LECTURE 9

Software requirements Engineering

Interaction between requirement and architecture

Instructor:
Dr. Natalia Chaudhry,
Assistant Professor, PUCIT, University of the Punjab, Lahore.

- relationship of architecture to requirements

Requirements give information to the system designers and to a wide range of stakeholders. They state what the stakeholders want the system to achieve. Requirements can be classified into ‘requirement types’ as follows:

1. Vision: at the highest level, the future direction for a system.

2. Function Requirements: what a system has to ‘do’: the essence of a system, its mission and fundamental functionality.

3. Performance Requirements: the performance levels that the stakeholders want – their objectives. How good?

- Two essential activities in the software development process are requirements engineering (RE) and software architecting.
  - The focus of RE is on eliciting, analyzing, and managing requirements.
  - Software architecting is concerned with providing an abstraction of the system as a blueprint to manage the complexity of software systems.
- The development of software architectures is a challenging task, even when the requirements for a software system are clear.
- Requirements in general and quality requirements in particular drive the architecture of a software system, whereas decisions made in the architectural phase can affect the achievement of initial requirements and thus change them

- The common way of traditional software development processes such as the waterfall model is to build a software architecture from requirement descriptions.
- This process considers the forward development process from requirements to the software architecture, it however does not consider the impact of design decisions on initial requirements

- The problem of the linear software development processes is twofold.
- On the one hand requirements are elicited, analyzed, and specified in isolation without considering the impact of architecture artifacts.
- On the other hand, design decisions are made without managing the conflicts and making necessary changes in the requirements.

requirement descriptions cannot be considered in isolation and should be co-developed with architectural descriptions iteratively and concurrently. There is, however, no structured solution on how to perform the co-development of requirements and software architecture.

- Imagine a software project aimed at developing a new e-commerce platform. The requirements gathering phase identifies that the platform should support high availability, scalability to handle peak loads during sales events, and a user-friendly interface for customers to browse products and make purchases.
- Now, in a traditional linear development process, the requirements might be documented and finalized without much consideration for the underlying architecture.
- For instance, the requirements might specify features like real-time inventory updates and personalized product recommendations without considering the architectural implications.

- However, when the architecture is later designed, the architects may realize that achieving real-time inventory updates requires a distributed microservices architecture with sophisticated synchronization mechanisms.
- Similarly, implementing personalized product recommendations might necessitate a scalable data processing pipeline.
- If these architectural considerations were not taken into account during the requirements phase, there could be several issues

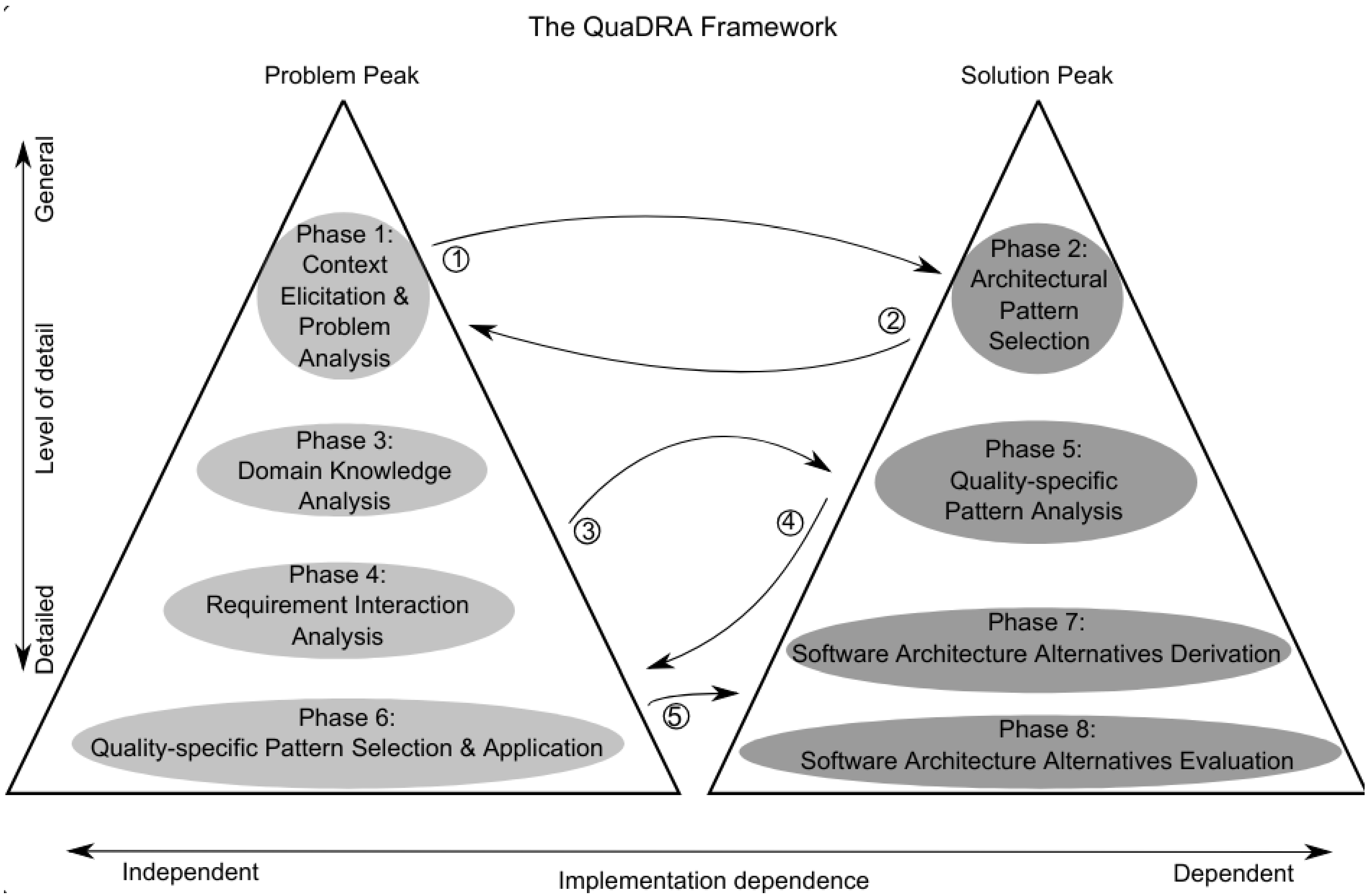
- **Feasibility Concerns**: The architectural constraints may render some requirements unfeasible or significantly more complex to implement than originally anticipated. For example, the requirement for real-time inventory updates might be challenging to achieve within the project's time and resource constraints.

