Application of design patterns in the development of the architecture of monitoring systems

Aleksandra Pasynkova
Faculty of Computer Science, Economics, and Social Sciences
HSE University
38 Studencheskaya str., Perm, 614070 Russian Federation
aapasynkova1@yandex.ru

Olga Vikentyeva
Faculty of Computer Science, Economics, and Social Sciences
HSE University
38 Studencheskaya str., Perm, 614070 Russian Federation
ovikenteva@hse.ru

***Abstract*—This article explores the relevance of using design patterns in the development of the architecture of monitoring systems. The increasing complexity of modern monitoring systems has made it challenging to maintain and evolve them. The use of design patterns can address these challenges by providing reusable solutions to common problems in monitoring system architecture. This article reviews the literature on monitoring systems and design patterns and identifies appropriate design patterns for monitoring system architecture. The article also analysis the requirements for monitoring systems and demonstrates how design patterns can be used to meet these requirements. The results show that the use of design patterns can improve the maintainability, flexibility, reliability, compatibility and scalability of monitoring systems. This article is relevant to software architects, developers, and system administrators who are involved in the development and maintenance of monitoring systems.**

***Keywords*—design patterns, monitoring systems, architecture, monitoring system requirements.**

I. INTRODUCTION

Monitoring systems have become an essential part of various industries, providing real-time information about the health and performance of critical systems. These systems are complex and require sophisticated architectures to handle the data flow, processing, and storage [1]. However, as the systems grow and evolve, they become increasingly challenging to maintain, and changes can have unforeseen consequences [2]. This is where the use of design patterns can be invaluable.

Design patterns are reusable solutions to commonly occurring problems in software design. By applying design patterns, developers can address specific design issues and improve the quality of the system [3-5]. Design patterns have proven to be effective tools in software development, providing solutions to common problems and ensuring that software systems are scalable, maintainable, and flexible [6-8].

The problem is that without using design patterns, the maintenance of monitoring systems can be difficult, time-consuming and prone to errors [9-11]. As the system grows, the complexity increases, and it becomes harder to make changes without causing unintended consequences. Therefore, it is essential to assess the possibility of using design patterns in the development of monitoring system architecture.

Also, the relevance of developing own architecture independently, without using ready-made open-source solutions is justified by the fact that some enterprises cannot do this because of high secrecy and the need to ensure security when working with a monitoring system. Therefore, the use of foreign solutions cannot be chosen.

This article will analyse the possibility of using design patterns to develop the architecture of monitoring systems and provide examples of design patterns that are well-suited to monitoring systems.

II. MOTIVATION

The motivation for exploring the topic of the use of design patterns in the development of the architecture of monitoring systems comes from the increasing demand for robust and scalable monitoring systems in various industries such as finance, healthcare, and telecommunications. The rapid growth of technology has led to the development of more complex and distributed systems, which require advanced monitoring capabilities to ensure their proper functioning.

However, building a monitoring system that is both scalable and maintainable can be a challenging task. It is difficult to predict all possible scenarios and requirements that the system may face in the future, making it hard to maintain and update the system over time. This is where design patterns come into play. By using proven design patterns, developers can build monitoring systems that are easier to maintain, more flexible, and more scalable [12].

The main goal of this article is to assess the possibility of using design patterns in the development of the architecture of monitoring systems, and to demonstrate their relevance and effectiveness [13-15]. By exploring different design patterns and their applications in monitoring systems, this article aims to provide a comprehensive overview of the benefits of using design patterns in monitoring systems development [16].

This article will be valuable to developers and architects who are involved in the development of monitoring systems, as well as to anyone interested in learning about the benefits of using design patterns in software development.

III. PROBLEM STATEMENT

Requirements analysis is an important part of the software development process. It involves collecting and documenting the needs and constraints of stakeholders to ensure that the final product meets their expectations. At this stage, it is necessary to analyse and document the requirements for the monitoring system.

