Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2022, 9(4):172-178**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Handling Reusability & Interoperability Issues with Component-Based Architecture: Solutions & Use Cases

Chakradhar Avinash Devarapalli

Software Developer
Email: avinashd7[at]gmail.com

Abstract Component-based architecture (CBA) has recently emerged in the world of modern software development, providing engineers with a modular and reusable approach for the development of various systems. This article goes through the challenges and solutions to CBA and highlights some of the key issues, like reusability or customizability. It also deals with the issues of loose coupling or decomposition of large applications, and it looks into interoperability. By studying these problems from all angles, the paper gives suggestions and ways to solve these problems efficiently.

Keywords Component-Based Architecture, Reusability, Customizability, Decomposition, Interoperability, Software Development, Domain-Driven Design

Introduction

In software development, a very common way of ensuring improved development efficiency is by breaking larger chunks of a design element or application into smaller "components." Each component acts as a standalone building block of the final build and is considered to be a reusable, interchangeable element. The use of component-based architecture (CBA) has evolved the way developers work, not only improving work efficiency, but also reducing the bug-audit timeframe of builds. Instead of having to develop the entire thing in one go and then testing it, each standalone component can be checked and integrated independently.

While each component has a specific functionality thanks to its fractal nature [1], they are designed to work together seamlessly to create a comprehensive and cohesive system. This approach of breaking down complex software systems into smaller components has gained popularity due to its ability to promote code reusability, modularity, and maintainability. However, like any other development methodology, component-based architecture also faces certain challenges that can impact the success of a software system. These challenges include issues related to communication and coordination among different components, reusability and customizability, difficulty in managing dependencies between components, decomposition of large applications, ensuring compatibility and interoperability between components, and handling versioning and updates of individual components [2].

This paper will focus on three main issues: reusability and customizability, decomposition of large applications, and component interoperability.

Literature Review

Various sources have been used to study component-based architecture (CBA) challenges and the potential solution to these challenges.

For instance, T. C. & J.-B. S. considered CBA through the Fractal initiative, highlighting its general challenges and implementation hurdles. They provided a comprehensive overview of CBA, touching upon its core principles such as reusability, modularity, and maintenance.

In the academic domain, P. C. Clements' work on *Software Architecture In Practice* considered real-world insights into component-based systems and a use case, providing invaluable perspectives on architectural design and implementation.

M. R. P. S. Z. D. and I. Cardei delved into architectural decomposition and design automation, offering methodologies for breaking down large software systems into reusable components.

Transitioning from component-based architectures to microservices was studied by M. L. F. L. & M. M., highlighting the evolution of software architectures in response to changing technological landscapes. Meanwhile, dshaps provided insights into designing and implementing component-based architectures, offering practical advice for developers navigating the intricacies of CBA.

Challenges Surfaced

Component-Based Architectures are considered to be one of the most efficient ways of software development due to their reusability and modularity. They provide developers with the flexibility to create complex software systems by combining smaller, independent components. Just like any other development method, though, it is not perfect. There are some critical elements that need attention for this approach, too [3].

Component Reusability Limitations

Reusability versus Customizability presents a key issue in terms of Component-Based Architectures (CBA). While CBA offers the advantage of reusing its components across applications or different parts of an application, this reusability often comes at a price. It can reduce the customizability of said component the more it is reused.

For instance, generic components, designed for broad reuse, may not align with the specific requirements of a particular new use case. This limitation is particularly pronounced in front-end development, where user interface components often necessitate extensive customization to match the application's unique look and feel.

While it is possible to reuse the code and upgrade it, the reusability may end up requiring extensive code overhaul, which may take more time than writing new code in some instances.

Decomposing Components & Its Constituents

Decomposing large applications poses another significant challenge. Breaking down a sizable application into smaller, independent components demands a deep understanding of the application's functionality and inter-component interactions. In front-end development, this complexity becomes even more prominent due to the visual and interactive nature of user interfaces [3] [4].