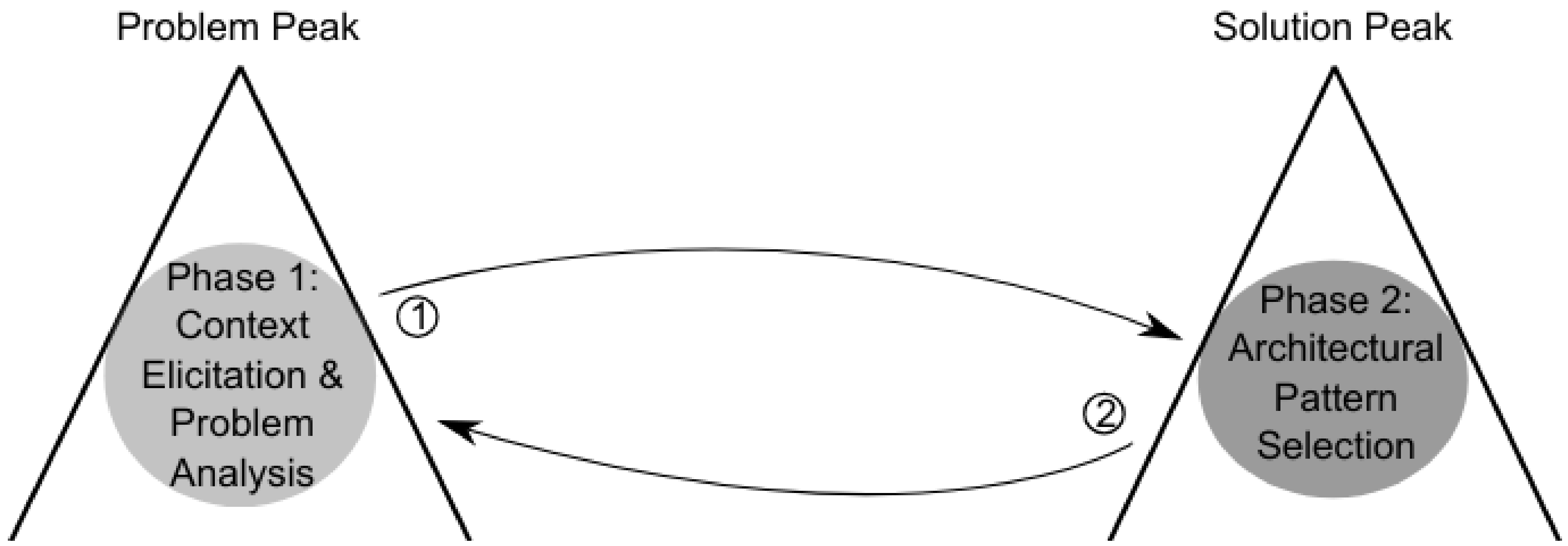
- **Performance and Scalability**: The architecture might not adequately support the scalability requirements specified in the requirements document. For instance, if the architecture lacks horizontal scalability, the platform might struggle to handle peak loads during sales events, leading to downtime or degraded performance.

•**User Experience**: The architectural design might not align with the user interface requirements, leading to a disjointed or suboptimal user experience. For example, if the architecture introduces latency or bottlenecks in certain parts of the system, it could impact the responsiveness of the user interface.

a framework for the Problem oriented and Quality-based Co-Development of Requirements and Architecture (QuaDRA).

- The QuaDRA Framework, or Quality-Driven Requirements Analysis, is a systematic approach used in software engineering to ensure that software requirements are well-defined, clear, and aligned with stakeholder expectations.


The QuaDRA Framework
Problem Peak
Solution Peak
General
Level of detail
Detailed
Phase 1: Context Elicitation & Problem Analysis
Phase 2: Architectural Pattern Selection
Phase 3: Domain Knowledge Analysis
Phase 4: Requirement Interaction Analysis
Phase 5: Quality-specific Pattern Analysis
Phase 6: Quality-specific Pattern Selection & Application
Phase 7: Software Architecture Alternatives Derivation
Phase 8: Software Architecture Alternatives Evaluation
①
②
③
④
⑤
Independent
Implementation dependence
Dependent


Problem Peak
Solution Peak
Phase 1: Context Elicitation & Problem Analysis
Phase 2: Architectural Pattern Selection
①
②

- Phase 1 (Context Elicitation & Problem Analysis) is concerned with modeling the functional and quality requirements. Existing documents including functional and quality requirements (also known as non-functional requirements) are required as input for this phase.

Here's a breakdown of this phase:

Understanding Context:
This involves gathering information about the environment in which the software will operate. It includes understanding the business objectives, organizational structure, market conditions, regulatory requirements, technological constraints, and any other relevant factors.
Stakeholder Identification:
Identifying all stakeholders involved in or affected by the software project is crucial. This includes end-users, customers, business owners, developers, testers, regulatory bodies, and other relevant parties.
Stakeholder Analysis:
Analyzing stakeholders involves understanding their roles, responsibilities, interests, expectations, and concerns regarding the software project. This analysis helps prioritize stakeholders and determine the level of involvement required from each.

Problem Identification:
Identifying the problem or opportunity that the software intends to address is fundamental. This involves understanding the main points, challenges, inefficiencies
Goal Definition:
Defining clear and measurable goals for the software project is essential. These goals should align with the broader organizational objectives and address the identified problems or opportunities.
Scope Definition:
Defining the scope of the software project involves specifying the boundaries of what will be included and excluded from the solution. This helps manage stakeholders' expectations, prevent scope creep, and ensure that the project remains manageable within the available resources and timeline.
Contextual Documentation:
Documenting the gathered information, including stakeholder profiles, problem statements, goals, and scope, is crucial for maintaining a shared understanding among team members and stakeholders. Clear and concise documentation serves as a reference throughout the requirements analysis process.

Example: For a project developing a new mobile banking application:

- **Context Understanding:** Researching the banking industry's regulatory requirements, technological trends in mobile app development, and customer preferences.
- **Stakeholder Identification:** Identifying stakeholders such as bank executives, customers, regulatory bodies, app developers, and customer service representatives.
- **Problem Identification:** Recognizing the need for a user-friendly mobile banking solution that offers secure transactions, easy account management, and personalized services.
- **Goal Definition:** Setting goals to increase customer satisfaction, streamline banking operations, and attract new customers through the mobile app.
- **Scope Definition:** Defining the app's features, such as account balance inquiry, funds transfer, bill payment, and customer support, while excluding complex investment services.
- **Contextual Documentation:** Creating documents summarizing stakeholder roles, the problem statement, project goals, and the defined scope for reference during the requirements analysis process.

Phase 2 of the QuaDRA Framework, "Architectural Pattern Selection," involves choosing appropriate architectural patterns to address the identified quality attributes and meet the project's goals. Architectural patterns are high-level design solutions that provide proven structures for organizing and designing software systems.

Review of Quality Attributes:
In this phase, the quality attributes identified in Phase 1 are revisited to understand their significance and impact on the software architecture. Key quality attributes such as performance, scalability, security, and usability are analyzed to determine their importance in the architectural design.
Selection of Architectural Patterns:
Based on the identified quality attributes and project goals, suitable architectural patterns are selected. These patterns serve as blueprints for organizing the system's components and interactions to achieve the desired qualities. Common architectural patterns include layered architecture, client-server architecture, microservices architecture, and event-driven architecture.