System requirements are the most detailed technical requirements, and they describe how the system will be designed and implemented. System requirements are often expressed in the form of functional and non-functional requirements, and they represent a plan that the development

team should follow. System requirements are usually collected during design sessions, technical reviews, and other development processes.

A. Functional requirements

Functional requirements describe what the system should do and how it should behave. Examples of functional requirements may include:

1) Data collection and storage: The system should be able to collect data from various sources, such as sensors, devices, and databases, and store them in a centralised location.

2) Data analysis: The system should be able to analyse the collected data and provide information about controlled processes in real time. This can include data aggregation, filtering, and visualisation.

3) Alerts and notifications: The system should be able to notify the relevant stakeholders when certain conditions or thresholds are met, for example, when an anomaly or process inconsistency is detected.

4) Reporting and dashboards: The system should provide customised reports and dashboards that allow users to view key performance indicators (KPIs), track progress towards achieving goals and identify areas for improvement.

B. Non-functional requirements

Non-functional requirements describe system qualities such as performance, reliability, and security. Next, examples of non-functional requirements will be analysed:

1) Scalability: The system should be able to handle a large amount of data and users and be able to zoom in and out as needed. Vertical scaling is characterised by an increase in the bandwidth of an individual server or resource, for example, by increasing computing power or memory, which allows you to handle a large load. Horizontal scaling involves adding more servers or resources to handle the increasing load by distributing the workload across multiple machines.

2) Flexibility: The system should be designed in such a way that it can easily adapt to changing requirements without requiring significant changes in its underlying architecture. In the context of monitoring systems, flexibility is important because monitoring requirements can change over time. For example, it may be necessary to add new sensors or devices, as well as to reconfigure the system considering changes in the controlled environment. Flexibility allows for greater maintainability and extensibility.

3) Reliability: The system should be able to work 24/7 without any downtime and provide accurate and reliable data. In the context of monitoring systems, this is important, since any failure can lead to large financial losses, downtime and potentially dangerous situations. One of the ways to achieve reliability is redundancy. Redundancy involves the duplication of critical components or subsystems in the system to ensure that if one component fails, another can take its place. For example, backup power supplies, network interfaces or data storage devices can be added to the monitoring system to increase reliability. Another way to achieve reliability is fault tolerance, which involves designing the system in such a way that it continues to function even when a component fails. Fault tolerance can be achieved by adding mechanisms such as error detection and correction or automatic failover. In general, reliability engineering involves considering all potential points of failure in the system and developing mechanisms to prevent or mitigate the consequences of these failures.

4) Compatibility: The system must be able to interact with other systems and devices using open standards and protocols. In the context of monitoring systems, compatibility can be used to achieve integration with other software components, devices, or platforms to perform their functions effectively. For example, a monitoring system in a manufacturing facility may need integration with sensors, programmable logic controllers (PLCs) and other industrial automation systems to collect data and perform analysis. The monitoring system must be designed in such a way as to be compatible with these various systems. In addition, the use of standard communication protocols, such as MQTT, REST, can help to implement compatibility between different systems.

5) Maintainability: Maintainability is the ability of a system to remain in good condition over time, which covers all actions related to maintaining and improving the quality of the system, including bug fixes, code refactoring and system updates. The serviced system is easy to understand, modify and expand, and it is less prone to errors and defects.

IV. IMPLEMENTATION

The architecture of the platform for intelligent environmental monitoring "Digital Ecomonitoring" is presented using a component diagram (Fig. 1).

