Software Design Patterns and Architectural Best Practices in Financial Software Development

Ashmitha Nagraj

Software Engineer
nagrajashmitha@gmail.com

Abstract:
Financial software development involves designing and building solid, flexible and secure application systems that can process many complex transactions, ensure that an organization is compliant with various laws and regulations, and process large volumes of data.
This paper reviews some of the most common software design patterns and architecture designs used in software engineering to build financially reliable and maintainable applications. It also draws on previous work in software engineering to provide an overview of some of the most popular software design patterns and their uses in financial systems. Some examples of financial design patterns are used such as Singleton, Factory, Observer, and Event-Driven Architecture (EDA) to describe how financial systems such as Trading Systems and Risk Management Tools are designed. In addition to describing the use of design patterns in financial systems, this paper also reviews and expands upon existing case studies of large-scale financial systems to demonstrate how design patterns have been used to address challenges related to concurrency, data integrity, and extending the capabilities of financial systems. Finally, several "best practices" are outlined for developing financial software including using a modular design, minimizing coupling between components, and using the SOLID design principles for sustainable development. It is concluded by providing suggestions for how to apply these design patterns and architecture designs in modern financial software systems to develop systems that will be able to perform well in terms of speed and adaptability.

INTRODUCTION

Financial software must be both reliable when there are failures and secure against threats, which makes compliance with regulatory requirements inevitable. Reusable, effective solutions to common problems can be found using software design patterns, while guidance for building the overall architecture of a system can be found using best practices for architectural design. This study will discuss all these aspects within the context of financial software development using only pre-scholarly knowledge that provides a foundation for the research presented in this study.

Applications related to finance, such as banking systems, trading systems, and risk assessment tools have unique characteristics to their processing and storage needs, particularly regarding providing real-time processing, always being available and maintaining consistency in their data. These common issues can be addressed using design patterns, such as those developed. In addition, architectural frameworks provide scalable solutions to address the growth and complexity of large-scale financial systems [8]. Agile methodologies can also enhance the process of developing financial systems, as discussed in previous studies regarding financial information technology systems [11].

Previously, the literature regarding the implementation of these practices has evolved significantly. For example, a study describes enterprise patterns for separating concerns in financial workflow, such as the approval of loans or the management of assets [4]. Likewise, highlight the importance of integration patterns for allowing financial systems, such as exchanges and back-office operations, to communicate effectively [5]. The format of this study is as follows: a review of the relevant pre-sources of literature, specific discussions about design patterns and architectures including SOLID principles, expanded case studies and the conclusions with the recommendations for future directions.

LITERATURE REVIEW

The early literature on design patterns established the foundational base for designing reusable software components. This was further identified into twenty-three patterns that were classified into three categories like Creational, Structural, and Behavioral and highlighted their ability to provide flexibility due to their adherence to object-oriented design principles [7]. Due to the complexity of the financial domain, the identified design patterns are particularly well-suited for financial software applications because they allow developers to create objects that can represent real-world financial concepts, such as bank accounts and transactions.

The idea of design patterns was further developed into a financial context to create enterprise software systems based on the foundation of both object-oriented design and enterprise application architecture [4]. The enterprise design patterns support the separation of three major layers in an application: the layer for accessing data, the layer for presenting the user interface, and the layer containing the business logic. By implementing design patterns that separate the layers of an application, financial software developers can create complex workflow processes such as loan processing or managing a portfolio, while supporting strong architectural guidelines and maintaining the maintainability of their overall system.

In addition to developing patterns for specific areas of software design, a study promoted the idea of using a pattern-oriented approach when developing the architecture of large-scale software applications. They also recognized that in order to effectively implement this type of architecture, it would be necessary to develop systems of patterns, rather than individual patterns [2]. The importance of creating software architecture as a discipline and identified several key elements to consider when developing software architecture, including components, connectors, and configurations are discussed. The recognition of the need for a pattern-based approach to software architecture is especially important in financial software development, where it is often required to integrate with existing legacy infrastructure.

The importance of developing design patterns to support the integration of systems has been demonstrated by. Their research detailed the use of enterprise integration patterns for developing software applications that require the integration of multiple systems, such as those found in financial markets, such as stock exchanges and clearing houses.

