Faculty of Informatics and Computer Science

Software Design & Architecture

Topics 8 & 9

Software Scalability & Domain Driven Architecture

Dr. Ahmed Maghawry

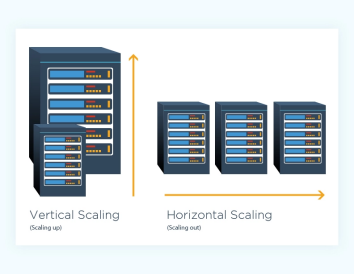
Contents

1. Introduction

- Software Design & Architecture practices lead to building a robust and successful software systems.
- Proper design decisions highly impact scalability, maintainability and overall performance.
- Scalability doesn't always mean upgrading our system to handle more users, it also involves efficiently utilize resources.
- DDA emphasizes understanding the domain, utilizing domain experts' knowledge, and building software that reflects the domain's concepts and behavior.
- DDA principles, such as bounded contexts and aggregates, can contribute to the scalability of a software system.

2. Software Scalability

Software scalability refers to the ability of a software system or application to handle increased workload or user demand without negatively impacting performance, responsiveness, or user experience.



The importance of software scalability lies in:

- its ability to ensure that software systems can meet the evolving needs of users and businesses.
- It allows applications to handle spikes in usage, support growing user bases, and scale resources both up and out as required.
- Scalable software systems provide a better user experience, maintain responsiveness even under heavy loads, and enable businesses to adapt to changing market conditions and demands.
- Scalability is a critical factor in building successful software solutions that can grow, evolve, and effectively serve their intended purpose over time.

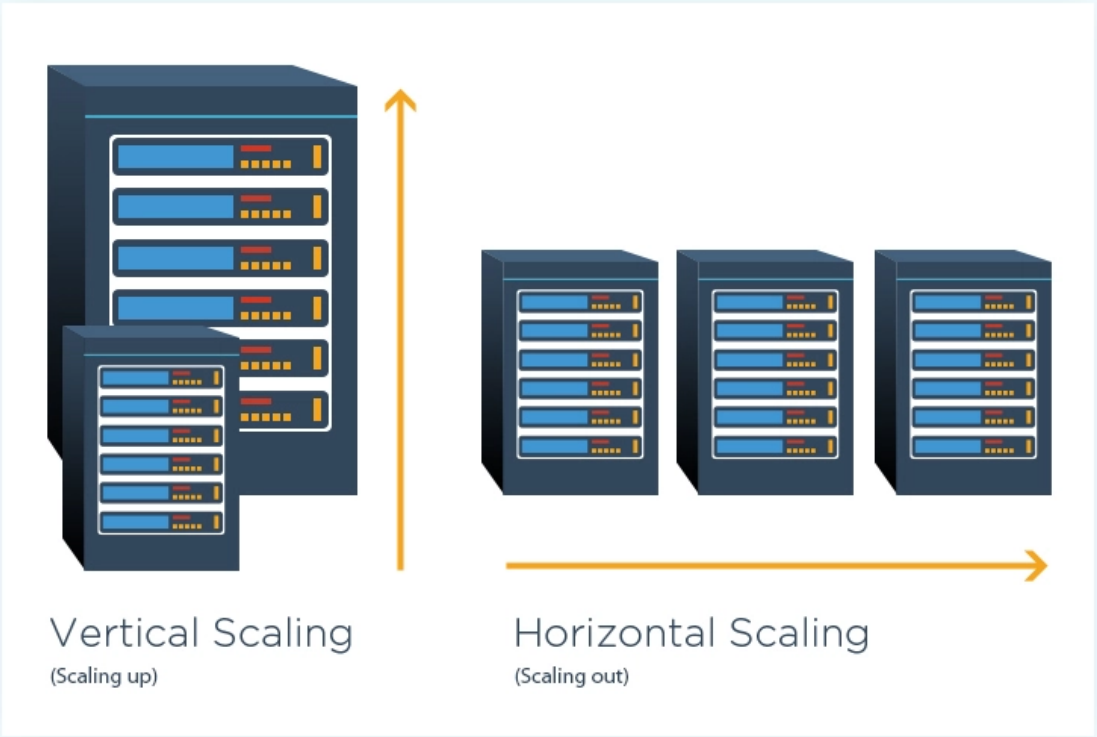
SW Scaling Types:

- **Vertical Scaling (Scaling Up):**
  - Vertical scaling involves increasing the capacity of a single server or machine to handle a larger workload.
  - This can be achieved by adding more resources such as CPU, memory, storage, or network bandwidth to the existing server.
  - Vertical scaling is typically limited by the hardware capabilities of the server and may reach a point where further scaling becomes impractical or cost-prohibitive.
  - It is well-suited for applications with a smaller number of users or when the workload can be efficiently handled by a single, powerful server.

- **Horizontal Scaling (Scaling Out):**
  - Horizontal scaling involves distributing the workload across multiple servers or machines to handle increased demand.
  - It is achieved by adding more servers to the system and load balancing the incoming requests across the available servers.
  - Horizontal scaling allows for improved performance, fault tolerance, and the ability to handle higher traffic loads.
  - It requires careful design and configuration to ensure that the workload is effectively distributed and that data consistency is maintained.
  - Horizontal scaling is well-suited for applications with a large number of users or when the workload can be parallelized and distributed across multiple servers.

2. Software Scalability

SW Scaling Types:



3. Software Scalability Challenges

- **Performance Bottlenecks:**
  - Bottlenecks can arise from inefficient algorithms, slow database queries, network latency, or resource limitations. Analyzing the system's performance, profiling the code, and optimizing critical components are necessary to overcome these bottlenecks.
- **Resource Limitations:**
  - Scaling a software system requires the availability of adequate resources such as CPU, memory, storage, and network bandwidth.
  - However, resource limitations can pose challenges, especially in vertical scaling scenarios where the maximum capacity of a single machine may be reached.
- **Data Consistency:**
  - Maintaining data consistency across distributed systems is a complex challenge in achieving horizontal scalability.
  - As the workload is distributed across multiple servers, ensuring that data remains consistent and synchronized becomes crucial.
  - Implementing proper data synchronization mechanisms, such as distributed transactions or event sourcing, is necessary to address this challenge.
- **State Management:**
  - In a distributed environment, where requests can be handled by different servers, maintaining session state or shared data becomes complex.
  - Implementing stateless architectures, utilizing shared caches, or employing distributed data stores can help overcome this challenge.
- **Communication and Coordination:**
  - Ensuring seamless communication, managing message queues, handling distributed transactions, and maintaining consistency across different services require careful design and implementation.
- **Testing and Validation:**
  - Validating the scalability of a software system is crucial but can be challenging.
  - Simulating and testing high load scenarios, identifying performance bottlenecks, and ensuring system reliability under heavy loads require comprehensive testing strategies and tools.
- **Cost and Complexity:**
  - Scaling a software system, especially horizontally, can introduce increased costs and complexity.
  - Adding more servers, managing distributed components, and maintaining the infrastructure can be resource-intensive and may require additional investments in hardware, networking, and operational resources.