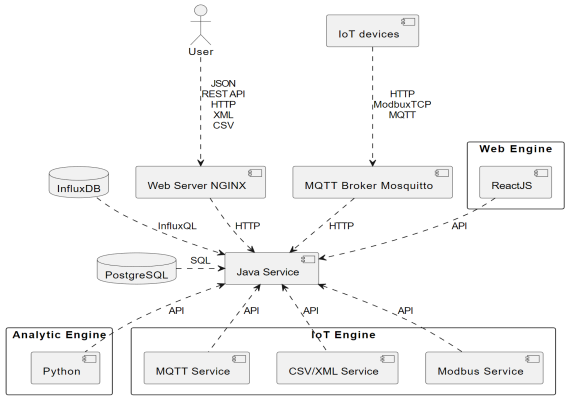

Fig. 1. Component diagram for the platform for intelligent environmental monitoring "Digital Ecomonitoring"

The "Digital Ecomonitoring" platform is designed to provide monitoring and analysis of environmental data in real time, as well as the implementation of emission forecasting. Users also have the ability to configure alerts based on predefined thresholds, which allows them to take proactive measures in response to environmental changes.

The platform has a multi-level architecture with several components working together. The InfluxDB time series database is used to store measurements read from controllers or uploaded by the user to the platform [insert link]. The PostgreSQL relational database management system is used to store dashboard and widget settings, accounts and roles, as well as the assignment of access rights [link]. ReactJS is used

to create user interfaces in the digital platform [place link]. Python is used as an analytical tool for processing data collected by the monitoring system, as well as for predicting values for emissions [link to source]. NGINX web server is used to process incoming requests from clients and forward them to the corresponding components of the digital platform.

The process of data collection and storage in the Digital Ecomonitoring platform is implemented using the Factory pattern. The abstract Data Collector class is a base class that allows you to create new classes responsible for new sensors without diving into the specific details of their implementation. Data Collector is part of Java Service. In the same way, the abstract Data Storage class is able to create new instances of data warehouses.

The process of data processing, analysis and visualization in the platform is implemented using the Decorator pattern, which allows you to add behavior to a single object without affecting the behavior of other objects in the system. In this case, all additional methods for analysis and forecasting are located in the analytical component implemented by Python.

The visualization process in the platform is implemented with an architecture similar to the MVC pattern. In this case, Java Service is a controller that manages communication between databases and ReactJS, which are a Model and a View, respectively [19-20].

The notification process is not clearly expressed in this architecture and is part of the Java Service, which does not allow it to be attributed to any pattern.

For those who want to build monitoring system architecture, there is such a solution as ThingsBoard. ThingsBoard is an open-source solution for IoT platforms. ThingsBoard is used to manage devices, data collection, processing and visualization of collected information. ThingsBoard allows to conveniently organize the process of collecting data from various devices, use a large number of widgets to build informative dashboards that can help with managerial decision-making.

Component diagram for monolithic architecture of ThingsBoard (Fig. 2).

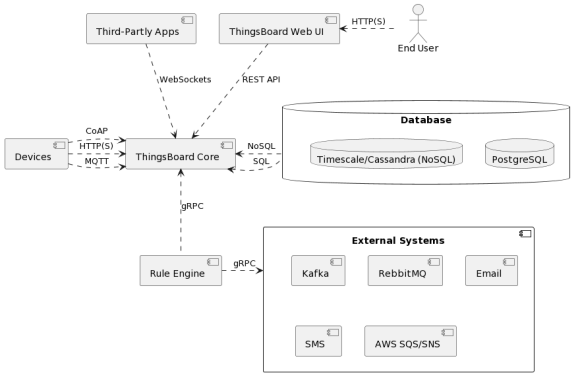

Fig. 2. Component diagram for monolithic architecture of ThingsBoard

The monolithic architecture of ThingsBoard is very popular as it makes it cheaper and faster to develop a monitoring system, which can help to implement it faster. With the help of various protocols, such as HTTP(S), MQTT, CoAP, data enters systems from various devices. Each transport protocol allows to send data to the Rule Engine, which allows devices to change behavior according to the information received, and through the ThingsBoard Core service there is an opportunity to access databases to evaluate the correctness of the information and make appropriate changes. It is assumed that the data collection process is implemented using the Decorator or Factory patterns.