Although many researchers have advocated for the use of design patterns in software development, a study emphasized the importance of developing software applications according to agile principles and practices and noted that one researcher had conducted research in the area of the use of Scrum in financial IT development.[7] As part of the research, they identified the benefits of using architectural extensions to support the development of new services in financial software applications.[11] Additionally, a study presented practical guidance for architects developing software applications, and included quality attributes, such as performance and security, which are both critical to financial software applications.[8]

A study proposed the 4+1 view model as a tool for helping describe the architecture of a software system from a variety of perspectives. Although the 4+1 view model was originally designed for use in the development of telecommunications systems, its use in describing architectures from multiple perspectives makes it a useful tool for communicating the needs of stakeholders in the development of financial software.[9]

Another study provided a comprehensive treatment of the different viewpoints and perspectives that are commonly associated with software architecture. It also emphasized the importance of understanding the needs of stakeholders in the development of complex systems, such as financial platforms.[10] As part of the research, a study presented domain-driven design, and modeled complex financial domains using bounded contexts and aggregates.[6] Another study demonstrated the practical application of design patterns in the development of software for pricing derivatives and showed how the design patterns could be used to improve the efficiency of the derivative pricing process in the development of financial software applications. [12]

There are several additional sources of information about the use of design patterns and other best practices in the development of financial software applications that have been published since the year 2010. One example includes the IBM Redbooks on Service-Oriented Architecture (SOA) in Banking, which presents case studies of the use of service-oriented architectures in financial institutions. Together, these sources provide the foundation for using design patterns and best practices in the development of financial software applications.

DESIGN PATTERNS IN FINANCIAL SOFTWARE

Design patterns solve recurring problems in software design. In financial development, they ensure efficiency, security, and scalability.

CREATIONAL PATTERNS

The Singleton pattern ensures a class has only one instance, useful for managing shared resources like a central configuration in trading systems to avoid inconsistent states [1]. In multithreaded financial environments, careful implementation prevents race conditions, often using double-checked locking.

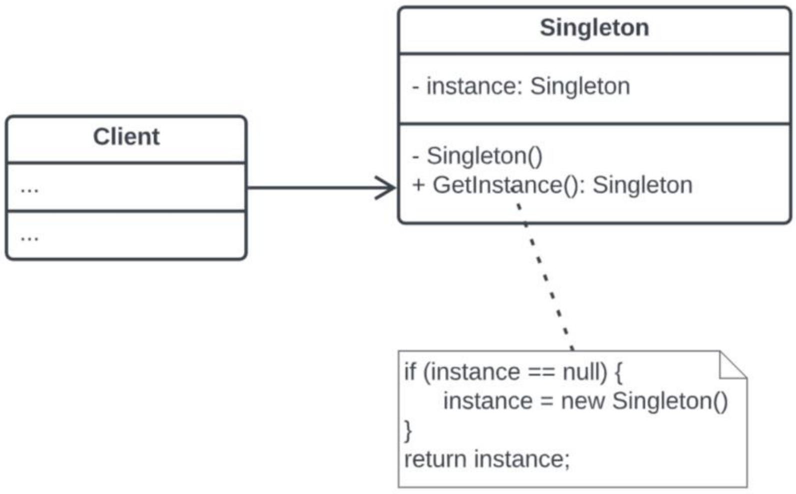

Figure 1: UML Diagram for Singleton Pattern

The Factory pattern abstracts object creation, allowing financial software to instantiate different transaction types (e.g., equity vs. derivative) without tight coupling [1]. This promotes extensibility when new financial instruments are introduced, adhering to the Open-Closed Principle.