Analysis of Pattern Suitability:
Each selected architectural pattern is evaluated to assess its suitability for the project's requirements. Factors such as scalability, flexibility, maintainability, and alignment with business goals are considered during this analysis. The goal is to choose patterns that best align with the project's needs and constraints.
Mapping Patterns to Requirements:
Once architectural patterns are selected, they are mapped to specific requirements and quality attributes. This mapping ensures that each pattern addresses the relevant aspects of the system design and contributes to achieving the desired qualities. For example, a microservices architecture may be chosen to enhance scalability and maintainability, while a layered architecture may be selected to promote modularity and reusability.

Documentation and Communication:
The selected architectural patterns, along with their rationale and implications, are documented and communicated to stakeholders. Clear documentation helps ensure a shared understanding among team members and facilitates communication throughout the development process.

Let's consider a different example to illustrate Phase 2 of the QuaDRA Framework, focusing on the development of a social networking platform. Here's how architectural pattern selection might be approached in this context:

Project Context: The goal is to develop a social networking platform where users can create profiles, connect with friends, share content such as photos and posts, and interact with each other through comments and messages. Key quality attributes identified include scalability, usability, and security.

1. **Review of Quality Attributes**:
    1. Scalability: The platform should be able to handle a growing user base and increasing amounts of user-generated content.
    2. Usability: The platform should provide an intuitive and engaging user experience to encourage user adoption and retention.
    3. Security: The platform should protect user data and privacy, prevent unauthorized access, and mitigate security risks such as data breaches and cyber-attacks.

2. Selection of Architectural Patterns:

1. Considering the scalability requirement, a suitable architectural pattern could be the "scalable web application architecture." This pattern typically involves a distributed system architecture with components such as load balancers, multiple application servers, and a scalable database infrastructure. Technologies like microservices and containerization might be utilized to achieve horizontal scalability.
2. For usability, a "single-page application (SPA)" architecture could be chosen. SPAs provide a responsive and interactive user experience by dynamically updating content without requiring full page reloads. Frameworks like React.js or Angular.js can be employed to build SPAs.
3. To address security concerns, a "layered security architecture" might be adopted. This architecture involves implementing multiple layers of security controls such as authentication, authorization, encryption, and intrusion detection at various levels of the application stack.

3. Analysis of Pattern Suitability:

1. The scalable web application architecture is suitable for handling the platform's growing user base and content volume. It allows for horizontal scaling by adding more server instances and distributing traffic across them.
2. The single-page application architecture enhances usability by providing a smooth and responsive user interface. It reduces page load times and enhances user engagement.
3. The layered security architecture helps mitigate security risks by implementing defense mechanisms at different layers of the application, thereby reducing the attack surface and protecting sensitive data.

4. Mapping Patterns to Requirements:

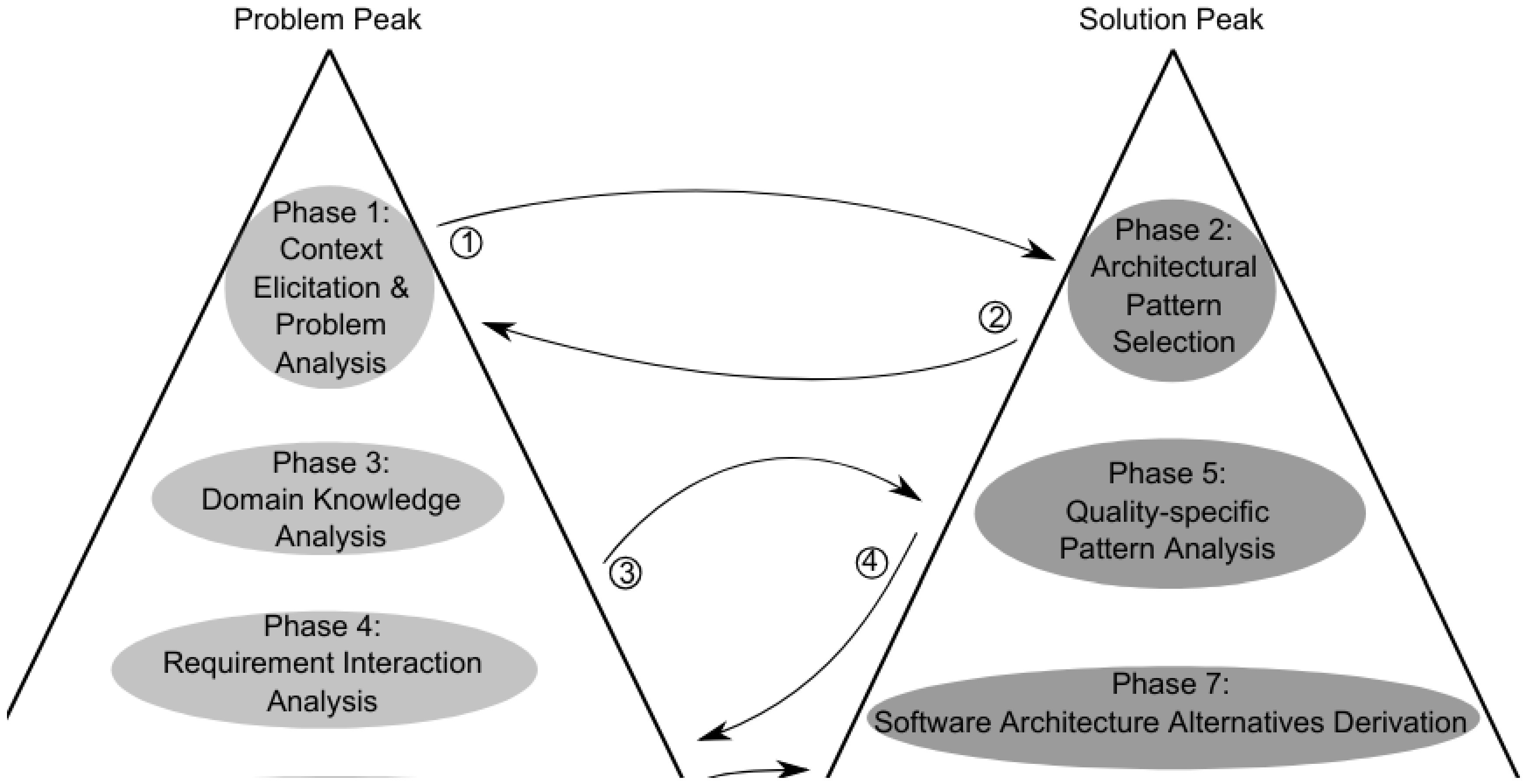
1. The scalable web application architecture addresses scalability requirements by allowing the platform to handle increased traffic and user interactions.
2. The single-page application architecture enhances usability by providing a fast and interactive user experience.
3. The layered security architecture ensures that the platform's data and user interactions are protected against security threats and unauthorized access.