Rule Engine is responsible for processing incoming information according to user-defined logic. It is possible to create a filter, configure alerts when threshold values are reached. This component is responsible for notifying users, which is implemented using the Observer pattern [17-18].

The ThingsBoard Core component is responsible for calling the corresponding APIs, managing via WebSocket and tracking the status of connecting devices to the developed system. This component allows to implement devices, users, management rules and connections in the system. It uses the gRPC framework to interact with other components. Also, interaction with databases for storing the received information is implemented through this component, and represents one of the following patterns by architecture: Factory or Decorator.

The ThingsBoard Core component is responsible for processing, analysis and forecasting, the implementation of which also corresponds to the Decorator or Factory patterns.

External systems can receive information from the system using the Rule Engine, which uses gRPC to transfer data to external systems, process data and create processing reports for visualization in ThingsBoard.

To organize visualization with the presented system, the MVC pattern is used, which is represented by the following components: Controller – ThingsBoard Core, View – ThingsBoard Web UI, Model – Database.

Component diagram for microservices architecture of ThingsBoard (Fig. 3).

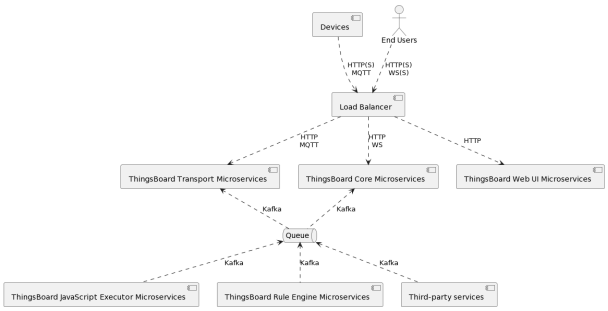

Fig. 3. Component diagram for microservices architecture of ThingsBoard

The microservices architecture of ThingsBoard allows to implement a monitoring system with greater flexibility and maintainability. Data from devices is collected using HTTP(S) and MQTT protocols through the corresponding components that are part of Load Balancer. Then the data is sent to the corresponding services, which transmit them further to other services, process or visualize for users in the system itself.

The applied patterns for the implementation of the monitoring system necessary for the functioning remain the same as for the monolithic architecture, but now there is a separation between the components implementing them into

various services, which contributes to easy scalability and increased maintainability [21-22].

After analysing component diagrams for various monitoring systems, a universal component diagram for monitoring systems was designed, which can help in designing your own monitoring system architecture (Fig. 4).

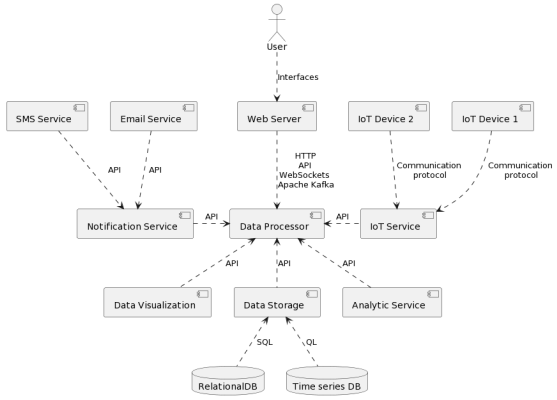

Fig. 4. Component diagram of the monitoring system architecture

In the diagram presented, you can see that the system is composed of microservices, which ensures stable operation, maintainability and easy scalability of the monitoring system. The user communicates with the system via a Web Server, so that the Data Processor component knows exactly what the user wants to do.

The list of Data Processor functions also includes communications with Analytic Service, IoT Service, Data Visualization, Data Storage and Notification Service. Analytic Service organizes the analysis and forecasting of the data available in the system. IoT Service communicates with different IoT devices that the monitoring system is connected. Data Visualization displays the data in user-friendly format. Data Storage stores the data in the monitoring system. Notification Service is responsible for informing users of the exceedance of thresholds or for regularly communicating the status of the monitoring system and related objects.