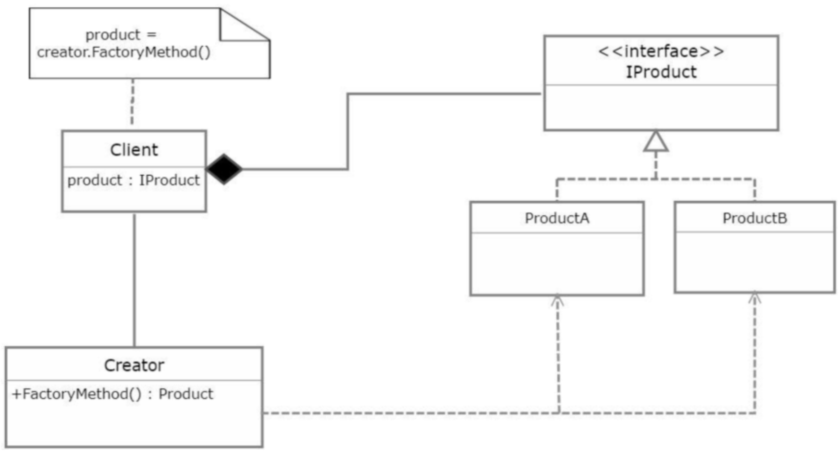

Figure 2: UML Diagram for Factory Method Pattern

BEHAVIORAL PATTERNS

The observer design pattern allows developers to create a dependency of one class on many other classes as well as to notify those classes about events happening to objects they depend upon. For example, if an organization has a financial dashboard that displays the latest information in real time and is updated by the current market conditions, it would be best to use an observer design pattern so that when there are any market changes, it will automatically notify all the subscribers to financial dashboard. Likewise, the observer pattern allows for the propagation of alerts from one module to another in a risk management system allowing for quick response to volatility in the markets. Observer can also work well in conjunction with event driven models in a trading system to provide efficient price feed handling [5].

STRUCTURAL PATTERNS

The adapter design pattern provides developers with the ability to interface with legacy financial systems that may require conversion of the incompatible interfaces to modern API's[1]. The decorator design pattern provides developers with the ability to add behaviors such as auditing to transactions without changing the underlying classes.

INTEGRATION PATTERNS

Integration Patterns describe how message channels and routers can be used for financial integrations to ensure the reliability of the data flowing between the different systems [5]. Content-based routers can also be used in high frequency trading to route orders based on certain criteria.

Pattern	**Application in Finance**	**Advantages**	**Limitations**
Singleton	Configuration management	Resource efficiency	Potential bottleneck in multithreaded environments
Factory	Transaction object creation	Flexibility in subtypes	Added complexity
Observer	Market data updates	Loose coupling	Overhead in notifications

Table 1: Compares common patterns in financial contexts.

ARCHITECTURAL BEST PRACTICES

Architectural best practices ensure financial software meets quality attributes like reliability and security [8].

LAYERED ARCHITECTURE

A study recommends layered designs separating presentation, business logic, and data access [4]. In banking apps, this isolates UI from core financial computations, allowing independent scaling.

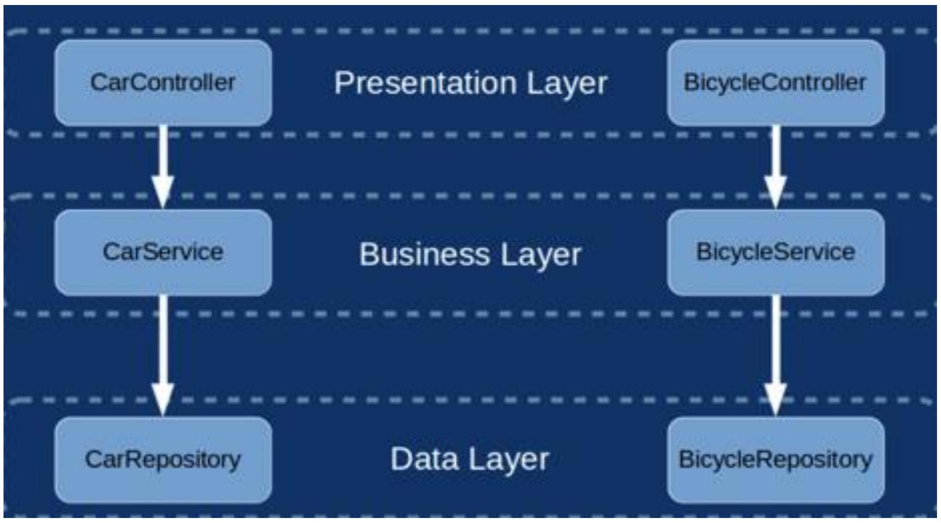

Figure 3: Diagram of Layered Software Architecture

EVENT-DRIVEN ARCHITECTURE

For high-throughput systems like trading platforms, event-driven models handle asynchronous events, improving responsiveness [5]. This is critical for processing market data streams without blocking.