5. Documentation and Communication:

1. The decision to adopt specific architectural patterns, along with their implications and design considerations, is documented and communicated to stakeholders. This ensures alignment with project goals and facilitates collaboration among team members.

In summary, architectural pattern selection in Phase 2 of the QuaDRA Framework involves identifying and choosing appropriate patterns to address the project's quality attributes and requirements. This process ensures that the resulting software architecture is well-suited to achieve the project's objectives effectively and efficiently.

Phase 3: Domain Knowledge Analysis


Problem Peak
Solution Peak
Phase 1: Context Elicitation & Problem Analysis
Phase 2: Architectural Pattern Selection
Phase 3: Domain Knowledge Analysis
Phase 4: Requirement Interaction Analysis
Phase 5: Quality-specific Pattern Analysis
Phase 7: Software Architecture Alternatives Derivation
①
②
③
④

Phase 3 of the QuaDRA Framework, "Domain Knowledge Analysis," focuses on gaining a deep understanding of the domain in which the software will operate.

This phase involves analyzing domain-specific concepts, terminology, processes, and constraints to ensure that the requirements capture the domain accurately.

Example: Healthcare Management System
Consider the development of a healthcare management system (HMS) for a hospital. The system aims to streamline various healthcare processes, including patient management, appointment scheduling, medical records management, and billing. Here's how Phase 3 might be applied:

1. **Gathering Domain Knowledge**:
    1. Domain experts, including healthcare professionals, administrators, and IT specialists, collaborate to provide insights into the healthcare domain. They share their expertise regarding hospital workflows, medical terminology, regulatory requirements and best practices in healthcare management.

2. **Analyzing Healthcare Processes**:
    1. Detailed analysis is conducted on various healthcare processes to understand how the hospital operates. This includes patient admission, diagnosis, treatment, medication management, discharge, and follow-up care. Understanding the sequence of activities, roles of different stakeholders (e.g., doctors, nurses, receptionists), and data flow between systems is essential.
3. **Understanding Medical Terminology**:
    1. Since healthcare involves specialized terminology, analysts delve into medical terminology to ensure accurate representation in the software requirements. This includes terminology related to diseases, treatments, medications, laboratory tests, and diagnostic procedures. For example, understanding the difference between ICD-10 (International Classification of Diseases) codes for diagnoses and CPT (Current Procedural Terminology) codes for procedures is crucial for billing and insurance processing.

4. Exploring Regulatory Compliance:

1. Compliance with healthcare regulations, such as HIPAA (Health Insurance Portability and Accountability Act) in the United States, is paramount. Analysts study relevant regulations and standards to ensure that the software complies with legal requirements regarding patient privacy, data security, electronic health records (EHR), and interoperability with other healthcare systems.

5. Identifying Stakeholder Needs:

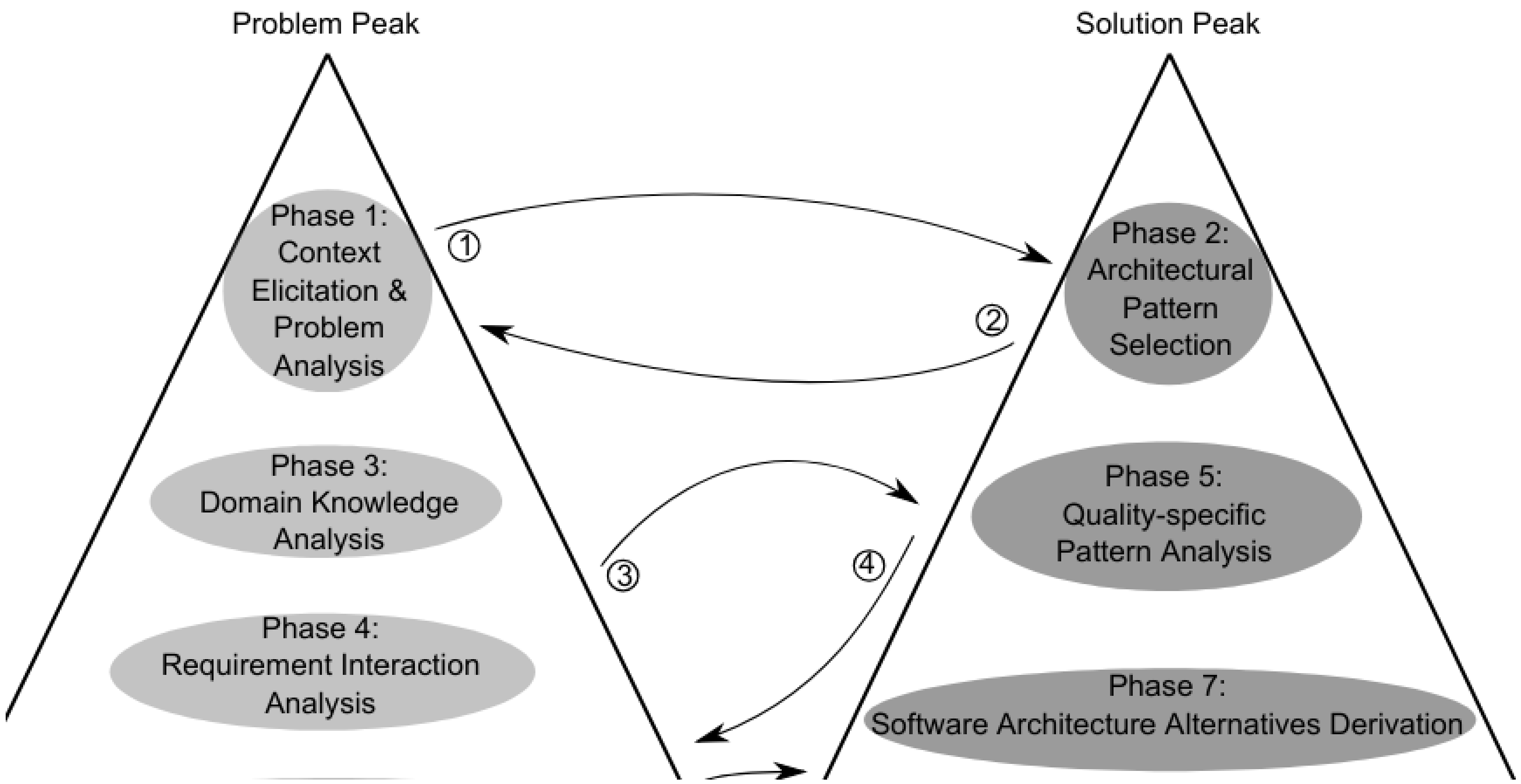
1. Different stakeholders in the healthcare domain have unique needs and priorities. Physicians may prioritize clinical efficiency and decision support tools, whereas administrators may focus on revenue cycle management and resource allocation. Understanding these diverse needs helps prioritize requirements and design features that cater to each stakeholder group.