The process of data collection and storage for the monitoring system, implemented using the Factory pattern, it presented using the class diagram (Fig. 5).

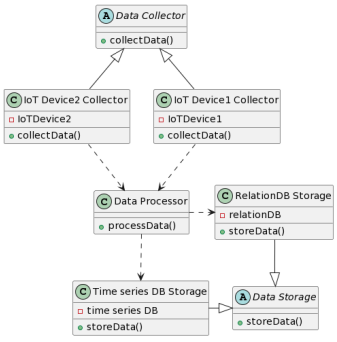

Fig. 5. Class diagram of the process of data collection and storage for the monitoring system

On the class diagram, there are several abstract classes that allow to easily add new elements to the monitoring system without making changes to its structure. So, Data Collector defines the methods that will be used when implementing specific Collector classes. And Data Storage records what functional features databases connected to the monitoring system, both relational and time series databases should have.

Sequence diagram of the process of data collection and storage for the monitoring systems (Fig. 6).

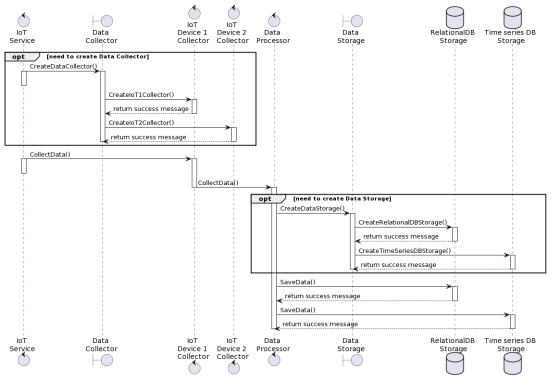

Fig. 6. Sequence diagram of the process of data collection and storage for the monitoring system

In the sequence diagram shown above, there is not only the process of data collection and storage, but also the creation of instances from an abstract base class that implement the appropriate collection method or database to save the collected data.

The process of analyzing and predicting data in the system can be implemented using the Decorator pattern that will allow to add behavior to a separate object without affecting the behavior of other objects in the system. Thus, it's possible to add new methods for data processing and forecasting without the risk of disabling existing methods.

Class diagram of the process of analyzing and forecasting data for the monitoring system (Fig. 7).

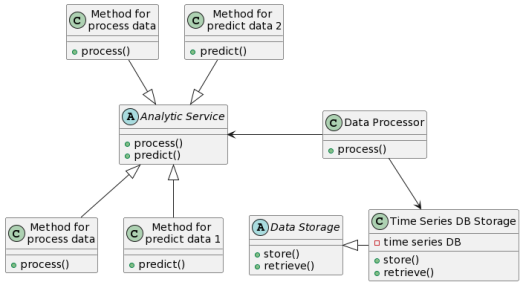

Fig. 7. Class diagram of the process of analyzing and forecasting data for the monitoring system

Methods for data processing and forecasting are extended using the Decorator pattern using the basic abstract class Analytic Service. Similarly, the Time Series DB Storage class is implemented, created according to the abstract Data Storage class.

Sequence diagram of the process of analyzing and forecasting for the monitoring system (Fig. 8).

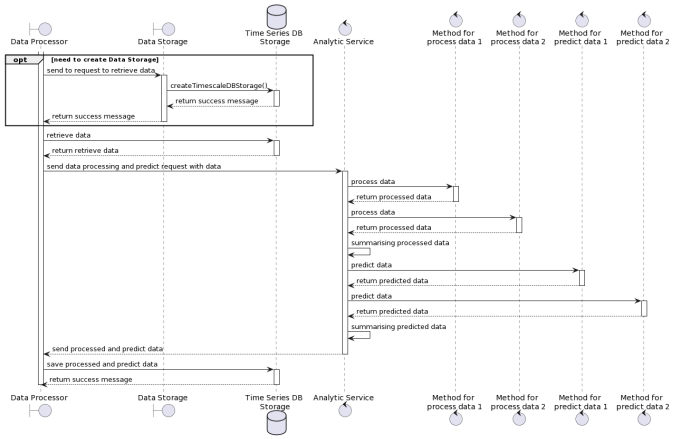

Fig. 8. Sequence diagram of the process of analyzing and forecasting data for the monitoring system

