Scientific Publications

Journal of Artificial Intelligence and Big Data, 2024, 4, 911
www.scipublications.org/journal/index.php/jaibd
DOI: 10.31586/jaibd.2024.911

Article

5V's of Big Data Shifted to Suite the Context of Software Code: Big Code for Big Software Projects

Linda Susan Amos, Eng. Tirivangani Magadza

School of Information Science & Technology (SIST)-Harare Institute of Technology, Zimbabwe

*Correspondence: Linda Susan Amos (lamos@hit.ac.zw)

Abstract: Data is the collection of facts and observations in terms of events, it is continuously growing, getting denser and more varied by the minute across different disciplines or fields. Hence, Big Data emerged and is evolving rapidly, the various types of data being processed are huge, but no one has ever thought of where this data resides, we therefore noticed this data resides in software's and the codebases of the software's are increasingly growing that is the size of the modules, functionalities, the size of the classes etc. Since data is growing so rapidly it also mean the codebases of software's or code are also growing as well. Therefore, this paper seeks to discuss the 5V's of big data in the context of software code and how to optimize or manage the big code. When we talk of "Big Code for Big Software's," we are referring to the specific challenges and considerations involved in developing, managing, and maintaining of code in large-scale software systems.

Keywords: Big Code, Big Software, Big Data, 5V's, Code Optimization, Scalability

How to cite this paper:
Amos, L. S., & Magadza, E. T. (2024). 5V's of Big Data Shifted to Suite the Context of Software Code: Big Code for Big Software Projects. *Journal of Artificial Intelligence and Big Data,* 4(1), 36–47. Retrieved from https://www.scipublications.com/journal/index.php/jaibd/article/view/911

Received: March 5, 2024
Revised: April 6, 2024
Accepted: April 9, 2024
Published: April 11, 2024

1. Introduction

"**Big Code**" typically refers to large-scale software systems that involves a significant amount of code, often consisting of thousands or even millions of lines of code. These codebases are typically complex, with multiple modules, dependencies, and interactions between different components. **"Big Software**" refers to software applications or systems that are extensive and complex in nature. This could include enterprise-level software solutions, operating systems, large-scale web applications, databases, or any software that requires substantial resources and infrastructure to develop, deploy, and maintain. When we talk about "Big Code for Big Software's," we are referring to the specific challenges and considerations involved in developing, managing, and maintaining large-scale software systems [1]. Some of the key aspects that developers and teams need to address when working with big code and big software include:

Scalability: Designing the software architecture to handle large amounts of data, high user loads, and growing demands over time [2]. **Modularity**: Breaking down the codebase into manageable modules or components to improve maintainability, reusability, and collaboration among developers. **Performance**: Optimizing the software to ensure efficient execution and response times, considering factors like algorithmic complexity, data structures, and system resources. **Testing and Quality Assurance**: Implementing robust testing strategies, including unit testing, integration testing, and performance testing, to ensure the reliability and stability of the software. **Documentation**: Providing comprehensive and up-to-date documentation to aid in understanding and maintaining the codebase, as well as facilitating collaboration between team members. **Version Control and Collaboration**: Utilizing version control systems, such as Git, to manage the codebase, track changes, and enable collaboration among multiple developers or teams. **Deployment and Infrastructure**: Considering the deployment strategies,

infrastructure requirements, and scalability options to ensure smooth deployment and operation of the software in production environments. These are just a few considerations when dealing with large-scale software systems. The specific challenges and approaches may vary depending on the nature of the software and the technologies involved [3].

Table 1. 1: Research Questions.

Number	Research Question
RQ1	How can we term the 5'Vs of big data in the context of big code for big Software's
RQ2	What are some common strategies for optimizing performance in big code and big software?
RQ3	How many libraries and modules so far, we have for all the languages in programming?
RQ4	How can developers deal with memorization of code? Is there any need to keep code in memory
RQ5	How can developers effectively collaborate and share knowledge when working on big software projects?
RQ6	How helpful is DevOps when dealing with big code?
RQ7	How can developers ensure the veracity of the code in big software projects?
RQ8	What is the future behind the increase in size of code & it's complexity & what can be done?
RQ9	How OOP Pattern be utilized in order to assist in reducing the complexity & increase in code size
RQ10	How can the SOLID principles be applied to manage code size effectively?
RQ11	How will future design patterns can be implemented besides OOP?