6. Analyzing Integration Points:

1. Healthcare systems often need to integrate with existing IT infrastructure, including EHR systems, laboratory information systems (LIS), picture archiving and communication systems (PACS), and billing systems. Analysts analyze integration points, data exchange formats, and interoperability standards (such as HL7) to ensure seamless communication between systems.

Outcome: Through Domain Knowledge Analysis, the software development team gains a comprehensive understanding of the healthcare domain's intricacies, processes, terminology, and regulatory requirements. This deep domain knowledge ensures that the software requirements accurately reflect the needs of healthcare professionals and align with industry standards and regulations. As a result, the healthcare management system developed using the QuaDRA Framework effectively addresses the complex challenges of healthcare delivery while meeting stakeholder expectations and regulatory compliance.

Phase 4 of the QuaDRA Framework, "Requirement Interaction Analysis,"


Problem Peak
Solution Peak
Phase 1: Context Elicitation & Problem Analysis
Phase 2: Architectural Pattern Selection
Phase 3: Domain Knowledge Analysis
Phase 4: Requirement Interaction Analysis
Phase 5: Quality-specific Pattern Analysis
Phase 7: Software Architecture Alternatives Derivation
①
②
③
④

- Phase 4 of the QuaDRA Framework, "Requirement Interaction Analysis," focuses on understanding how different requirements interact with each other within the software system.
- This phase aims to identify dependencies, conflicts among requirements to ensure that they work together.

Consider the development of an e-learning platform that provides online courses to students. The platform includes features such as course enrollment, video lectures, quizzes, assignments, and discussion forums.

1. **Identifying Requirement Dependencies**:
    1. Requirement analysts identify dependencies between different requirements. For example, the requirement for "user authentication" is dependent on the requirement for "user registration." Similarly, the requirement for "course enrollment" depends on the existence of available courses.
2. **Analyzing Conflicting Requirements**:
    1. Conflicts between requirements are analyzed to resolve inconsistencies. For instance, if one requirement specifies that users must complete a quiz before accessing course materials, while another requirement allows users to skip quizzes, this conflict needs to be addressed through negotiation or prioritization.

3. Detecting Requirement Interactions:

1. Analysts examine how requirements interact with each other to achieve the desired functionality. For example, the requirement for "grading assignments" interacts with the requirement for "student submissions" and "instructor feedback." Understanding these interactions ensures that the grading process aligns with submission and feedback mechanisms.

4. Managing Requirement Synergies/union:

1. Synergies between requirements are identified to leverage opportunities for enhancing system functionality. For instance, the requirement for "real-time collaboration tools" synergizes with the requirement for "discussion forums," enabling students and instructors to engage in synchronous discussions and collaborative activities.

5. Addressing Cross-Cutting Concerns:

1. Cross-cutting concerns, such as security, scalability, and usability, are analyzed to ensure that they are addressed consistently across different requirements. For example, the requirement for "secure user authentication" should be considered across all user-related functionalities, including registration, login, and profile management.

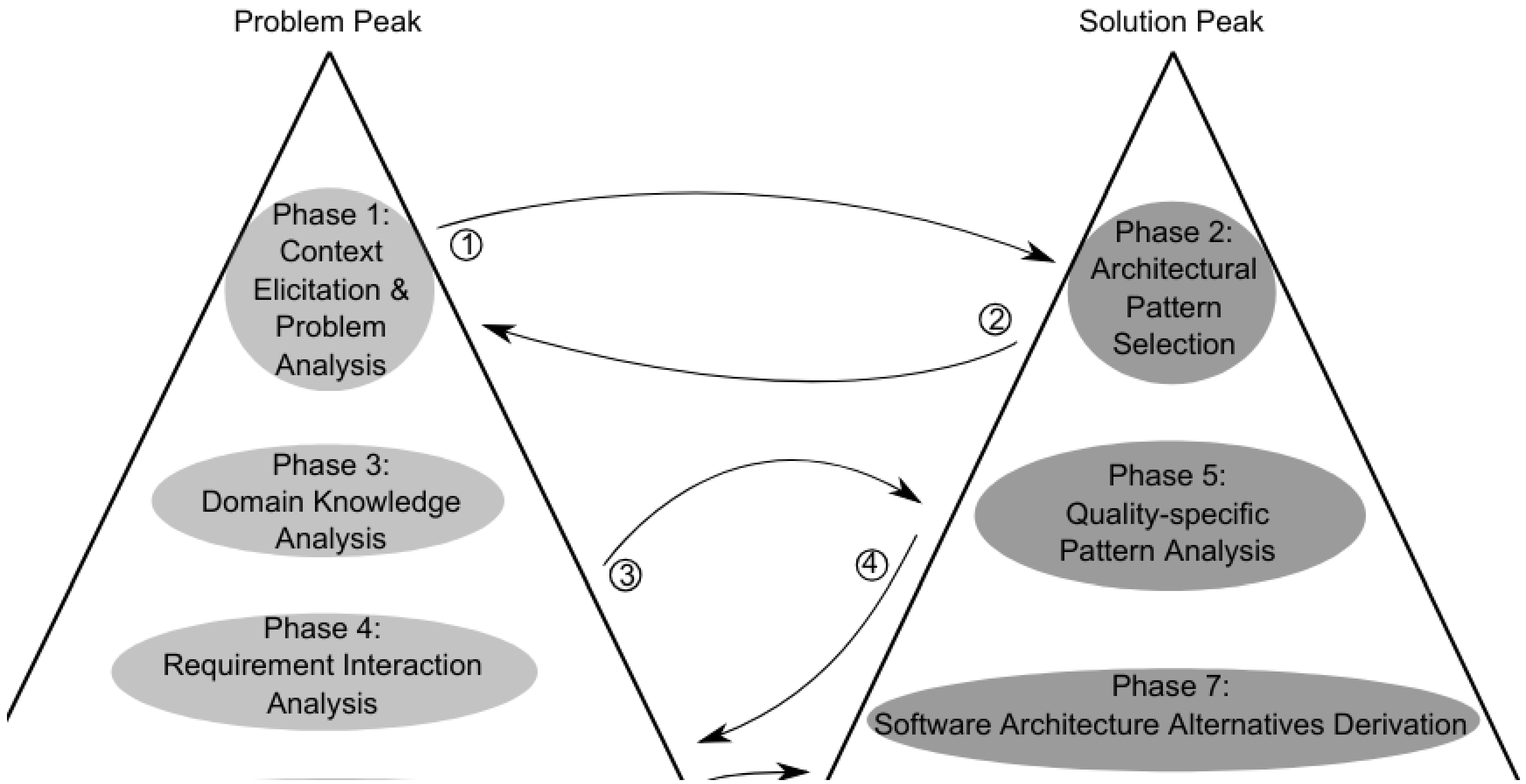
Outcome

Through Requirement Interaction Analysis, the software development team gains insights into how different requirements interact with each other to deliver the desired functionality.

