

- Provide comprehensive documentation for each unit, including its purpose and usage.
- Include technical details like data structures, algorithms, and interfaces.
- Ensure documentation supports future maintenance and updates.

3. Prepare Units for Testing

- Validate that unit implementation meets design and requirement specifications.
- Include unit-level test cases for functionality and error handling.
- Address any identified gaps or issues before unit verification.

BP4: Ensure Consistency and Traceability

Consistency and traceability link software units and designs to requirements and architecture

SWE3.BP4: Ensure consistency and establish bidirectional traceability. Ensure consistency and establish bidirectional traceability between the software detailed design and the software architecture. Ensure consistency and establish bidirectional traceability between the developed software units and the software detailed design. Ensure consistency and establish traceability between the software detailed design and the software requirements.

Note 8: Redundancy should be avoided by establishing a combination of these approaches.

Note 9: Examples for tracing a software unit to a software requirement directly are communication matrices or basis software aspects such as list of diagnosis identifiers inherent in an Autosar configuration.

Note 10: Bidirectional traceability supports consistency, and facilitates impact analysis of change requests, and demonstration of verification coverage.

1. Establish Traceability Links

- Link software units and detailed designs to architectural elements and requirements.
- Ensure bidirectional traceability to support validation and updates.
- Use tools like DOORS, Jira, or Git for managing traceability artifacts.

2. Verify Consistency Across Artifacts

- Ensure alignment between unit implementation, design, and architecture.
- Address conflicts or inconsistencies during reviews and updates.
- Regularly validate consistency throughout the development lifecycle.

3. Maintain Updated Traceability

- Update traceability links as units or designs evolve.
- Include traceability reviews as part of quality assurance processes.
- Document changes to ensure transparency and maintain alignment.

BP5: Communicate Detailed Design and Units

Clear communication of designs and units ensures stakeholder alignment and informed decision-making

SWE.3.BP5: Communicate agreed software detailed design and software units.
Communicate the agreed software detailed design and software units to all affected parties.

1. Prepare Documentation

- Compile detailed design specifications and unit documentation.
- Include diagrams, implementation details, and test case descriptions.
- Use standardized templates for clarity and consistency.

2. Engage Stakeholders

- Present detailed designs and unit updates in reviews and meetings.
- Address stakeholder feedback to refine designs and units.
- Ensure stakeholders understand the rationale behind design decisions.

3. Leverage Collaborative Tools

- Use platforms like Confluence, Jira, or Git to share documentation and updates.
- Facilitate collaborative reviews with tools supporting annotations and feedback.
- Ensure version control for all shared artifacts.

Challenges in SWE.3 - Software Detailed Design and Unit Construction

Software detailed design often faces challenges in alignment, modularity, and maintainability

1. Ensuring Alignment with Architecture

- Difficulty maintaining consistency between detailed design and architecture.
- Misinterpretation of architectural inputs can lead to design conflicts.
- Traceability gaps between detailed design and requirements.

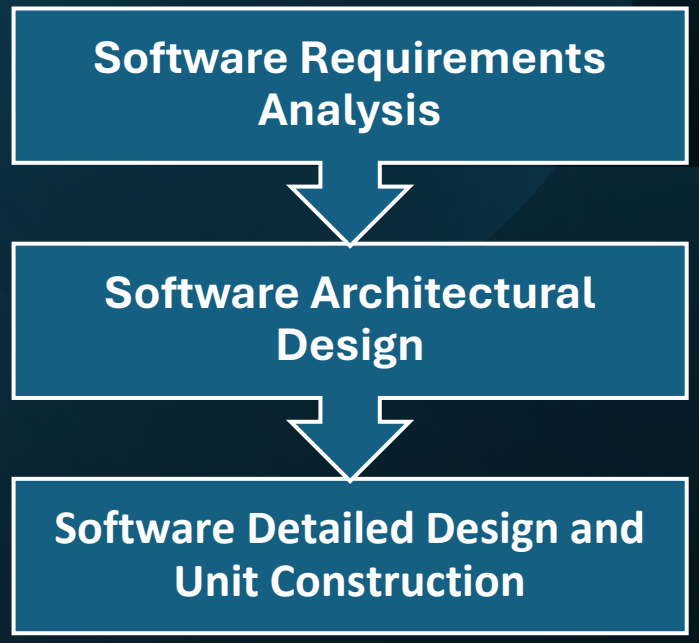
2. Designing for Modularity and Scalability

- Challenges in designing reusable and modular software units.
- Ensuring scalability without overcomplicating the design.
- Balancing design complexity with performance requirements.

3. Maintaining Code Quality and Documentation

- Translating technical architecture into understandable formats for stakeholders.
- Addressing conflicting stakeholder requirements or expectations.
- Managing feedback cycles without scope creep.

Summary and Q&A



Software engineering processes ensure structured design, implementation, and traceability for successful project outcomes.

1. SWE.1 - Software Requirements Analysis

- Focused on deriving, structuring, and analyzing software requirements.
- Emphasized consistency with system requirements and traceability.
- Identified challenges in alignment and stakeholder communication.

2. SWE.2 - Software Architectural Design

- Defined static and dynamic aspects of software architecture.
- Ensured alignment with system architecture and project constraints.
- Addressed challenges in complexity, scalability, and feedback management.

3. SWE.3 - Software Detailed Design and Unit Construction

- Transformed architecture into detailed designs and implemented units.Maintained traceability and prepared units for testing.Highlighted challenges in modularity, maintainability, and code quality.