2. Methodology

The process involved in conducting the research is presented in this section. The research identifies, assesses and interprets all the papers applicable to the application of big data analytics in different domains. The method that was used in the process of reviewing the papers for literature includes planning which involves the searching process by using keywords like big data and 5 Vs of big data as well as the formulation of the research questions. Conducting was the next process by removing the extraneous studies that is the inclusion and exclusion criteria. Lastly it was the reporting phase in which results and findings of the research was shared.

3. Reporting of the Results

RQ1-How can we term the 5'Vs of big data in the context of big code for big Software's?

The five Vs of big data Volume, Velocity, Variety, Veracity, and Value can be adapted and applied to the context of big code for big software projects. The following is how we can term them:

Volume

In the context of big code, Volume refers to the sheer size and scale of the codebase. Big software projects often have extensive codebases with a large number of files, classes, functions, and lines of code. The Volume of the code can present challenges in terms of understanding, maintenance, and scalability.

Velocity

Velocity in the context of big code for big software refers to the speed at which code changes and updates occur. Big software projects typically involve multiple developers working simultaneously on different features or components. The Velocity of code changes can be high, requiring efficient collaboration, version control, and continuous integration practices.

Variety

Variety refers to the different types of code and technologies present in a big software project. Large-scale projects often involve a variety of programming languages, frameworks, libraries, and modules. The Variety of code requires developers to have a diverse skill set and the ability to work with different technologies seamlessly.

Veracity

Veracity in the context of big code relates to the accuracy, reliability, and quality of the codebase. Big software projects demand adherence to coding standards, best practices, and code quality guidelines. Ensuring Veracity involves code reviews, testing, and continuous quality assurance processes to maintain a high level of code integrity.

Value

Value refers to the ultimate benefit or worth that the big software project delivers. A big codebase should contribute to the value proposition of the software by enabling the desired functionalities, performance, and user experience. Value is realized when the software meets the needs of its users, achieves business objectives, and provides a competitive advantage. Considering these five Vs which are Volume, Velocity, Variety, Veracity, and Value. Developers and project teams can better understand and address the challenges associated with big code in big software projects. This understanding can help guide decisions related to code management, collaboration, quality control, and overall project success.

RQ2-What are some common strategies for optimizing performance in big code and big software?

Optimizing performance in big code and big software involves various strategies to enhance the efficiency and responsiveness of the software:

Algorithmic Optimization: Analyze the algorithms used in the software and identify opportunities for improvement. This may involve selecting more efficient algorithms, reducing redundant computations, or optimizing time or space complexity. **Data Structures**: Carefully select appropriate data structures based on the specific requirements of the software. Efficient data structures can significantly impact performance by reducing the time complexity of operations such as searching, insertion, and deletion. Caching: Implement caching mechanisms to store frequently accessed or computed data. Caching can help reduce expensive computations or database queries, improving response times and overall performance [4]. **Parallel Computing**: Utilize parallel processing techniques, such as multithreading or distributed computing, to execute tasks concurrently and take advantage of modern hardware capabilities. This can enable better utilization of resources and improved performance. **Database Optimization**: Optimize database queries, indexes, and schema design to minimize data retrieval and manipulation overhead. Techniques like query optimization, denormalization, and database indexing can significantly enhance database performance. **Code Profiling and Performance Monitoring**: Use profiling tools to identify performance bottlenecks in the codebase. Profiling can help pinpoint specific functions, methods, or code segments that consume excessive resources or exhibit poor performance. Monitoring tools can help track and analyze system performance in production environments. **I/O Optimization**: Minimize disk I/O and network operations by optimizing file handling, database access, and network communication. This can involve techniques like batch processing, asynchronous operations, or data compression to reduce latency and improve throughput. **Resource Management**: Efficiently manage system resources, such as memory, CPU, and network bandwidth. Avoid memory leaks, unnecessary resource allocations, and excessive context switching to optimize resource utilization and prevent performance degradation. **Code**

Refactoring: Refactor complex or poorly performing code segments to improve readability, maintainability, and performance. Simplifying code logic, reducing code duplication, and eliminating unnecessary computations can lead to better performance. **Load Testing and Performance Tuning**: Conduct load testing to simulate real-world usage scenarios and identify performance limitations. Based on the results, fine-tune the software by adjusting configurations, optimizing critical paths, and addressing identified performance issues. It's worth noting that the specific strategies employed for performance optimization may vary depending on the programming language, frameworks, and technologies used in a particular software project. It's essential to profile and analyze the application's performance to identify the most effective optimization strategies for a given scenario [5].