Component Interoperability Limitations

And finally, component interoperability adds another layer of complexity. In CBA, components must seamlessly collaborate to deliver desired functionality. However, ensuring interoperability, especially across components developed by disparate teams or vendors, is a grueling task.

This issue is exacerbated in front-end development by the large number of available technologies and frameworks out there, not to mention an equally large number of libraries, elements, methodologies, code options, and more.

Components crafted using different technologies may fail to integrate seamlessly, leading to inconsistencies, performance issues, or even application failures.

These challenges become even more prominent in front-end development and require prompt solutions.

Balancing reusability and customizability calls for a nuanced approach. Designing components with a flexible API is key, allowing developers to inject custom content or behavior. This flexibility allows businesses to tailor components to their specific needs. Utilizing CSS-in-JS libraries further enhances customization, enabling dynamic styling based on props, thus maintaining reusability while offering ample flexibility.

Solution

Components, like people, vary in their offerings, catering to diverse needs. Sharing these elements via APIs grants businesses access to ready-to-use components for their applications or software. To ensure code is both decomposable and highly customizable, certain characteristics must be integrated into the system's architecture:
Components should be self-contained with interfaces communicating through ports.
The system should be decomposable into reusable, cohesive, and independent components.
Modules built from components should be expandable and updatable without necessitating adjustments.

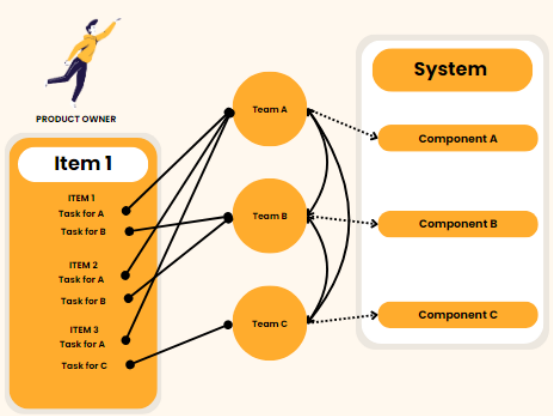

Figure 1: Component Team Structure

Component Teams for Better Development

Establishing component teams introduces a novel approach to collaboration. These cross-functional Agile teams focus on producing specific components, promoting efficiency and expertise in component development.
Component teams prove invaluable when dealing with legacy technology, complex algorithms, security, and compliance. Their diverse expertise, spanning design, development, and testing, ensures the delivery of refined components.
Effectively decomposing large applications requires a profound understanding of the application's domain and the interactions between its parts. Techniques like Domain-Driven Design (DDD) and tools such as Storybook and Atomic Design aid in visualizing and managing components.
A key aspect of the solution involves closing the semantic gap by using a common language for product requirements and component capabilities. The OPP Design Language (ODL) serves as this language, employing the Ontology Web Language (OWL) to represent domain-specific ontologies.
Ontologies prove beneficial in software engineering projects, extending the idea of reuse to the modeling level. In the OPP project, prototype ontologies are developed with limited scope, covering mobile systems and applications. The input to the methodology includes component and UML classifier metadata in ODL, UML models in XMI, and formal ODL language-specified requirements.
The methodology yields partial UML structural diagrams, component configuration files, and reports on requirements consistency. The ODL, defined by the Top-down Software Decomposition team, covers domains like mobile applications, cell phone systems, hardware/software components, and MOF metaschema. Protegé is used as an ontology modeling tool for ODL definition. The flow for the decomposition can be seen in Figure 2 below [2].

Top-Down Decomposition System

Requirements specification involves marketing and product management using modeling tools to generate ODL ontologies. Component specification, in the "evolved" stage, assumes a stable population of components. UML components are tagged with metadata in ODL, describing their semantics. Design modeling involves creating software models based on requirements, utilizing the Top-down Software Decomposition methodology.