The diagram shows the process of data processing and forecasting, the process of which begins with the creation of a data warehouse according to the abstract base class Data Storage for a time series database. If such a database exists, the Data Processor immediately accesses the database and extracts the necessary information. The received information is sent to the Analytic Service, when it is processed and forecasted using previously established methods in the same way.

The visualization process can be performed using an MVC pattern. This can help to simplify maintenance and system updates. Using this pattern can help achieve separation of the tasks.

Class diagram of the process of visualization data for the monitoring system (Fig. 9).

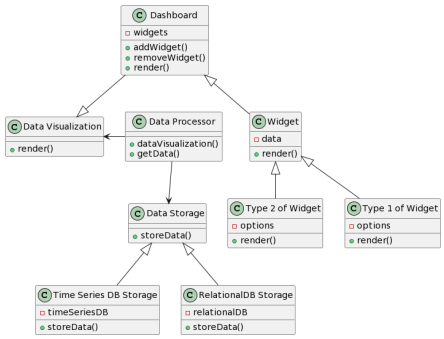

Fig. 9. Class diagram of the process of visualization data for the monitoring system

In this case, the Data Processor will be a Controller that will interact between the Model and the View, which are represented by Data Storage and Data Visualization, respectively. The Model is a database repository that can support both relational databases and time series databases, the View is associated with the Dashboard class, which implements widgets defined in the dashboard system.

Sequence diagram of the process of visualization data for the monitoring system (Fig. 10).

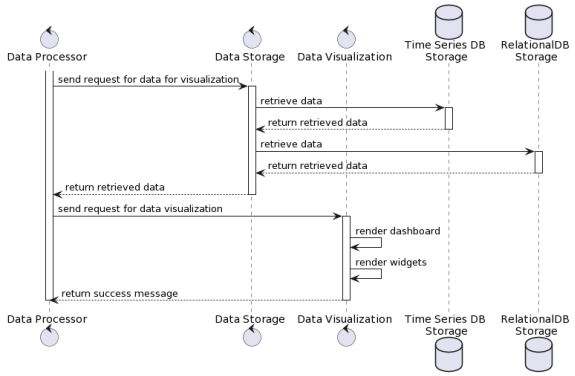

Fig. 10. Sequence diagram of the process of visualization data for the monitoring system

The diagram shows the interaction of the elements of the system built according to the MVC pattern.

The process of notifying users in the monitoring system can be implemented using the Observer pattern. This pattern allows you to update the values of related objects when the observed objects change.

Class diagram of the process of notification users for the monitoring system (Fig. 11).

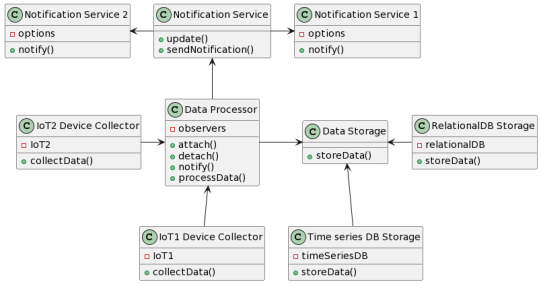

Fig. 11. Class diagram of the process of notification users for the monitoring system

This diagram shows the process of notifying users by applying the Observer pattern, which allows to support instantons change in the state of an object with changes in the observed objects.

Sequence diagram of the process of notification users for the monitoring system (Fig. 12).