RQ3-How many libraries and modules so far, we have for all the languages in programming?

It is challenging to provide an exact number of libraries and modules available for all programming languages, as the landscape is constantly evolving with new libraries being created and existing ones being updated. Additionally, the availability and number of libraries can vary across different programming languages. However, below is a general overview of the popularity and abundance of libraries/modules in some widely used programming languages:Python has a rich ecosystem of libraries and modules due to its popularity and extensive community support. The Python Package Index (PyPI) is the official repository for Python packages, and it hosts over 300,000 packages as of my knowledge cutoff in September 2021. Popular libraries include NumPy, Pandas, TensorFlow, Django, Flask, and many more.JavaScript also has a vast number of libraries and modules available, primarily due to its widespread use in web development. The npm (Node Package Manager) registry is the main repository for JavaScript packages, with over 1.5 million packages as of my knowledge cutoff. Popular libraries include React, Angular, Vue.js, Express.js, Lodash, and many others.Java has a large ecosystem of libraries and frameworks. Maven Central and the Java Development Kit (JDK) are common sources for Java libraries. Popular libraries and frameworks include Spring Framework, Hibernate, Apache Commons, JUnit, Log4j, and many more.The .NET ecosystem provides a wide range of libraries and frameworks for C#. The NuGet package manager is the primary source for C# packages, with thousands of libraries available. Popular libraries and frameworks include Entity Framework, ASP.NET Core, NUnit, Newtonsoft.Json, and more.Ruby's package manager, RubyGems, hosts a significant number of libraries and frameworks. Popular libraries include Rails, RSpec, Devise, ActiveRecord, and many others.These are just a few examples, and there are numerous other programming languages with their own ecosystems of libraries and modules. The availability and number of libraries evolve rapidly, with new ones being created and existing ones being updated regularly to cater to the needs of developers and evolving technologies.

RQ4-How can developers deal with memorization of code? Is there any need to keep code in memory?

In most cases, developers do not need to memorize every line of code in a big software project. Memorizing an entire codebase, especially in large-scale projects, would be impractical and inefficient. Instead, developers should focus on understanding the underlying concepts, design patterns, and architectural principles employed in the software. It's important to have a good grasp of the overall structure and flow of the codebase, as well as a solid understanding of the key components and their interactions. Developers should be familiar with the modules or components they work on regularly, and they should have a general understanding of the purpose and functionality of other parts of the codebase. This knowledge helps them navigate the codebase effectively,

identify relevant code sections, and understand how changes in one area may impact other parts of the system. In practice, developers rely on various tools and techniques to assist them in working with large codebases, such as:

Well-documented codebases provide valuable insights into the purpose, usage, and dependencies of different code components. Developers can refer to documentation to understand how different parts of the codebase fit together. Inline comments within the code can provide additional context and explanations for complex or critical sections. These comments help developers understand the code's functionality and intent. Integrated development environments (IDEs) often provide powerful search functionalities that allow developers to locate specific functions, classes, or variables within the codebase quickly. IDEs also offer navigation features like code folding, outlines, and jump-to-definition, which facilitate code exploration. Developers use version control systems like Git to track changes made to the codebase over time. With proper commit messages and branching strategies, developers can review the history of the codebase and understand the rationale behind specific changes. Communication and collaboration among developers are crucial when working on big software projects. Teams can share knowledge, discuss design decisions, and document important insights to collectively understand and maintain the codebase. Though it's not necessary to memorize every line of code, developers should strive for a good understanding of the codebase's structure, key functionalities, and design principles. This knowledge, coupled with effective tools and collaboration, enables developers to navigate and work with big codebases efficiently.

RQ5-How can developers effectively collaborate and share knowledge when working on big software projects?