By identifying dependencies, conflicts, synergies, and cross-cutting concerns, the team ensures that the requirements are cohesive, coherent, and aligned with the overarching goals of the e-learning platform.

This analysis lays the foundation for designing a robust and integrated system that meets user needs effectively.

Phase 5: Quality-specific Pattern Analysis


Problem Peak
Solution Peak
Phase 1: Context Elicitation & Problem Analysis
①
Phase 2: Architectural Pattern Selection
②
Phase 3: Domain Knowledge Analysis
③
④
Phase 5: Quality-specific Pattern Analysis
Phase 4: Requirement Interaction Analysis
Phase 7: Software Architecture Alternatives Derivation

Phase 5 of the QuaDRA Framework, "Quality-specific Pattern Analysis," involves identifying and analyzing specific design patterns that address the quality attributes prioritized earlier in the framework. This phase aims to select appropriate design patterns that will help achieve the desired quality goals effectively.

Consider the development of a mobile health monitoring application designed to track patients' vital signs, such as heart rate, blood pressure, and blood glucose levels, remotely. The application aims to ensure high reliability, scalability, and security.

1. **Identification of Quality-specific Patterns**:
    1. Quality-specific patterns that address reliability, scalability, and security are identified.
    2. For reliability, patterns such as "Retry Pattern" and "Circuit Breaker Pattern" may be considered to handle transient failures gracefully and ensure system availability.
    3. For scalability, patterns like "Microservices Architecture" and "Load Balancing Pattern" may be chosen to distribute workload efficiently and scale components horizontally.
    4. For security, patterns such as "Authentication and Authorization Patterns" and "Data Encryption Pattern" may be selected to enforce access control and protect sensitive data.

2. Analysis of Pattern Suitability:

1. Each identified pattern is analyzed to assess its suitability for the application's requirements and constraints.
2. For example, the "Retry Pattern" is suitable for handling irregular network connectivity issues in mobile environments, ensuring reliable data transmission.
3. The "Microservices Architecture" is appropriate for achieving scalability by allowing independent deployment and scaling of modular services.
4. The "Authentication and Authorization Patterns" are essential for enforcing user authentication and access control to protect patient data.

3. Evaluation of Pattern Trade-offs:

1. Trade-offs associated with each pattern are evaluated to understand their implications on other quality attributes and system characteristics.
2. For instance, while the "Retry Pattern" enhances reliability by retrying failed requests, it may increase network traffic and latency.
3. Similarly, while the "Microservices Architecture" offers scalability and flexibility, it introduces complexity in service communication and management.

4. Mapping Patterns to Requirements:

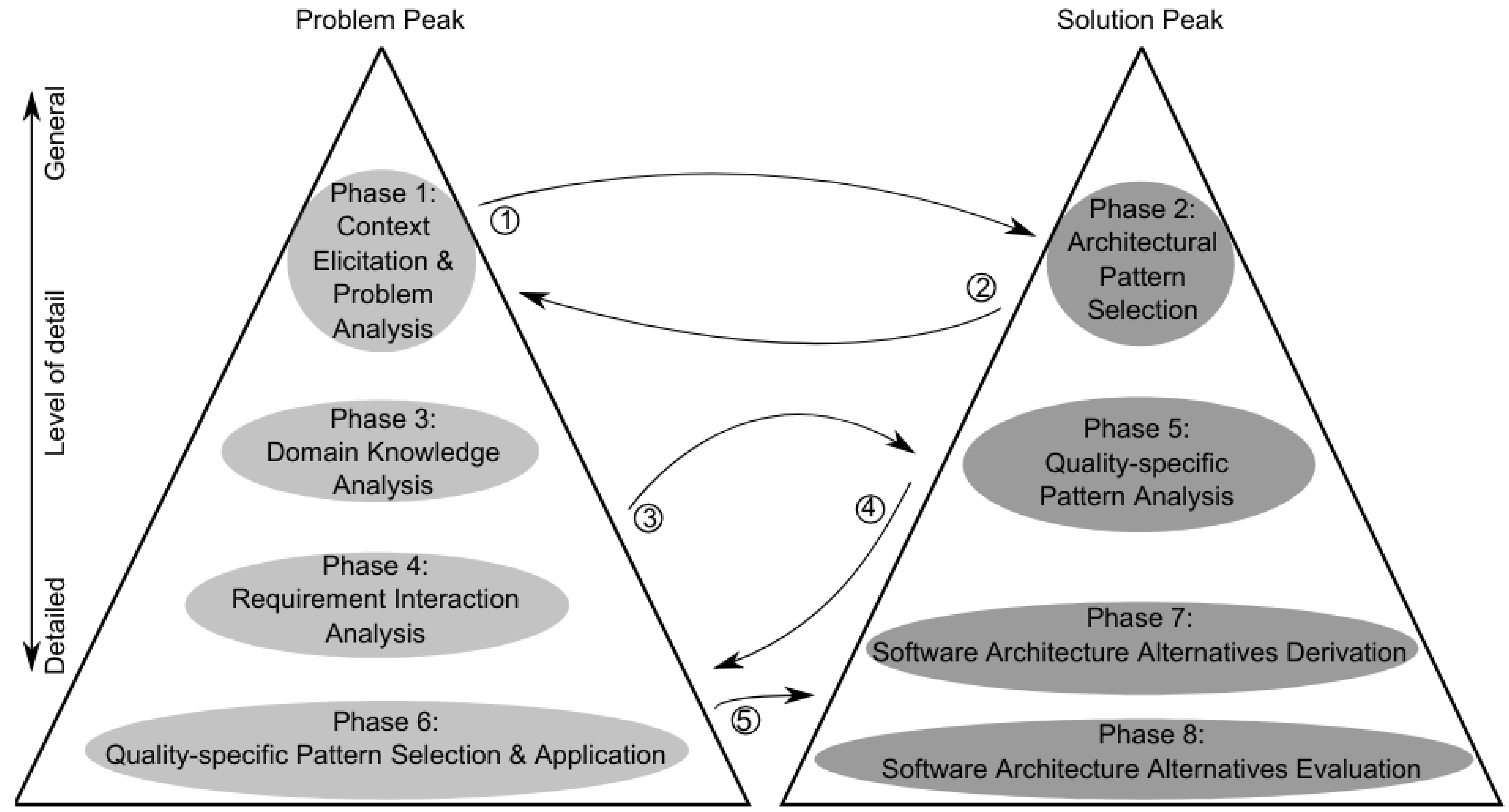
1. Each selected pattern is mapped to specific requirements and quality attributes to ensure that they address the desired quality goals effectively.
2. For example, the "Retry Pattern" is mapped to the reliability requirement to ensure uninterrupted data transmission.
3. The "Microservices Architecture" is mapped to the scalability requirement to support the growing number of users and data volume.
4. The "Authentication and Authorization Patterns" are mapped to the security requirement to protect patient privacy and comply with healthcare regulations.