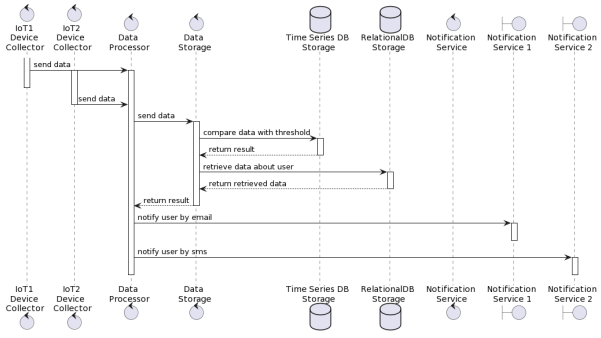

Fig. 12. Sequence diagram of the process of notification users for the monitoring system

In this sequence diagram, the process of notifying users of the monitoring system occurs when the values received from IoT devices exceed the set range of acceptable values. Data Processor, Data Storage and the databases themselves change their state when updates are required from IoT

devices. Also, the Notification Service can change its state in those situations when it is necessary to notify the system user of the events that are taking place.

V. EVALUATION

The design of the monitoring system architecture depends on the non-functional requirements that will need to be implemented. The following is a list of patterns that can implement the non-functional requirements listed above.

1) Observer pattern - to implement reliability and maintainability by monitoring the state of the object and notifying its dependent elements of any changes.

2) Decorator pattern: To implement vertical scaling, flexibility and maintainability in order to dynamically add functionality to an object without affecting the behaviour of other objects.

3) Factory pattern: to implement vertical scaling, flexibility and maintainability in order to create objects without specifying the exact class of the object to be created

4) Microservices pattern: to implement horizontal scaling.

5) Model-View-Control (MVC): to achieve maintainability dividing into three main components: the model, view and control.

VI. CONCLUSION

Monitoring systems can solve a wide range of tasks depending on its purpose. Some of them are:

1) Data collection and storage: The monitoring system should collect data from various sensors and devices, process them and store them in a database for further analysis. The Decorator or Factory patterns can be used to create objects representing different types of data.

2) Data analysis and processing: Once the data is collected, the monitoring system needs to analyse it to extract meaningful information. The Decorator or Factory patterns can be used to add new analysis capabilities to the system without changing the existing structure.

3) Data visualisation: The monitoring system should present the data in a clear and understandable form for the user. The Model-View-Controller (MVC) pattern can be used to separate data from the user interface, allowing developers to create different representations of the same data without affecting the underlying data model.

4) Notifying users about problems: The monitoring system should notify users when certain conditions are met, for example, when the sensor detects an abnormal value or when the device goes offline. The Observer pattern can be used to trigger alerts when certain events occur.

Conclusion: By considering and implementing best practices and design patterns, it is possible to ensure that the architecture of the monitoring system is scalable, flexible and easy to maintain. This will allow the system to effectively meet the needs of the organisation over time as monitoring requirements change.

VII. REFERENCES

[1] D. Gurdur *et al.*, 'Knowledge Representation of Cyber-physical Systems for Monitoring Purpose', *Procedia CIRP*, 2018, vol. 72, pp. 468–473.

[2] P. I. Sosnin, Arhitekturnoe modelirovanie avtomatizirovannyh sistem: uchebnik. Sankt-Peterburg : *Lan'*, 2020 (in Russian).

[3] N. Nazar, A. Aleti, and Y. Zheng, 'Feature-based software design pattern detection', *Journal of Systems and Software*, 2022, vol. 185, pp. 1–12.

[4] D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen, 'Efficiently detecting structural design pattern instances based on ordered sequences', *Journal of Systems and Software*, 2018, vol. 142, pp. 35–56.

[5] S. K. Lo, Q. Lu, L. Zhu, H.-Y. Paik, X. Xu, and C. Wang, 'Architectural patterns for the design of federated learning systems', *Journal of Systems and Software*, 2022, vol. 191, p. 111357.