Effective collaboration and knowledge sharing are crucial when working on big software projects. The strategies that developers can employ to facilitate collaboration and knowledge sharing:

Establish open and effective communication channels within the development team. This can include regular team meetings, stand-ups, and dedicated communication platforms such as Slack or Microsoft Teams. Encourage team members to ask questions, share insights, and provide updates on their work. Encourage developers to document important aspects of the project, such as architecture, design decisions, coding conventions, and guidelines. This documentation serves as a reference for team members and helps new developers onboard more easily. Consider using tools like wikis or shared documents to centralize and organize project documentation. Implement a code review process where team members review each other's code changes. Code reviews provide an opportunity for knowledge sharing, identify potential issues, and ensure code quality. Encourage constructive feedback, discussions, and knowledge transfer during code reviews. Consider using pair programming techniques, where two developers work together on the same piece of code. This collaborative approach promotes real-time knowledge sharing, problem-solving, and immediate feedback. It can be particularly beneficial for onboarding new team members or tackling complex tasks.

Knowledge Sharing Sessions

Organize regular knowledge sharing sessions, where team members can present and discuss various topics related to the project or relevant technologies. These sessions can include demos, presentations, or workshops on specific aspects of the codebase, tools, or best practices.

Shared Code Repositories

Utilize version control systems like Git to maintain a shared code repository. This allows team members to collaborate, track changes, and review each other's code.

Encourage the use of branching and pull request workflows to facilitate collaboration and provide a platform for discussions and feedback.

Collaborative Tools and Platforms

Leverage collaboration tools and platforms that enable real-time collaboration, such as online whiteboards, project management software, and shared code editors. These tools can enhance remote collaboration and facilitate synchronous or asynchronous discussions.

Continuous Integration and Deployment

Implement continuous integration and deployment (CI/CD) pipelines to automate the build, testing, and deployment processes. CI/CD pipelines encourage collaboration by providing a shared and consistent development environment. They also enable frequent integration of code changes, reducing conflicts and ensuring the project maintains a stable state.

Team Knowledge Base

Create a team knowledge base or wiki where developers can contribute and share their insights, lessons learned, and best practices. This centralized repository of information can help address frequently encountered issues, document workarounds, and serve as a valuable resource for the team.

Regular Retrospectives

Conduct regular retrospectives to reflect on the team's collaboration and knowledge sharing practices. Identify areas for improvement, discuss challenges, and implement actions to enhance collaboration and knowledge sharing within the team. By implementing these strategies, developers can foster a collaborative and knowledge-sharing culture, ensuring that the collective expertise of the team is leveraged effectively to tackle the challenges of working on big software projects.

RQ6-How helpful is DevOps when dealing with big code?

DevOps practices can be highly beneficial when dealing with big code and big software projects. DevOps, which combines development (Dev) and operations (Ops), focuses on streamlining the software development lifecycle, improving collaboration, and ensuring the efficient delivery of software.

Ways in which DevOps can help when dealing with big code:

Continuous Integration and Deployment: DevOps encourages the use of CI/CD pipelines, which automate the process of integrating code changes, running tests, and deploying software. This allows for frequent integration of code changes, reducing conflicts and enabling faster feedback loops. With big code projects, where multiple developers are working simultaneously, continuous integration helps identify integration issues early and ensures that the software remains in a stable state. Infrastructure as Code: Infrastructure as Code (IaC) is an essential DevOps practice that involves managing infrastructure elements (such as servers, networks, and databases) using code. By treating infrastructure as code, DevOps enables version control, automation, and repeatability. This is particularly helpful in big code projects where complex infrastructure setups are required. IaC helps maintain consistency, reduces manual errors, and ensures that the infrastructure can scale efficiently. Collaboration and Communication: DevOps emphasizes collaboration and effective communication between development teams and operations teams. This is crucial in big code projects where multiple teams and stakeholders are involved. By fostering better collaboration, DevOps practices help bridge the gap between developers, operations personnel, and other stakeholders, ensuring a shared understanding of goals, requirements, and challenges. Monitoring and