Outcome:
Through Quality-specific Pattern Analysis, the software development team selects appropriate design patterns that align with the prioritized quality attributes of the mobile health monitoring application.

By leveraging these patterns, the team can design a robust, scalable, and secure system that meets the reliability, scalability, and security requirements effectively.

This analysis ensures that the architectural decisions are well-informed and tailored to achieve the desired quality goals of the application.

Phase 6: Quality-specific Pattern Selection & Application


Problem Peak
Solution Peak
General
Level of detail
Detailed
Phase 1: Context Elicitation & Problem Analysis
Phase 2: Architectural Pattern Selection
Phase 3: Domain Knowledge Analysis
Phase 4: Requirement Interaction Analysis
Phase 5: Quality-specific Pattern Analysis
Phase 6: Quality-specific Pattern Selection & Application
Phase 7: Software Architecture Alternatives Derivation
Phase 8: Software Architecture Alternatives Evaluation
①
②
③
④
⑤

Phase 6 of the QuaDRA Framework, "Quality-specific Pattern Selection & Application," involves the actual selection and application of quality-specific design patterns identified in Phase 5.

This phase focuses on integrating these patterns into the software architecture to address the prioritized quality attributes effectively.

Consider the development of an e-commerce platform that prioritizes performance, scalability, and security. The platform aims to handle a large number of concurrent users, ensure fast response times, and protect customer data from security threats.

1. **Selection of Quality-specific Patterns**:
    1. Based on the prioritized quality attributes, specific design patterns are selected to address performance, scalability, and security concerns.
    2. For performance, patterns such as "Caching Pattern" and "Asynchronous Processing Pattern" may be chosen to optimize response times and reduce server load.
    3. For scalability, patterns like "Microservices Architecture" and "Distributed Cache Pattern" may be selected to support horizontal scaling and handle increased user traffic.
    4. For security, patterns such as "Token-based Authentication" and "Role-based Access Control (RBAC)" may be adopted to enforce authentication and authorization mechanisms.

2. Application of Selected Patterns:

1. The selected patterns are applied within the software architecture to address the identified quality goals. For example:
   1. The "Caching Pattern" is applied to cache frequently accessed data, such as product listings and user profiles, to improve response times and reduce database load.
   2. The "Microservices Architecture" is implemented to break down the application into smaller, independently deployable services, allowing horizontal scaling of individual components based on demand.
   3. The "Token-based Authentication" pattern is used to generate and validate authentication tokens for user sessions, ensuring secure access to protected resources.

3. Integration with Existing Architecture:

1. The selected patterns are integrated with the existing software architecture, ensuring compatibility and coherence with other architectural components and design decisions.
2. For example, the "Microservices Architecture" is seamlessly integrated with service discovery mechanisms and API gateways to facilitate communication between microservices.

4. Testing and Validation:

1. The implemented patterns are thoroughly tested to validate their effectiveness in addressing the prioritized quality attributes.
2. Performance testing measures response times and system throughput, scalability testing evaluates the system's ability to handle increased load, and security testing assesses vulnerabilities and compliance with security requirements.

Phase 7: Software Architecture Alternatives Derivation

Phase 7 of the QuaDRA Framework, "Software Architecture Alternatives Derivation," involves exploring and deriving alternative architectural solutions to address the project's requirements and quality attributes.

This phase aims to consider different architectural options and evaluate their strengths, weaknesses, and trade-offs before making final architectural decisions.

Consider the development of a ride-sharing service similar to Uber. The service aims to connect passengers with drivers for on-demand transportation. Key requirements include real-time tracking of vehicles, efficient matching of drivers and passengers, and secure payment processing. Quality attributes prioritized include scalability, reliability, and security.

1. **Identification of Architectural Alternatives**:
    1. Different architectural alternatives are explored based on the project's requirements and quality attributes. For example, alternatives might include:
        1. **Monolithic Architecture:** A single, centralized system where all components, such as user management, trip management, and payment processing, are tightly integrated.
        2. **Microservices Architecture:** Decomposing the system into loosely coupled, independently deployable services, each responsible for specific functions, such as user authentication, trip management, and payment processing.
        3. **Serverless Architecture:** Using cloud-based serverless functions to handle specific tasks, such as real-time vehicle tracking, trip notifications, and payment processing, without managing server infrastructure.
        4. **Hybrid Architecture**: Combining elements of different architectures, such as using microservices for core business functions and serverless functions for event-driven tasks.

2. Evaluation of Alternatives:

1. Each architectural alternative is evaluated against the project's requirements and quality attributes. Considerations include:
    1. Scalability: How well does each architecture support scaling to accommodate increasing numbers of users and requests?
    2. Reliability: What mechanisms are in place to ensure high availability, fault tolerance, and recovery from failures?
    3. Security: How are sensitive data and transactions protected against unauthorized access, data breaches, and cyber-attacks?
    4. Development and Operations: What are the implications in terms of development complexity, deployment, maintenance, and operational overhead?

3. Analysis of Trade-offs:

1. Trade-offs associated with each architectural alternative are analyzed to understand their implications on various aspects of the project. For example:
   1. Monolithic Architecture may offer simplicity in development but could pose challenges in scaling and maintaining the system as it grows.
   2. Microservices Architecture provides flexibility and scalability but introduces complexity in managing distributed systems and inter-service communication.
   3. Serverless Architecture offers cost-effectiveness and scalability but may limit customization and control over the underlying infrastructure.