[6] J. Arm, Z. Bradac, O. Bastan, J. Streit, and S. Misik, 'Design pattern for the runtime model-based checking of a real-time embedded system', *IFAC-PapersOnLine*, 2019, vol. 52, no. 27, pp. 127–132.

[7] Z. Moudam and N. Chenfour, 'Design Pattern Support System: Help Making Decision in the Choice of Appropriate Pattern', *Procedia Technology*, 2012, vol. 4, pp. 355–359.

[8] F. Pfister, V. Chapurlat, M. Huchard, and C. Nebut, 'A Design Pattern meta model for Systems Engineering', *IFAC Proceedings Volumes*, 2011, vol. 44, no. 1, pp. 11967–11972.

[9] A. Ampatzoglou, O. Michou, and I. Stamelos, 'Building and mining a repository of design pattern instances: Practical and research benefits', *Entertainment Computing*, 2013, vol. 4, no. 2, pp. 131–142.

[10] J. Dong, D. S. Lad, and Y. Zhao, 'DP-Miner: Design Pattern Discovery Using Matrix', in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, Tucson, AZ, USA: IEEE, Mar. 2007, pp. 371–380.

[11] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, 'A methodology to assess the impact of design patterns on software quality', *Information and Software Technology*, 2012, vol. 54, no. 4, pp. 331–346.

[12] A. V. Kychkin, A. I. Derjabin, O. L. Vikent'eva, and L. V. Shestakova, 'Shablony proektirovanija programmnogo obespechenija kiberfizicheskih sistem zdanij', *Prikladnaja informatika*, 2020, vol. 15, no. 86, pp. 48–62 (in Russian).

[13] C. Liu and P. Jiang, 'A Cyber-physical System Architecture in Shop Floor for Intelligent Manufacturing', *Procedia CIRP*, 2016, vol. 56, pp. 372–377.

[14] J. E. Correa, R. Toro, and P. M. Ferreira, 'A new paradigm for organizing networks of computer numerical control manufacturing resources in cloud manufacturing', *Procedia Manufacturing*, 2018, vol. 26, pp. 1318–1329.

[15] S. J. Oks, M. Jalowski, A. Fritzsche, and K. M. Moslein, 'Cyber-physical modeling and simulation: A reference architecture for designing demonstrators for industrial cyber-physical systems', *Procedia CIRP*, 2019, vol. 84, pp. 257–264.

[16] M. M. Hamdan, M. S. Mahmoud, and U. A. Baroudi, 'Event-triggering control scheme for discrete time Cyberphysical Systems in the presence of simultaneous hybrid stochastic attacks', *ISA Transactions*, 2021, vol. 122, pp. 1–12.

[17] J. Hu, W. Wu, F. Zhang, T. Chen, and C. Wang, 'Observer-based dynamical pattern recognition via deterministic learning', *Neural Networks*, 2023, vol. 159, pp. 161–174.

[18] K. Aljasser, 'Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns', *Computer Languages, Systems & Structures*, 2016, vol. 45, pp. 1–15.

[19] B. V. Ivanovich, B. V. Vladimirovich, N. F. Victorovich, B. V. Viktorovich, and A. L. Vitalievna, 'Using MVC pattern in the software development to simulate production of high cylindrical steel ingots', *Journal of Crystal Growth*, 2019, vol. 526, p. 125240.

[20] A. Sunardi and Suharjito, 'MVC Architecture: A Comparative Study Between Laravel Framework and Slim Framework in Freelancer Project Monitoring System Web Based', *Procedia Computer Science*, 2019, vol. 157, pp. 134–141.

[21] N. I. Bazenkov *et al*, 'An Office Building Power Consumption Dataset for Energy Grid Analysis and Control Algorithms', *IFAC-PapersOnLine*, 2022, vol. 55, pp. 111–116.

[22] J. Trevathan and S, Schmidtke, 'Open-source Internet of Things remote aquiatic environmental sending', *HardwareX*, 2022, vol. 12, pp. 1-22.