Performance Optimization: DevOps promotes the use of monitoring and performance optimization techniques. By implementing monitoring tools and collecting relevant metrics, teams can gain insights into the performance of the software in production. This is especially important in big code projects, as it allows developers to identify performance bottlenecks, optimize code, and fine-tune the system based on real-time data. Automation and Configuration Management: DevOps encourages the automation of repetitive tasks and the use of configuration management tools. Automation helps reduce manual effort, increases efficiency, and minimizes the chances of human error when working with big codebases. Configuration management tools enable consistent and repeatable deployments across different environments, ensuring that the software works as expected in various settings. Scalability and Resilience: Big code projects often require systems that can handle high loads and scale effectively. DevOps practices, such as infrastructure scaling, load testing, and fault tolerance strategies, help ensure that the software can scale horizontally or vertically as needed. DevOps also focuses on building resilient systems that can recover quickly from failures, minimizing downtime and providing a better user experience. Overall, the devOps practices provide a framework and set of tools that can enhance the development, deployment, and maintenance of big code projects. By improving collaboration, automation, scalability, and monitoring, DevOps enables teams to tackle the challenges associated with large-scale software development and deliver high-quality software efficiently.

RQ7-How can developers ensure the veracity of the code in big software projects?

Ensuring the veracity of code in big software projects is crucial for maintaining code quality, reliability, and overall project success. The following are some practices that developers can follow to ensure the veracity of the code:

Implement a robust code review process where developers review each other's code changes. Code reviews help identify potential issues, ensure adherence to coding standards, and promote knowledge sharing. Encourage thorough and constructive feedback during code reviews to ensure the veracity of the code. Implement a comprehensive suite of automated tests, including unit tests, integration tests, and end-to-end tests. Automated testing helps validate the functionality and behavior of the code, ensuring that it performs as expected. Develop test cases that cover various scenarios and edge cases to ensure the veracity of the code across different situations. Utilize static code analysis tools that can automatically analyze the codebase for potential issues, code smells, and adherence to coding standards. These tools can help identify potential bugs, security vulnerabilities, and maintainability issues. Enforce the use of static code analysis as part of the development process to improve the veracity of the code. Maintain accurate and up-to-date documentation for the codebase. This includes documenting the purpose, usage, and interfaces of key components and modules. Documentation helps developers understand the code and its intended behavior, improving the veracity of the code and facilitating future maintenance. Establish and enforce coding standards and best practices within the development team. Consistent coding styles, naming conventions, and guidelines improve code readability and maintainability. By following established coding standards, developers can ensure that the codebase is consistent, reliable, and easier to understand. Utilize version control systems like Git and establish effective branching strategies. Version control systems help track changes made to the codebase over time and enable collaboration. By using proper branching strategies, developers can isolate new features or bug fixes and ensure that changes are thoroughly reviewed and tested before merging into the main codebase. Implement continuous integration and deployment (CI/CD) pipelines that automate the process of integrating code changes, running tests, and deploying the software. CI/CD pipelines help catch integration issues early and ensure the veracity of the codebase by providing a reliable and repeatable deployment process. Code Documentation and Comments: Include inline comments and

documentation within the codebase to explain complex or critical sections, algorithms, or design decisions. Well-documented code enhances the veracity of the code by providing insights into the code's intent and implementation details. Encourage peer discussions, knowledge sharing sessions, and technical discussions within the development team. Regular team meetings or knowledge-sharing sessions allow developers to share insights, discuss challenges, and collectively ensure the veracity of the codebase. Foster a culture of continuous learning and improvement within the development team. Encourage developers to stay updated with the latest programming techniques, tools, and best practices. By continuously improving their skills, developers can contribute to the veracity of the codebase and enhance its overall quality. By following these practices, developers can significantly improve the veracity of the code in big software projects. Regular code reviews, automated testing, documentation, adherence to coding standards, and continuous improvement processes ensure that the codebase remains reliable, maintainable, and aligned with the project's goals.

RQ8-What is the future behind the increase in size of code & it's complexity & what can be done?