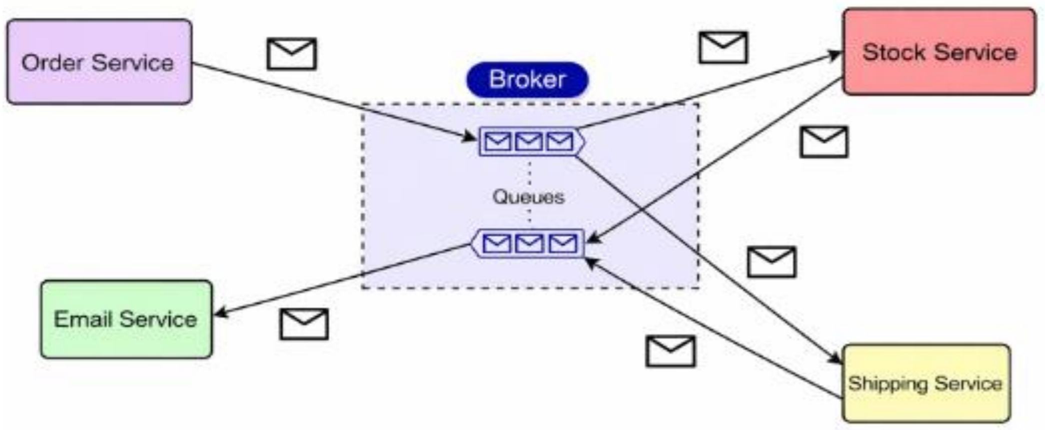

Figure 4: Event-Driven Architecture Diagram for Financial Trading System

MICROSERVICES VS. MONOLITHS

A study discusses trade-offs; microservices offer scalability for financial services but increase complexity [8]. In fintech, decomposing monoliths into services allows independent deployment.

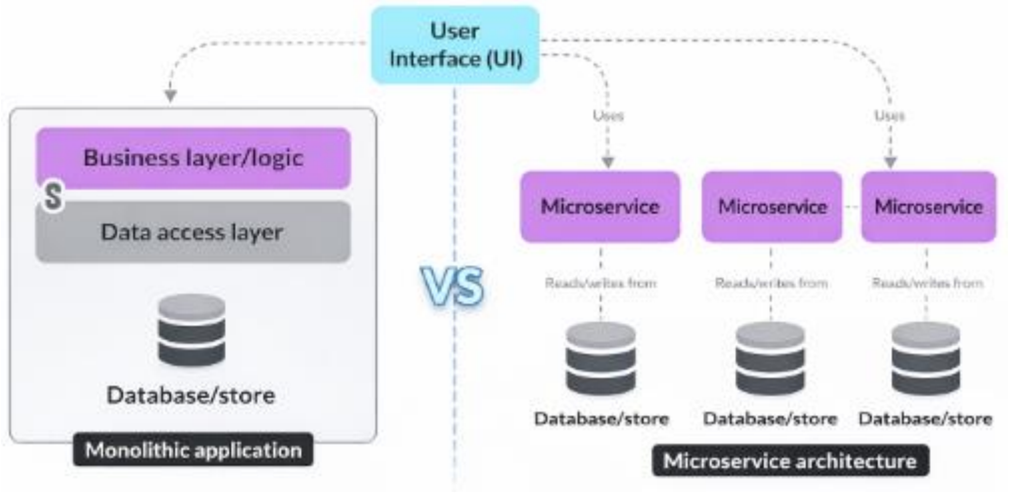

Figure 5: Comparison Diagram of Monolith vs. Microservices Architecture

SOLID PRINCIPLES

SOLID principles were developed to enhance the design quality of object-oriented programming so that it is easier to evolve and modify [7]. They provide support for financial software to respond to changing needs without modifying a large portion of the software.