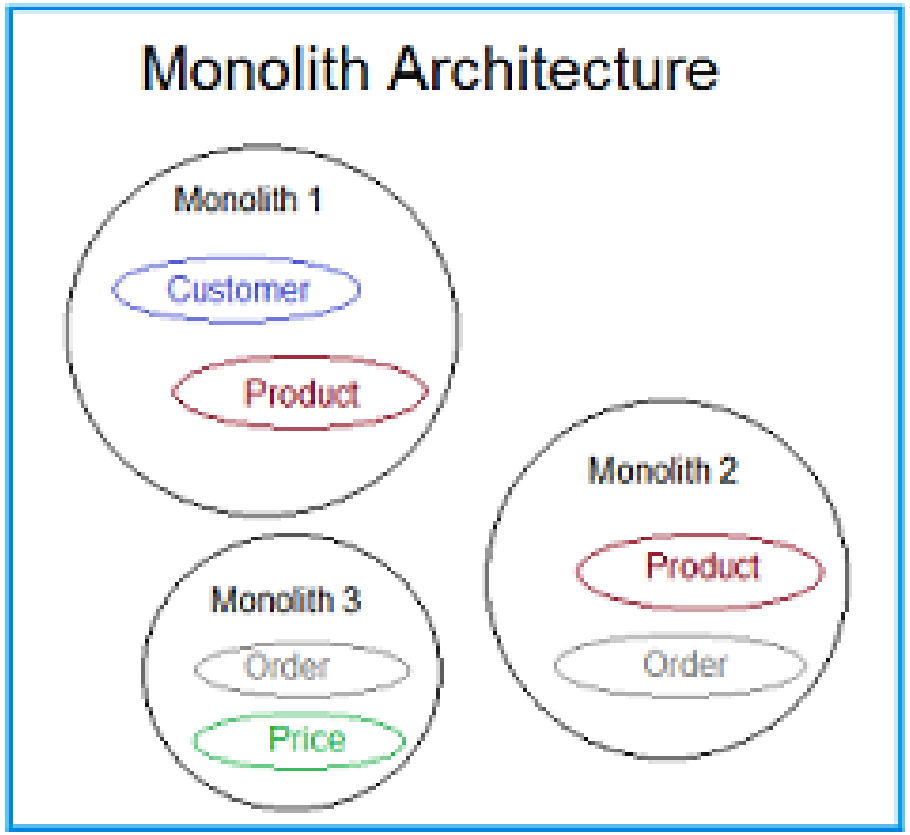
4. Domain Driven Architecture

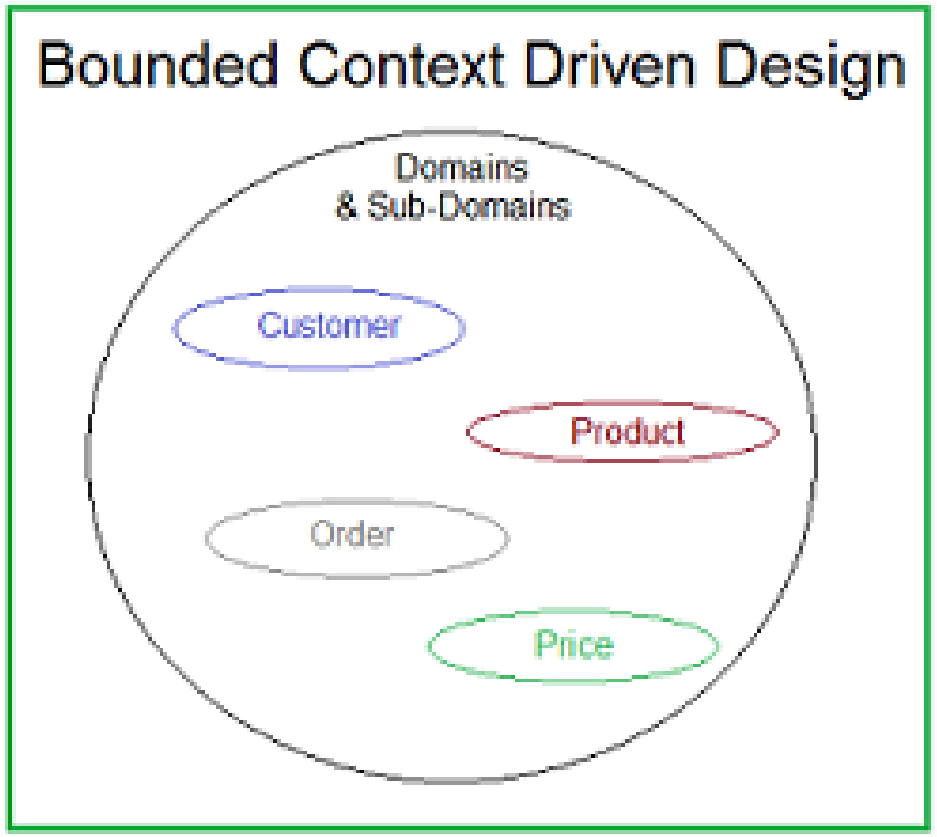
Domain-Driven Architecture (DDA) is an architectural approach that emphasizes aligning software design with the problem domain of the application. It is based on the principles and patterns introduced in Domain-Driven Design (DDD), a methodology for designing complex software systems.

