Patterns of Service Oriented Architecture

Adeeb Ahmed, *Undergraduate Student* Christopher Doherty, *Undergraduate Student*
and Eric Tendian, *Undergraduate Student*

I. Introduction

MODERN software systems can be extremely complex, so to address the complexity there is a need for abstraction. Such abstraction can make software systems more manageable. This abstraction needs to follow certain principles. Design patterns are often introduced to deal with common problems that are encountered when building software systems. One such common problem in service oriented architecture is ensuring that many different services can interact with each other, producing and consuming data. There are a variety of ways to make service oriented architecture more clean and avoid problems that arise when logic is separated into different modules. This paper will discuss three of them, the Enterprise Service Bus, the Mediator design pattern, and Aspect-Oriented Software Development.

II. Enterprise Service Bus

The Enterprise Service Bus (ESB) is based off a concept found in the architecture of computer hardware, the bus, applied to work with modular operating systems. [2] Roy Schulte's 2002 paper on the concept is credited with the first published use of the enterprise service bus term. [1] [6] It is an architectural design that allows for software components to be put together easily. The components can be used in a standalone fashion, requiring zero dependencies on any other software to run being easily combined with other modular programs to extend their usability and capabilities. ESB is commonly found integrating loosely coupled software components commonly referred to as services. [2] Service oriented architecture implements the ESB pattern to extend the usability of any web service, local to a network or over the Internet.

A bus in computer hardware terms allows for any piece of hardware to be connected to the computer, as long as it fits in the bus's port. As such, the integration of any hardware or software in the case of an ESB, is limitless. With the use of a bus, any software can be integrated to communicate with any other piece of integrated software.

The primary duties of an ESB is to monitor and control routing between the different pieces of software, resolve connection problems, control deployment and versioning of software, eliminate the need for redundant services, and cater for commodity services like event handling and queuing.

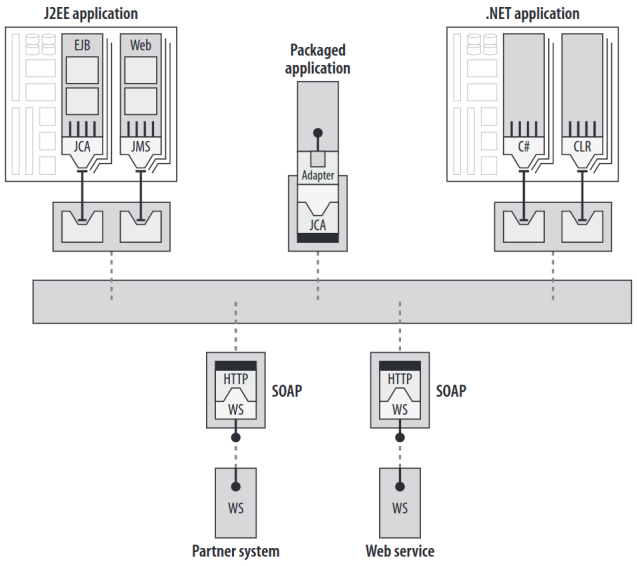

Fig. 1. The enterprise service bus in use, demonstrating its ability to deal with many different types of applications. [2]

The ESB plays an important role in service oriented architecture, serving many different roles as described above. Without the ESB, applications with different technology stacks as shown in Figure 1 must develop logic to talk to each other. Such logic can be time consuming and may involve duplication of logic and increase coupling. However, a lack of an ESB means eliminating a single point of failure when accessing the various applications. There could also be a performance boost as the communication between services is more direct.

III. Mediator Design Pattern

The mediator design pattern defines an object that encapsulates how a set of object interact. It promotes loose coupling of software by keeping the objects from explicitly referring to other objects and lets their interactions vary independently. This pattern endorses the concept of many-to-many relationships by integrating the interactions by peers to full object status. [3]

The problem this pattern was designed to solve was the issue known as spaghetti code. It aimed to solve the dependencies between potentially reusable pieces of code. If we take users accessing software as an example, without this pattern each user would need to independently query each piece of software individually to receive a coherent set of data. With this pattern, a user would only need to use the design once to receive the same set of coherent data. [3]