The increase in the size and complexity of code is a natural consequence of the evolving software landscape and the growing demands for more sophisticated applications. Several factors contribute to this trend:

Advancements in Technology: As technology advances, new frameworks, libraries, and tools are introduced, allowing developers to build more complex and feature-rich applications. These advancements often come with additional layers of abstraction and complexity, resulting in larger codebases. **Growing Functional Requirements**: Software applications are continuously evolving to meet the increasing demands of users and businesses. New features, integrations, and functionalities are added over time, leading to an expansion of the codebase. As applications become more comprehensive and versatile, the size and complexity of the code naturally increase. **Scalability and Performance Considerations**: Applications are designed to handle larger user bases, process larger datasets, and scale across multiple devices and platforms. To achieve scalability and performance, developers often introduce more complex algorithms, distributed systems, and optimization techniques, resulting in larger and more intricate codebases. **Code Reusability and Modularity**: Developers strive to build reusable code components and modular architectures, which allow for easier maintenance, extensibility, and collaboration. While code reusability can lead to more efficient development practices, it may also result in larger codebases as reusable components are shared across multiple projects.

To address the challenges associated with the increase in code size and complexity, developers can adopt several strategies:

Effective Code Organization

Employ modular design principles to break down the codebase into smaller, manageable components. This improves code readability, maintainability, and reusability. Encourage the use of design patterns and software architecture principles to ensure a well-structured and scalable codebase.

Documentation and Knowledge Sharing

Comprehensive documentation, including code comments, API references, and architectural overviews, is crucial for understanding complex codebases. Encourage developers to document their code and share knowledge within the team to facilitate collaboration and reduce the learning curve for new team members.

Code Refactoring

Regularly review and refactor the codebase to eliminate redundancies, improve code quality, and reduce complexity. Refactoring involves restructuring the code without changing its external behavior, making it easier to understand, maintain, and enhance over time.

Test-Driven Development (TDD)

Adopt TDD practices to ensure that the codebase remains reliable and behaves as expected. Writing automated tests before implementing new features or making changes helps catch issues early and provides a safety net for refactoring and code optimization.

DevOps and Continuous Integration

Implement DevOps practices to streamline the software development process, automate builds, testing, and deployment. Continuous integration ensures that code changes are frequently merged and tested, reducing integration issues and improving code veracity.

Code Reviews and Pair Programming

Encourage peer code reviews and pair programming sessions to promote collaboration, knowledge sharing, and early identification of potential issues. Effective code reviews can help maintain code quality, consistency, and reduce code complexity.

Monitoring and Performance Analysis

Implement comprehensive monitoring and performance analysis tools to detect bottlenecks, identify areas of code inefficiency, and optimize critical sections. This helps ensure that the codebase performs well and scales appropriately.

Continuous Learning and Skill Development

Encourage developers to stay updated with the latest technologies, programming languages, and best practices. Investing in continuous learning and skill development ensures that developers are equipped to handle the complexities of large codebases effectively. By adopting these strategies, developers can mitigate the challenges associated with the increase in code size and complexity. Well-structured, modular, and maintainable codebases can facilitate efficient development, easier collaboration, and long-term project success.

RQ9-How OOP Pattern be utilized in order to assist in reducing the complexity & increase in code size?

Object-Oriented Programming (OOP) patterns can indeed be utilized to assist in reducing complexity and managing code size. OOP patterns provide design solutions for common programming challenges and promote code organization, reusability, and maintainability. Here are some ways OOP patterns can help: **Encapsulation**: Encapsulation is a fundamental principle in OOP that involves bundling data and related behavior into objects. By encapsulating code within objects, you can reduce complexity by hiding implementation details and providing a clear interface for interacting with the object. This helps in managing code size by keeping related functionality together and promoting modular design. **Abstraction**: Abstraction allows you to focus on essential characteristics and behavior while hiding unnecessary details. By abstracting concepts into classes and interfaces, you can simplify code and reduce its size. Abstraction also enables code reusability, as common functionality can be defined in abstract classes or interfaces, and specific implementations can be provided by subclasses. **Inheritance**: Inheritance is a mechanism in OOP that allows classes to inherit properties and behavior from other classes. It promotes code reuse by enabling the creation of specialized classes that inherit and extend the functionality of base classes. Inheritance can help reduce code