Key Concepts of DDA

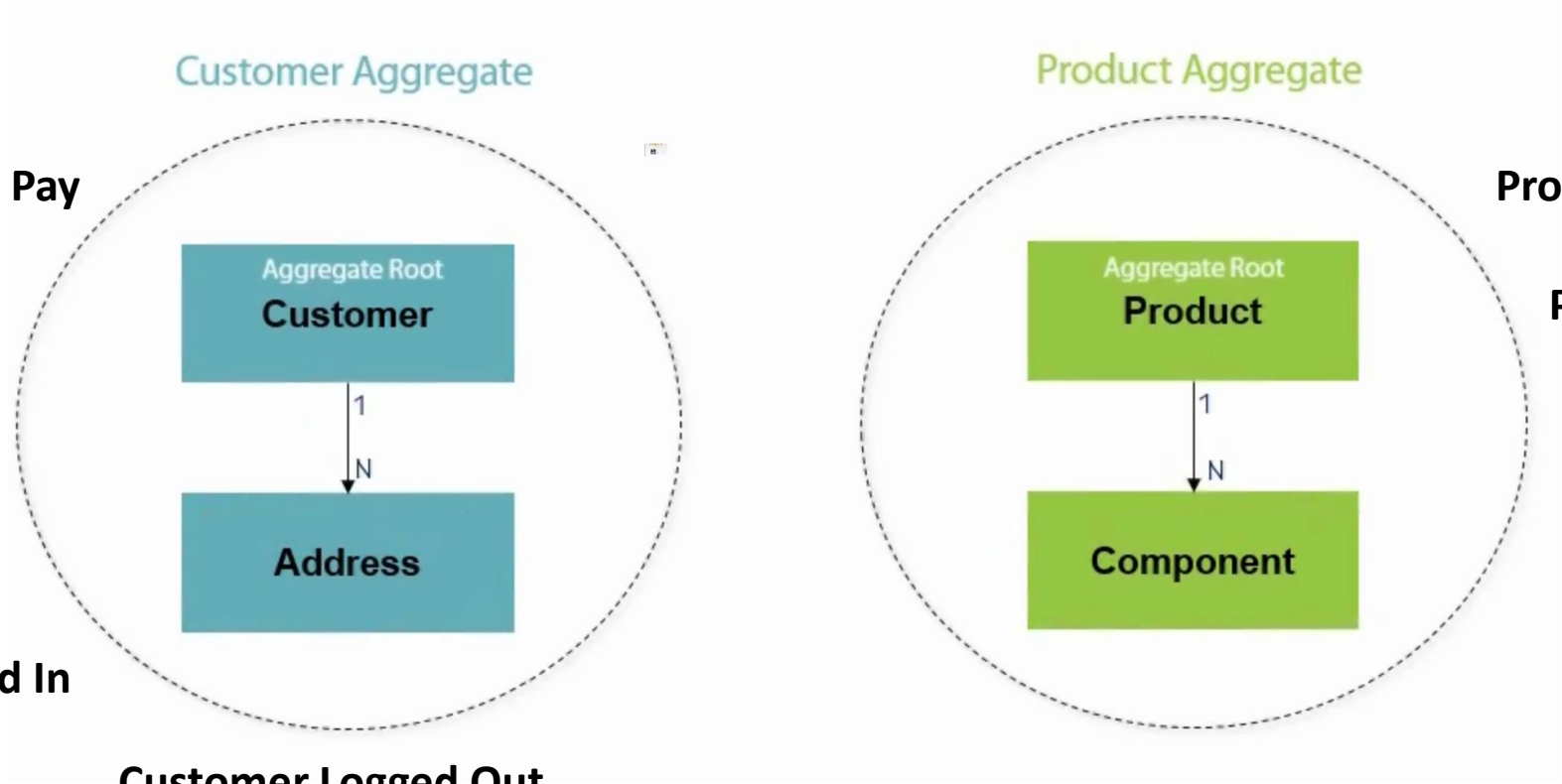
- **Bounded Context:**
  - A bounded context defines a clear boundary within which a particular domain model, language, and set of rules apply.
  - It encapsulates a specific subdomain within the larger problem domain.
  - Bounded contexts help manage complexity and enable different parts of the system to evolve independently.
- **Aggregates:**
  - Aggregates are cohesive clusters of domain objects that are treated as a single unit for data consistency and transactional boundaries.
  - Aggregates encapsulate the behavior and state of related objects and provide a consistency boundary within the domain model.
- **Domain Events:**
  - Domain events capture significant changes or occurrences within the domain.
  - They represent business-relevant incidents or facts that other parts of the system might be interested in. Domain events enable loose coupling and communication between different components of the system.
- **Ubiquitous Language:**
  - Ubiquitous language is a shared vocabulary and set of terms that bridges the communication gap between domain experts and technical team members.
  - It ensures that the domain model and codebase use a consistent and meaningful language, aligning with the understanding of the domain experts.
- **Context Mapping:**
  - Context mapping deals with the interaction and integration of different bounded contexts within a larger system.
  - It defines strategies and patterns for managing the communication, consistency, and collaboration between different bounded contexts.