1) **The Single Responsibility Principle (SRP):** A class should only be responsible for one thing. In the world of banking, separating classes for logging transactions and validating transactions will prevent bloat [7]. For instance, in this scenario, a TransactionProcessor class would handle processing a transaction, while a separate class called Auditor would manage the auditing of a transaction to reduce bugs in high-risk situations. Therefore, when there are regulatory changes affecting unrelated code, the impact will be lessened.
2) **The Open-Closed Principle (OCP):** Software entities should be open for extension but closed for modification. Using a base Asset class in a Portfolio Management application, new instrument, such as a cryptocurrency can be added, without having to alter existing code to ensure stability [7]. As a result, financial applications will be able to incorporate new financial products available in the market with ease.
3) **The Liskov Substitution Principle (LSP):** Subclasses must be capable of being substituted for their base classes without causing any unintended side effects. In payment systems, a CreditCardPayment class must act similarly to a Payment class without producing any unforeseen results, to maintain consistency in the flow of transactions [7]. It may seem minor, however, if a violation occurs it can produce subtle errors in financial computations.
4) **The Interface Segregation Principle (ISP):** Clients should not depend upon interfaces they do not use. For tools used to assess risks, break up a large interface called IFinancialService into multiple interfaces, such as IRiskCalculator and IReportGenerator, to avoid creating dependency between clients and interfaces they do not use [7]. By breaking down the interfaces, the number of coupling points in the modular financial modules is reduced.
5) **The Dependency Inversion Principle (DIP):** High level modules should not depend on low-level modules; both modules should depend on abstractions. In trading platforms, instead of depending on a concrete implementation of an IMarketDataProvider, allow them to depend on the abstraction, which enables swapping out different implementations without requiring any modifications to the core [7]. The increased testability and flexibility of vendors are two benefits of applying the DIP in financial systems.

Using SOLID in finance has resulted in more modular systems as described in a study review of agile adoption [11].

S
O
L
I
D
Single Responsibility
Do one thing well
Open-closed Principle
Extend without modifying
Liskov Substitution
Behavior must not change
Interface Segregation
Many small interfaces are better
Dependency Inversion
Depend on abstractions
Single Responsibility
Do one thing well
Open-closed Principle
Extend without modifying
Liskov Substitution
Behavior must not change
Interface Segregation
Many small interfaces are better
Dependency Inversion
Depend on abstractions

Figure 6: Infographic Diagram Explaining SOLID Principles

OTHER BEST PRACTICES

- Use domain-driven design [6] to model financial domains, creating bounded contexts for areas like credit risk.
- Ensure security through patterns like secure pipes [2].
- Adopt 4+1 views for comprehensive architecture documentation [9].

Architecture	Suitability	Pros	Cons
Layered	Core banking	Modularity	Performance overhead
Event-Driven	Trading	Scalability	Event management complexity
Microservices	Fintech apps	Flexibility	Deployment challenges

Table 2: Compares architectures for finance.

CASE STUDIES

These examples show how the design patterns and architectures used in these applications help solve real-world problems in the finance area.

APPLYING LAYERED ARCHITECTURES TO MOBILE BANKING SERVICES IN A MONOLITHIC SYSTEM

A study describes the adoption of Scrum in the IT department of a large financial institution that had to extend its monolithic system to provide mobile banking services [11]. A layered architecture was applied to separate different concerns [4]. Presentation layers for mobile user interfaces and business logic for transactions were developed. Observer patterns were used to send real-time notifications to users when their accounts are updated, i.e., when there is a balance update or a fraudulent attempt occurs.

There were some significant challenges. Legacy mainframes could be integrated into the system only by means of adapters [1] and enterprise integration patterns such as message translators [5]. The SOLID principles [9] were applied during the refactoring process. For example, the Single Responsibility Principle (SRP) was applied to separate service logic from data access, and the Dependency Inversion Principle (DIP) was used to abstract over the legacy database. As a result, it took 30% less time to deploy the system, compliance was improved due to modular testing, and the system became scalable enough to handle the increasing volume of mobile requests [5]. This case study illustrates how agile methods can be combined with design patterns to create incremental changes in regulated environments.

IMPLEMENTING SERVICE ORIENTED ARCHITECTURE IN BANKING: TRANSFORMING JKHL BANK

In IBM Redbooks a case study on JKHL Bank is presented, which is a fictional but typical financial institution that implemented Service Oriented Architecture (SOA) to modernize its core banking and customer care systems [13]. The architecture applied a layered design, where services act as components [8]. Factory patterns were applied to create service instances dynamically depending on the type of customer.
Integration patterns were described, which were used to enable messaging between silos of the bank [5]. Event-driven elements were used to manage asynchronous updates, e.g., to notify risk modules about transaction events. Data consistency issues between the distributed services were solved by using transactional integrators and secure pipes [2].