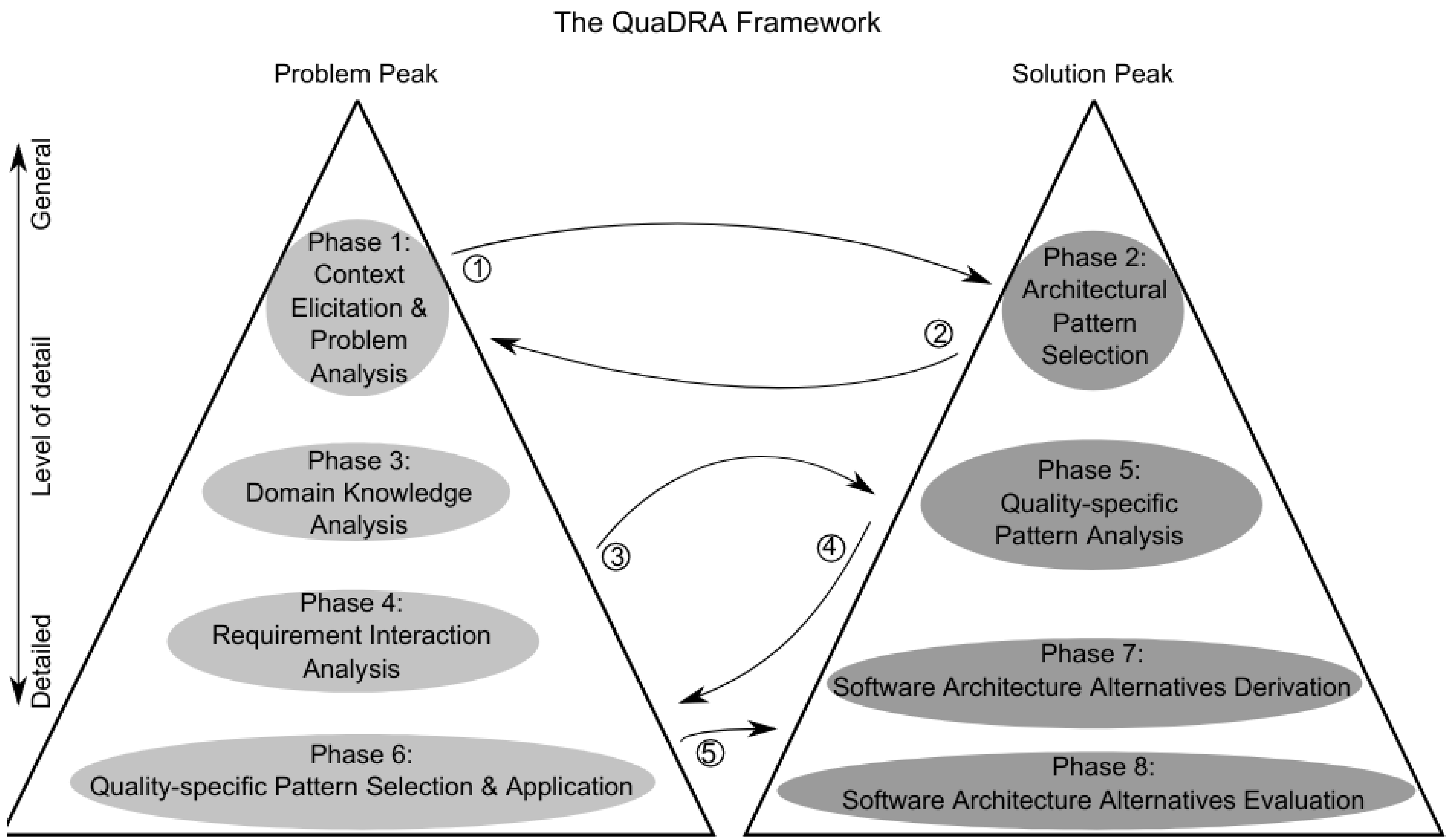
4. Selection of Preferred Architecture:

1. After evaluating the alternatives and considering their trade-offs, the preferred architecture is selected based on its alignment with the project's requirements, quality attributes, and constraints. For example:
    1. Given the scalability requirement and the need for flexibility in adding new features, a Microservices Architecture might be chosen.

5. Documentation and Communication:

1. The selected architecture, along with its rationale, trade-offs, and implications, is documented and communicated to stakeholders to ensure a shared understanding and alignment with project goals.

Phase 8: Software Architecture Alternatives Evaluation


The QuaDRA Framework
Problem Peak
Solution Peak
General
Level of detail
Detailed
Phase 1: Context Elicitation & Problem Analysis
Phase 2: Architectural Pattern Selection
Phase 3: Domain Knowledge Analysis
Phase 4: Requirement Interaction Analysis
Phase 5: Quality-specific Pattern Analysis
Phase 6: Quality-specific Pattern Selection & Application
Phase 7: Software Architecture Alternatives Derivation
Phase 8: Software Architecture Alternatives Evaluation
①
②
③
④
⑤

- Phase 8 of the QuaDRA Framework, "Software Architecture Alternatives Evaluation," involves a detailed assessment of the architectural alternatives derived in Phase 7.
- This phase aims to compare and evaluate the pros and cons of each alternative against the project's requirements, quality attributes, and constraints to make an informed decision on the final architecture.

Consider the development of a content management system (CMS) for a media company. The CMS aims to enable content creators to publish articles, images, and videos across various platforms, including websites, mobile apps, and social media channels.
Key requirements include content management, user authentication, and integration with third-party services. Quality attributes prioritized include usability, scalability, and extensibility.

1. **Architectural Alternatives**:
    1. Three architectural alternatives have been derived:
        1. Monolithic Architecture: A single, integrated system where all components, such as content management, user authentication, and third-party integrations, are tightly coupled.
        2. Microservices Architecture: Decomposing the system into loosely coupled, independently deployable services, each responsible for specific functions, such as content management service, authentication service, and integration service.
        3. Headless CMS Architecture: Separating the content management backend from the presentation layer, allowing content to be managed independently of the frontend, enabling multi-channel content delivery.

2. Evaluation Criteria:

1. Criteria for evaluating the architectural alternatives include:
    1. Usability: How intuitive and user-friendly is the system for content creators and administrators?
    2. Scalability: How well does the architecture support scaling to accommodate increasing content volume and user traffic?
    3. Extensibility: How easily can new features and integrations be added to the system in the future?
    4. Development and Operations: What are the implications in terms of development complexity, deployment, maintenance, and operational overhead?

3. Evaluation Process:

•Each architectural alternative is evaluated against the defined criteria. For example:

- Usability: The Monolithic Architecture might offer a consistent user experience but could become complex and overwhelming for content creators as the system grows. The Microservices Architecture could provide a more modular and customizable user interface, but inconsistencies between services may affect usability. The Headless CMS Architecture might offer flexibility in designing frontend experiences but may require additional effort to customize the content management interface.
- Scalability: The Monolithic Architecture might have limitations in scaling due to its centralized nature. The Microservices Architecture could offer better scalability by allowing individual services to be scaled independently. The Headless CMS Architecture might provide scalability benefits by decoupling content management from frontend delivery, enabling horizontal scaling of each layer.

- Extensibility: The Monolithic Architecture might face challenges in adding new features without affecting existing functionality. The Microservices Architecture could offer flexibility in extending or replacing individual services to accommodate evolving requirements. The Headless CMS Architecture might provide extensibility through its API-driven approach, allowing seamless integration with third-party services and frontend frameworks.

4. Trade-off Analysis:

1. Trade-offs associated with each architectural alternative are analyzed, considering the implications on usability, scalability, extensibility, and other factors. For example:
    1. The Monolithic Architecture might offer simplicity in development and deployment but could limit scalability and extensibility.
    2. The Microservices Architecture could provide flexibility and scalability but may introduce complexity in managing distributed systems.
    3. The Headless CMS Architecture might offer decoupling of content management and presentation layers but may require additional effort in frontend development and integration.

5. **Selection of Preferred Architecture**:
    1. Based on the evaluation results and considering trade-offs, the preferred architecture is selected. For example, the Microservices Architecture might be chosen for its flexibility, scalability, and modularity, enabling the CMS to meet the project's requirements effectively.

Outcome:
Through Software Architecture Alternatives Evaluation, the CMS project evaluates different architectural options in depth, considering their alignment with project requirements, quality attributes, and constraints. This phase ensures that the final architectural decision is well-informed and balanced, ultimately contributing to the success of the CMS solution.