5. Bounded Context





6. Aggregates & Domain Events



Customer Trying to Pay

Customer Logged In

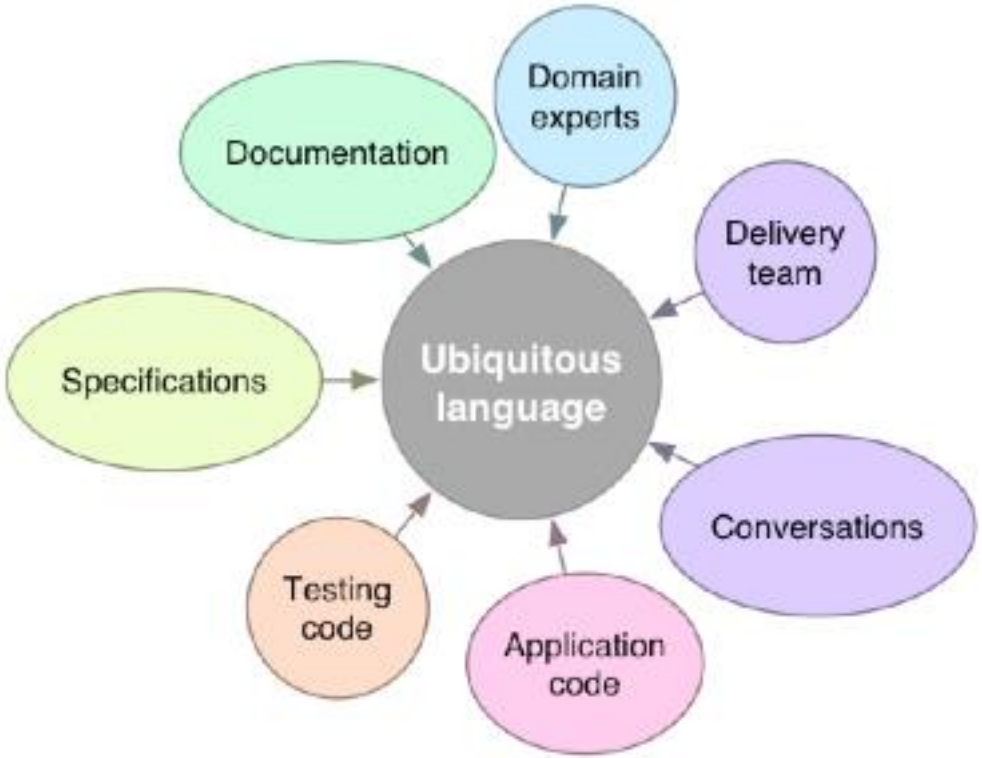
Customer Logged Out

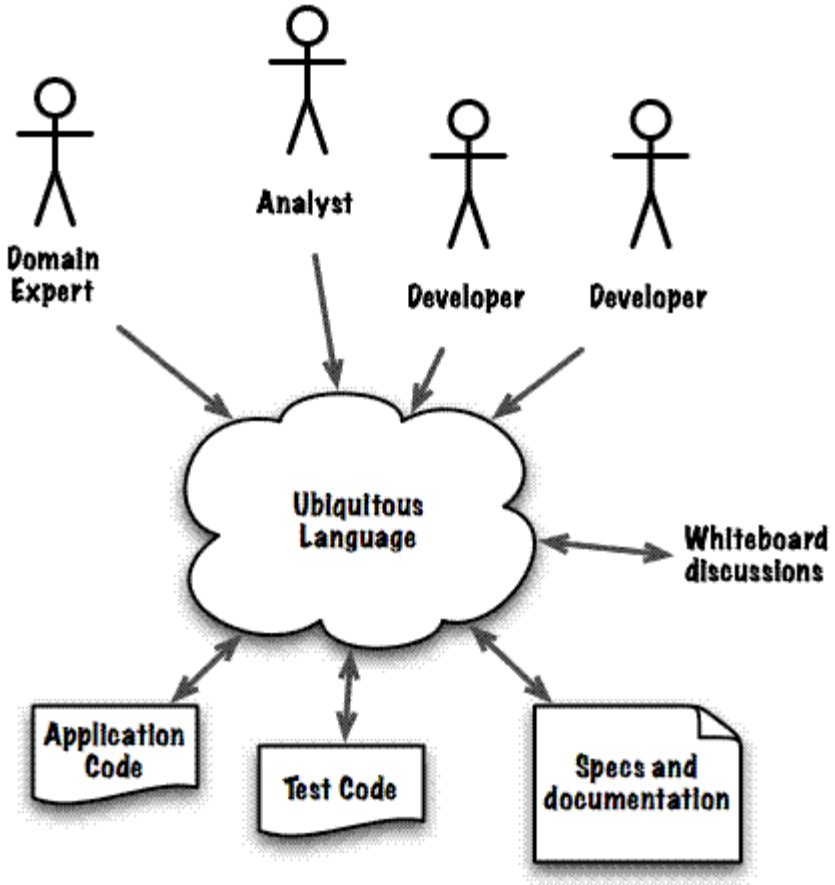
Product Added To Cart

Product Removed from Cart

Product Stock Changes

7. Ubiquitous Language





8. Scaling SW with DDA

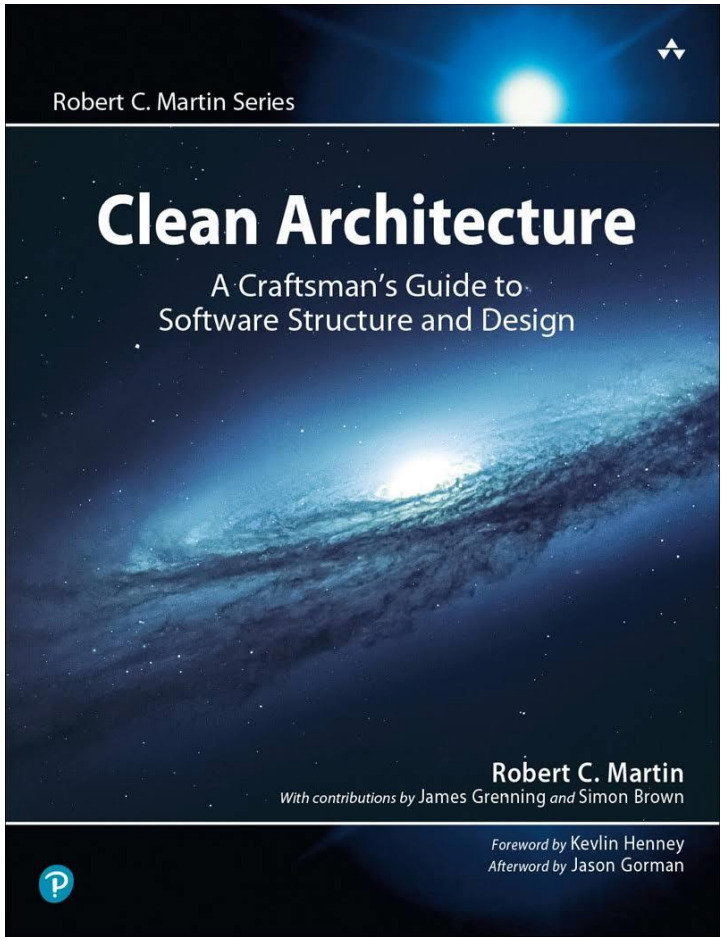
- **Bounded Contexts:**
  - Clearly define and identify the bounded contexts within the system. Bounded contexts allow for modularization and isolation of different parts of the system, which can be individually scaled as needed.
- **Aggregates:**
  - Design aggregates with scalability in mind. Aggregates encapsulate related domain objects and define consistency boundaries. When scaling, consider the distribution and partitioning of aggregates to distribute the workload across multiple instances or nodes.
- **Event-Driven Architecture:**
  - Leverage domain events to enable scalability and loose coupling. By using an event-driven architecture, you can distribute processing and scale specific components based on event types or event handlers.
- **Distributed Data Management:**
  - Scaling software systems often involves distributed data management. Consider using distributed databases, caching mechanisms, or data partitioning strategies to distribute and scale data storage.
- **Scalable Infrastructure:**
  - Scaling software requires a scalable infrastructure. Ensure that the underlying infrastructure, such as servers, network, and storage, can handle increased loads and traffic. Consider utilizing cloud-based solutions that offer elasticity and scalability.
- **Performance Optimization:**
  - Identify performance bottlenecks and optimize critical components of the system. Use profiling tools and techniques to identify areas that require optimization for better performance and scalability.
- **Load Balancing and Elasticity:**
  - Implement load balancing mechanisms to distribute the workload across multiple instances or nodes. Load balancing ensures that the system can handle increased traffic and evenly distribute the processing load.
- **Continuous Monitoring and Scaling:**
  - Implement robust monitoring and observability mechanisms to continuously monitor the system's performance, resource utilization, and scalability.