JKHL Bank was able to improve its operational efficiency and reduce its operational risks by integrating information across all silos of the bank. It was possible to offer better services to customers and therefore improve the customer satisfaction. The case study demonstrates how SOA can contribute to cost savings and growth in financial systems.

APPLYING DOMAIN-DRIVEN DESIGN TO DERIVATIVES PRICING: USING QUANTLIB

A study describes the usage of design patterns in C++ for developing the QuantLib library for derivatives pricing [12]. The system modelled the complex financial products using Domain-Driven Design [6], with bounded contexts for pricing models and risk calculation. Factory and Strategy patterns [1] made it possible to create and replace the pricing algorithms at runtime, so that new financial products could be supported. Observer patterns were used to propagate market data changes to dependent calculations to ensure the accuracy of the results in real-time.

Computational intensity was a challenge in quantitative finance and was overcome by applying concurrent patterns from a study [2] to process tasks in parallel. SOLID principles [9] were also adhered to during the development. For instance, the Open-Closed Principle (OCP) was applied to allow extensions to models without modifying them. The Liskov Substitution Principle (LSP) was applied to ensure substitutability of the instrument classes [7]. As a result, it was possible to develop new pricing tools quickly, to improve the accuracy of risk assessments and to integrate the system with trading platforms [3]. The case study illustrates how design patterns support precision in high-risk financial computing.

JOHN DEERE’S GLOBAL IT TRANSFORMATION: AN EXAMPLE FOR FINANCE

Although John Deere’s global IT transformation, does not directly relate to finance, it has similarities. Scrum@Scale and DevOps were used to modernize infrastructure in a manner like what financial IT transformations would require [8]. Layered and microservice-based architectures decomposed monolithic systems, using DIP for dependency management [8]. Event-driven patterns were used to handle data flows in supply chain modules; they correspond to the event-driven elements found in transaction processing.
In summary, the above case studies demonstrate how design patterns and practices lead to stable, flexible and compliant financial software systems, capable of addressing integration, scalability and compliance-related challenges.

CONCLUSION

Design Patterns and architectural best practices are required for financial software to address Complexity and Reliability issues. The expanded case studies illustrate the practical benefits of these approaches i.e., Efficiency Gains, Risk Reduction. Future work could build upon these foundations by integrating emerging concepts like SOA Extensions. Developers should focus on modularity, adherence to SOLID Principles, and agile integration to meet evolving demands of the Financial Industry, developing secure, scalable and maintainable systems.

REFERENCES:

1. Gamma, Erich & Helm, Richard & Johnson, Ralph & Vlissides, John. (1993). Design Patterns: Abstraction and Reuse of Object-Oriented Design. 406-431. 10.1007/978-3-642-48354-7_15.
2. Buschmann, Frank & Henney, Kevlin. (1993). Pattern-oriented software architecture.
3. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline, vol. 1. Prentice Hall, Englewood Cliffs (1996)
4. Martin Fowler. 2002. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., USA. isbn:0321127420.
5. Gregor Hohpe and Bobby Woolf. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley
6. E. Evans and E. J. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2004)
7. R. C. Martin, *Clean architecture: A craftsman's guide to software structure and design.* Pearson Education, 2018.
8. Bass, Len & Clements, Paul & Kazman, Rick. (2003). Software Architecture In Practice.
9. Kruchten, P.B. (1995). The 4+1 View Model of Architecture. *IEEE Softw., 12*, 42-50.

10. Beidler, J. (2012). Rozanski, Nick. Software systems architecture: working with stakeholders using viewpoints and perspectives. *CHOICE: Current Reviews for Academic Libraries*, *49*(10), 1914. https://link.gale.com/apps/doc/A291615717/AONE?u=anon~f68afc1&sid=googleScholar&xid=f5a62207.
11. Storey, Margaret-Anne & Treude, Christoph. (2019). Software Engineering Dashboards: Types, Risks, and Future. 10.1007/978-1-4842-4221-6_16.
12. Mayo, Anita & Wong, Sherman. (2006). C++ Design Patterns and Derivatives Pricing by Mark Joshi. SIAM Review. 48. 178-180. 10.2307/20453775.
13. Papazoglou, Mike & Heuvel, Willem-Jan. (2007). Service oriented architectures: Approaches, technologies and research issues. The VLDB Journal. 16. 389-415. 10.1007/s00778-007-0044-3.