duplication and improve code size management by structuring classes hierarchically. **Polymorphism**: Polymorphism allows objects of different types to be treated as instances of a common superclass or interface. This enables code to be written more generically, reducing the need for repetitive code and minimizing code size. Polymorphism also supports flexibility and extensibility by allowing new classes to be added without modifying existing code. **Design Patterns**: Design patterns are reusable solutions to commonly occurring design problems. They provide proven approaches for structuring and organizing code. By utilizing design patterns such as the Factory, Singleton, Observer, or Strategy patterns, you can reduce code complexity and promote code modularity, scalability, and maintainability. **Composition over Inheritance**: The composition principle suggests favoring object composition over class inheritance. Instead of relying heavily on inheritance, you can compose objects by combining smaller, more specialized classes to achieve desired functionality. This approach promotes code reuse, reduces code size, and provides flexibility in combining and modifying behavior. **SOLID Principles**: SOLID is a set of principles that guide software design and promote maintainable and scalable code. By adhering to principles like Single Responsibility, Open-Closed, and Dependency Inversion, you can reduce code complexity, improve code modularity, and facilitate code size management. When using OOP patterns, it's essential to strike a balance between code organization and avoiding unnecessary complexity. Applying patterns judiciously based on the specific needs of the software project can help reduce code size, improve maintainability, and manage complexity effectively.

RQ10-How can the SOLID principles be applied to manage code size effectively?

The SOLID principles provide guidelines for designing software that is modular, maintainable, and scalable. While their primary focus is not specifically on managing code size, adhering to the SOLID principles can indirectly help in managing code size effectively. Here's how each SOLID principle can contribute to code size management:

Single Responsibility Principle (SRP)

The SRP states that a class should have only one reason to change. By adhering to this principle, you ensure that each class has a clear and focused responsibility, reducing its size and complexity. When a class is responsible for a single task, it tends to be smaller, easier to understand, and less likely to grow in size over time.

Open-Closed Principle (OCP)

The OCP states that software entities (classes, modules, etc.) should be open for extension but closed for modification. By designing code that is open for extension, you can add new features or behavior without modifying existing code. This helps manage code size by minimizing the need to modify and potentially increase the size of existing code when extending functionality.

Liskov Substitution Principle (LSP)

The LSP states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. By adhering to this principle, you ensure that subclasses do not introduce new dependencies or behaviors that increase code size unnecessarily. LSP promotes code reuse and helps avoid the need for duplicative code.

Interface Segregation Principle (ISP)

The ISP states that clients should not be forced to depend on interfaces they do not use. By adhering to this principle, you avoid bloated interfaces that contain unnecessary methods, reducing code size. Breaking down interfaces into smaller, more focused ones

allows clients to depend only on the specific functionality they require, resulting in more concise and manageable code.

Dependency Inversion Principle (DIP)

The DIP states that high-level modules should not depend on low-level modules; both should depend on abstractions. By applying DIP, you introduce abstraction layers that decouple modules and reduce direct dependencies. This promotes code modularity and allows for the replacement or modification of low-level modules without affecting the size or structure of high-level modules. Though the SOLID principles themselves do not directly address code size management, they promote good design practices that inherently support code modularity, maintainability, and scalability. By adhering to SOLID, you can create smaller, focused classes with clear responsibilities, avoid code duplication through inheritance and abstraction, and reduce unnecessary dependencies. These practices contribute to managing code size effectively in the long term.

RQ11-How will future design patterns can be implemented besides OOP?

Besides Object-Oriented Programming (OOP) design patterns, there are other design patterns and architectural patterns that can be implemented to address specific challenges in software development. Here are a few notable ones:

Functional Programming Design Patterns: Functional programming design patterns focus on immutability, pure functions, and composition. Patterns like map/reduce, monads, and immutability can be employed to create functional and declarative code that is concise, maintainable, and easier to reason about. **Reactive Programming Design Patterns**: Reactive programming patterns, such as Observer, Publisher-Subscriber, and Reactive Streams, are designed to handle asynchronous, event-driven systems. These patterns enable efficient handling of streams of data and events, providing responsiveness and scalability in modern applications. **Microservices Architecture**: Microservices architecture is an architectural pattern where an application is decomposed into smaller, loosely coupled services. Each service focuses on a specific business capability and can be developed, deployed, and scaled independently. Microservices promote modularity, flexibility, and scalability, making them suitable for complex and distributed systems. **Event-Driven Architecture (EDA)**: Event-Driven Architecture is an architectural pattern that revolves around the production, detection, consumption, and reaction to events. Events represent significant occurrences within a system, and they are used to trigger actions or communicate between components. EDA promotes loose coupling, scalability, and responsiveness, making it suitable for event-driven systems. **Domain-Driven Design (DDD)**: Domain-Driven Design is an approach that focuses on modeling a software system based on the domain it represents. DDD emphasizes collaboration between domain experts and developers to create a rich and expressive domain model. It introduces patterns like Aggregates, Entities, Value Objects, and Ubiquitous Language to create maintainable, business-focused designs. **Data-Driven Design**: Data-Driven Design patterns focus on organizing and processing data efficiently. Patterns like Data Access Object (DAO), Repository, and Data Mapper help decouple data access and manipulation from the rest of the application, promoting reusability and maintainability. **Serverless Architecture**: Serverless architecture is a cloud computing model where the cloud provider manages the infrastructure and automatically scales resources based on demand. Serverless patterns, such as Function-as-a-Service (FaaS) and Event-Driven Architecture, enable developers to focus on writing functions or services without worrying about infrastructure management. Serverless architectures provide cost-efficiency, scalability, and rapid development. These are just a few examples of design patterns and architectural patterns beyond OOP that can be implemented to address specific software development challenges. The choice of pattern depends on the specific requirements, system architecture, and the problem being addressed. It's

important to consider the trade-offs and suitability of each pattern based on the context of your project.

4. Discussion

Design patterns, whether based on Object-Oriented Programming (OOP) or other paradigms, provide valuable solutions to recurring software design challenges. They offer a set of proven approaches that promote code organization, reusability, maintainability, and scalability.

OOP patterns, such as encapsulation, abstraction, inheritance, and polymorphism, help reduce complexity and manage code size effectively. By encapsulating code within objects, abstracting concepts, utilizing inheritance hierarchies, and leveraging polymorphism, developers can create modular, understandable, and extensible code. These patterns promote code reuse, reduce duplication, and facilitate maintainability.

The SOLID principles, which are guiding principles for software design, also indirectly contribute to code size management. By adhering to the Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle, developers create code that is focused, extendable, reusable, and loosely coupled. These principles lead to smaller, more manageable code units and promote modular design.

Beyond OOP, there are other design patterns and architectural patterns that address specific challenges in software development. Functional programming patterns, reactive programming patterns, microservices architecture, event-driven architecture, domain-driven design, data-driven design, and serverless architecture are just a few examples. These patterns introduce new ways of structuring and organizing code, handling concurrency and asynchrony, managing data, and designing distributed systems.

5.Conclusion

In conclusion, the effective use of design patterns is crucial in managing code complexity and size. OOP patterns, such as encapsulation, abstraction, inheritance, and polymorphism, along with the SOLID principles, provide a solid foundation for creating maintainable and scalable code. Additionally, exploring other design patterns and architectural patterns beyond OOP can further enhance code organization, reusability, and modularity. The choice of patterns depends on the specific requirements, context, and problem domain of the software project. By applying appropriate patterns, developers can achieve code that is easier to understand, maintain, and extend, ultimately contributing to efficient code size management.

References

[1] N. &. H. L. Saeed, "Big data characteristics (V's) in industry. Iraqi Journal of Industrial Research," vol. 8(1), pp. 1-9., 2021.

[2] B. S. &. J. S. Jyothi, "A study on big data modelling techniques," International Journal of Computer Networking, Wireless and Mobile Communications (IJCNWMC), vol. 5(6), pp. 19-26., 2015.

[3] F. B. L. K. M. &. S. C. (Bachmann, "Designing software architectures to achieve quality attribute requirements," IEE Proceedings-Software, vol. 152(4), pp. 153-165, 2005.

[4] V. D. &. A. P. Kumar, "Software engineering for big data projects: Domains, methodologies and gaps. .," IEEE International Conference on Big Data (Big Data), pp. pp. 2886-2895, 2016, December.

[5] T. Arndt, Big Data and software engineering: prospects for mutual enrichment. Iran journal of computer science, vol. 1(1), pp. 3-10, 2018.