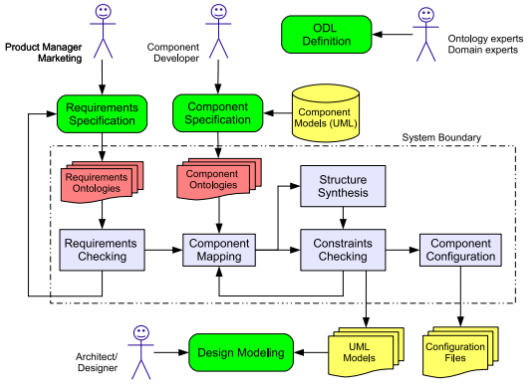

Figure 2: Top-down Software Decomposition methodology flow.

Functional components of the top-down decomposition system include requirements checking, component mapping, structure synthesis, constraints checking, and component configuration.

To ensure interoperability between components, establishing clear and consistent conventions for component design and communication is crucial. This involves maintaining consistent naming conventions, defining a well-structured component API, and creating a shared understanding of state management strategies.

Classification for Devices and Services

The proposed IoT architecture defines five major classifications for devices and services:

1. Central control system component,
2. End devices component,
3. Data-driven feedback model component,
4. External data input component, and
5. Software application component.

Each component classification serves specific functions within the architecture, facilitating efficient creation and management of services, as can be seen in Figure 3.

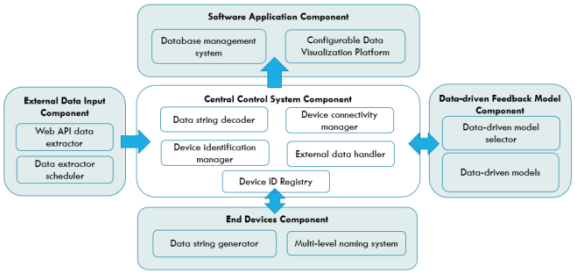

Figure 3: Proposed Solution for better component interoperability.

Central Control System Component

This component hosts the central platform and server responsible for managing interactions between components. Sub-components include the data string decoder, device identification manager, device connectivity manager, external data handler, and device ID registry.

End Devices Component

This component covers the lowest-level hardware and includes sub-components such as the data string generator and the multi-level naming system. The data string generator aggregates sensor data into a standardized format, while the multi-level naming system facilitates device ID creation and data segregation based on application domains and services.

Data-driven Feedback Model Component

This component provides a development environment for data-driven models, allowing users to build, select, or update models for optimization and prediction. Sub-components include data-driven models and the data-driven model selector, which facilitate model creation, selection, and performance evaluation for end devices.

External Data Input Component

This component handles reference data not directly obtained from end devices, such as API data from weather forecast stations. Sub-components include the web API data extractor and the API data extractor scheduler, which automate data extraction processes from designated APIs at user-defined intervals.

Software Application Component

This component comprises a configurable data visualization platform and a database management system. The configurable data visualization platform offers a customizable interface for data monitoring and analysis, while the database management system automatically generates databases for incoming data, ensuring efficient data storage and retrieval.

Academic Review of Perceived Challenges

Table 1: Table of Studied Literature Regarding Challenges

Name	Title	Challenge Discussed
T. C. & J.-B. S.	Component-based architecture: the Fractal initiative	General challenges implementing component-based architecture
P. C. Clements	Software Architecture In Practice	Real-world software architecture projects, including component-based systems
M. R. P. S. Z. D.	Software Architecture: The Hard Parts	Architectural decomposition and design of component-based systems
M. L. F. L. & M. M.	From Component-Based Architectures to Microservices	Evolution of component-based architectures. Transitioning to microservices
C. S. L. K. S. T. S. M.	An Interoperable Component-Based Architecture for Data-Driven IoT System	Interoperability and integration in data-driven IoT systems

Potential Use Cases

There are several instances where the solutions mentioned above have been used. Various components, such as headers, footers, side panels, and content bodies constitute the application's architecture. Each element functions

as a self-contained unit with its own interface, creating a seamless communication pathway among components via designated ports. The utilization of component teams, characterized by their cross-functional Agile nature, proves instrumental in addressing intricate aspects of the application, including legacy technology, algorithms, and compliance requirements [6].