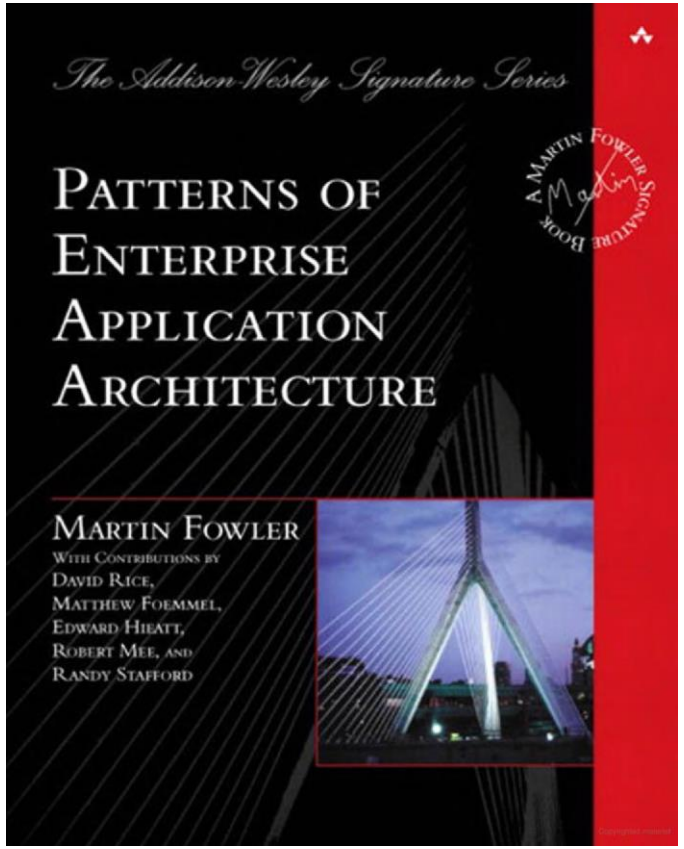
9. Conclusion

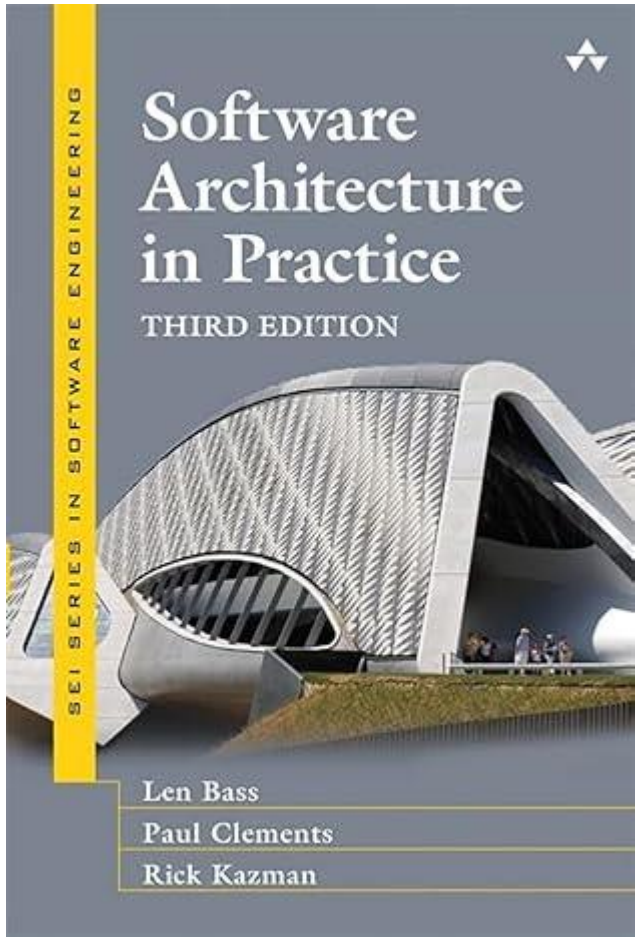
- Good software design and architecture leads to endless opportunities for future enhancement.
- Scaling software makes it upgradable to adapt to new circumstances.
- Bad software design makes it hard to scale that software.
- Good software design makes it easy to scale that software, hence, continue to operate.

References

- Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.
- Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice (3rd ed.). Addison-Wesley Professional.
- Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.







Faculty of Informatics and Computer Science

Questions?