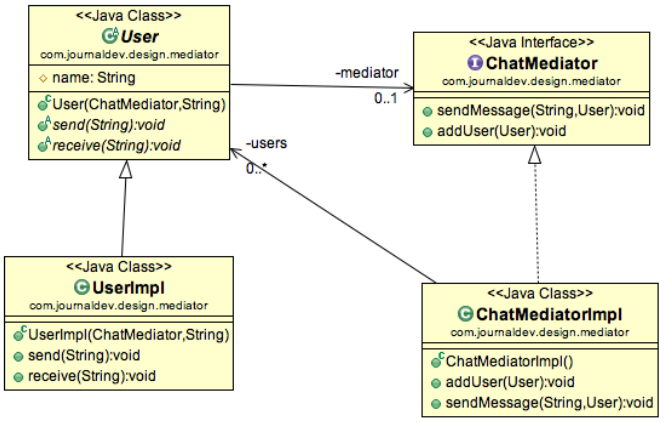

Fig. 2. An example of the mediator pattern in a chat application. [5]

Similar to the ESB this pattern identifies one point of entry to several software pieces. The pattern in short abstracts the need to independently communicate with each piece of software and implements once point of contact with each piece of software.

IV. Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) is a relatively new paradigm in software development, in contrast with others such as object-oriented programming and functional programming. It has been catching a lot of support lately and is expected to to grow as the practice advances. The principles of AOSD include techniques for managing modular dependencies with the overall goal being to easily evolve the software and and reuse code. AOSD is essentially an extension of common software design principles such as having low coupling, high cohesion, and proper separation of concerns. [4]

Separation of concerns is an important principle for AOSD, as programs which are developed with AOSD should have concerns defined accurately and broken up. Such concerns may be about caching or security, for example, which apply to not just one class but many different classes. Applying object-oriented principles, the proper approach would be to create a class for each concern and have classes inherit from it. As the mediator pattern points out, such inheritance can become quite messy, particularly when one concern may rely on another. [4]

However, AOSD takes a different approach by identifying such concerns which apply to many different classes as ”cross-cutting concerns”. In that situation, the cross-cutting concerns become their own ”aspects”. [4] Looking at the example of a Bank class, there are cross-cutting concerns such as authentication and logging. Taking the AOSD approach, a new aspect (similar to a class in code) would be created for authentication, and another one for logging. Then, as the Bank class is being called to do an action such as deposit or withdraw money, the aspects can handle the work of ensuring the user is authenticated, and log the actions they take to their account.

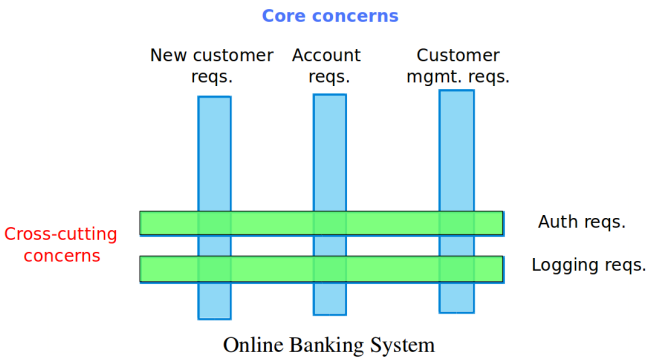

Fig. 3. Diagram of cross-cutting concerns in a banking application.

AOSD is similar to the mediator design pattern and enterprise service bus as it solves the abstracted problem of needing to deal with many cross-cutting concerns. Both the ESB and mediator pattern can be seen as opportunities for AOSD to be applied, particularly in the case of an ESB which performs many functions on top of routing.

V. Conclusion

The three architecture patterns discussed here are actually quite similar. Such patterns can be applied to solve certain service oriented architecture problems. However, often they do not appear together. By providing more context around the similarities these patterns have, we hope to present new opportunities for the patterns to be combined and create an elegant software architecture.

References

[1] R. S. Bhadoria, N. S. Chaudhari, G. Tomar, and S. Singh, *Exploring enterprise service bus in the service-oriented architecture paradigm*. Hershey, PA: IGI Global, 2017.

[2] D. A. Chappell, *Enterprise service bus*. Sebastopol, CA: O'Reilly, 2004.

[3] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison-Wesley, 1995.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, Aspect-oriented programming, *ECOOP'97 Object-Oriented Programming Lecture Notes in Computer Science*, pp. 220242, 1997.

[5] Mediator Design Pattern in Java, *JournalDev*, 02-Aug-2016. [Online]. Available: http://www.journaldev.com/1730/mediator-design-pattern-java. [Accessed: 30-Apr-2017].

[6] R. Schulte, Predicts 2003: Enterprise Service Buses Emerge, *Gartner Research*, 08-Dec-2002. [Online]. Available: https://goo.gl/zGLE3j. [Accessed: 30-Apr-2017].