To effectively manage the application's complexity, techniques such as Domain-Driven Design (DDD) and tools like Storybook and Atomic Design are employed.

The adoption of the Ontology Web Language (OWL) serves to bridge the semantic gap by establishing a common language for articulating product requirements and component capabilities. This standardized approach to ontology representation promotes clarity and consistency in software engineering projects, thereby promoting effective component reuse [7].

During the development phase, requirements are specified using modeling tools to generate Ontology Definition Language (ODL) ontologies. Components are subsequently tagged with metadata in ODL, elucidating their semantics and facilitating effective component modeling and integration.

The top-down decomposition system encompasses various functional components, including requirements checking, component mapping, structure synthesis, constraints checking, and component configuration.

To ensure seamless interoperability between components, clear and consistent conventions for component design and communication are established. This involves adhering to standardized naming conventions, delineating well-structured component APIs, and fostering a shared understanding of state management strategies.

A simple example of a React component that could be used in this context contains a component which is self-contained and communicates through props (which can be thought of as ports). It's also decomposable and highly customizable [7].

```
import React from 'react';
// Define a simple Button component
const Button = ({ onClick, children }) => {
 return (
 <button onClick={onClick}>
 {children}
 </button>
 );
};

// Use the Button component
const App = () => {
 const handleClick = () => {
 alert('Button clicked!');
 };

 return (
 <div>
 <h1>Welcome to our app!</h1>
 <Button onClick={handleClick}>
 Click me
 </Button>
 </div>
 );
};

export default App;
```

In this example, the “Button” component is self-contained and has an interface that communicates through props. Furthermore, the “onClick” prop is a function that gets called when the button is clicked, and the “children” prop defines what’s displayed inside the button.

Going through further, the “App” component uses the “Button” component and passes in the necessary props. The “handleClick” function is defined in the “App” component and passed to the “Button” component via the “onClick” prop.

This code example is very basic, though, showcasing the solution discussed in this paper. The idea is for the components to be self-contained using a top-down structure. This also allows the components to communicate through interfaces (props in this case), and the system (the "App" component) is decomposable into reusable, cohesive, and independent components ("Button" in this case).

6. Conclusion

This paper for component-based architecture (CBA) has considered both the promises and challenges that CBA presents in modern software development, particularly front-end development.

References

[1]. T. C. &. J.-B. S. Gordon Blair, "Component-based architecture: the Fractal initiative," Annals of Telecommunications - Annales des Télécommunications , vol. 64, no. 1, pp. 1-4, 2009.

[2]. P. C. Clements, Software Architecture in Practice, Pittsburgh, PA: Carnegie Mellon University, 2002.

[3]. J. Bradbury, "A survey of self-management in dynamic software architecture specifications," in Proc. of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, Newport Beach, California, USA, 2004.

[4]. M. R. P. S. Z. D. Neal Ford, "Chapter 4. Architectural Decomposition," in Software Architecture: The Hard Parts, O'Reilly Media, Inc., October 2021, p. Chapter 4.

[5]. M. L. F. L. &. M. M. Giuseppe De Giacomo, "From Component-Based Architectures to Microservices: A 25-years-long Journey in Designing and Realizing Service-Based Systems," Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future, vol. 12521, no. 1, pp. 3-15, 10 April 2021.

[6]. C. S. L. K. S. T. S. M. Sin Kit Lo, "An Interoperable Component-Based Architecture for Data-Driven IoT System," Sensors 2019, 19 10 2019.

[7]. Hiren Dhaduk, “Component Based Development: The Definitive Guide to Making a Scalable Frontend” Simform 18 8 2021

[8]. AppMaster “React's Component-Based Architecture: A Case Study” AppMaster