Making Use of Design Patterns in IoT Middleware Implementation

Lasse Harjumaa, Ilkka Kivelä, Petri Jyrkkä and Ismo Hakala
Kokkola University Consortium Chydenius, University of Jyväskylä, Kokkola, Finland

Keywords: Internet of Things, Design Patterns, Scalability, Maintainability, Software Design, Wireless Sensor Networks.

Abstract: This paper describes the usage of object-oriented and microservice design patterns to enhance system maintainability. The project involved bringing together data from multiple sensor networks and providing single endpoint for client applications. The middleware consists of purpose-specific components, databases and various off-the-shelf IoT components. Key lessons learned include the role of design patterns in simplifying complex system interactions and improving understandability. The importance of a modular approach, where design patterns provide a structured framework that promote reuse of proven solutions and reduce technical complexity becomes clear during the implementation of the middleware.

1 INTRODUCTION

Architectures of IoT systems become often complex due to their diverse and distributed nature. The systems consist of a wide range of components, including sensors, devices, gateways, and cloud services, each with unique set of protocols, standards and configurations. This complexity is compounded by the need for real-time data processing, intercommunication, and energy-efficient resource management. Ideally, IoT architectures should be scalable to handle the growing number of devices and adaptable to evolving technologies. Ensuring interoperability between heterogeneous devices and ensuring undisturbed operation across the entire system in case of extending or modifying the components of the system may be a challenging task. Thus, designing scalable and expandable IoT software architectures is important, but at the same time very demanding.

At the component level, version control and dependency management mechanisms provide very helpful tools. However, interrelations of subsystems, microservices and frameworks that the system is composed of, may cause issues, if the responsibilities of components are not clear (Eugster et al. 2003).

Complexity of IoT systems require intersecting knowledge and skills from developers. Systems consist of both hardware and software components. Expertise on deployment, platforms, databases and networking is needed. Domain-specific knowledge and good communication between stakeholders are crucial (Patel and Cassou 2015).

Design patterns provide standardized solutions to recurring problems in software design. They aim at more efficient, maintainable, and scalable code. The object-oriented design patterns introduced by (Gamma et al. 1994) are categorized into three main types: *creational* involving object creation, *structural* focusing on class and object composition, and *behavioral* providing guidelines for managing interactions and responsibilities between objects.

Generic guidelines for assembling IoT systems from multiple microservices and middleware can be derived from software library design principles, since designing reusable software deals with similar challenges: defining uniform data formats, allocating tasks to subsystems and decoupling of components. Real-time system design best practices also provide valuable advice for organizing middleware components without compromising performance or maintainability.

Microservices patterns (Richardson, 2018; Taibi et al. 2018) have been introduced to guide architecture design in cases where the system is composed of several microservices. Patterns such as API Gateway, Database per Service, Aggregator and Circuit Breaker aim at minimizing couplings between services and breaking monolithic applications into smaller, independently deployable modules. These patterns mostly concern high-level architectural design decisions. (Hohpe and Woolf, 2003)

Harjumaa, L., Kivelä, I., Jyrkkä, P. and Hakala, I.
Making Use of Design Patterns in IoT Middleware Implementation.
DOI: 10.5220/0013278000003944
In *Proceedings of the 10th International Conference on Internet of Things, Big Data and Security (IoTBDS 2025)*, pages 254-262
ISBN: 978-989-758-750-4; ISSN: 2184-4976

Serverless architecture approach is perpetually gaining popularity especially in IoT ecosystems. It refers to a cloud computing model where the cloud provider manages the server infrastructure, allowing developers to focus solely on writing code. Code is deployed as small components, even one function, that are executed in response to predefined events, such as HTTP requests. Some popular serverless platforms include AWS Lambda and Microsoft Azure Functions. Serverless solutions are naturally more cost-effective than those that are based on full-scale servers. As with microservices, the design principles of serverless model emphasize decoupling and event-based interaction. For example, (Bardsley et al. 2018) introduces case studies utilizing design patterns to improve performance of serverless implementations.

(Bloom et al. 2018) introduce a pattern catalog for industrial IoT systems. These patterns concern mostly with data flows in edge devices. (Fernandez et al. 2021) have studied the security-oriented design patterns for IoT and propose new patterns to achieve more secure IoT architectures. (Qanbari et al. 2016) present a set of IoT design patterns to aid implementing coherent edge applications. There are also vendor-specific patterns for cloud applications. For example, Amazon provides a pattern catalog for solving common design problems in their cloud platform (Young, 2015). In (Mateos et al. 2010), utilization of Adapter and Dependency Injection patterns have proven to have positive effect on system maintenance in service-oriented environment.

IoT-specific design patterns have been proposed in (Reinfurt et al. 2016). These patterns present solutions to common problems in IoT systems, such as adding Device Gateway when a device cannot directly connect to network due to incompatible communication technologies, and Remote Lock and Wipe to protect the sensor network against security attacks.

A good number of design patterns have been suggested for IoT development. They can be classified, for example, according to the IoT layer they are related to: edge device management, communication, security, local network, integration, IoT cloud, infrastructure, monitoring, application and information model (Chandra and Mahindra, 2016). IoT patterns have proven useful to overcome recurring design problems in complex IoT architectures (Tounsi et al. 2023). On the other hand, some patterns may deteriorate energy efficiency (Crestani et al.2021), so choosing suitable patterns is essential.

In this paper, we will focus mainly on the original design patterns introduced by Gang of Four (Gamma et al. 1994) and examine how the ideas behind the patterns can be applied to a complex IoT environment. They could, for example help in communicating the design decisions effectively, or record and encourage reusing best practices and compare alternative solution proposals (Beck et al. 1996). We believe that design patterns can improve the quality and robustness of IoT software component implementations and reinforce understanding of the IoT system architecture.

2 OUR IOT SOFTWARE ARCHITECTURE

The main goal of the IoT platform built in our research project is to provide stakeholders one easy-to-use endpoint for querying sensor data. The data can originate from multiple sensor networks and the consumer of the data does not need to be aware the details of the actual sensors or protocols used to transfer the data. The principle of the solution is depicted in figure 1. At this point of the project, we have put up two sensor networks, smart home for monitoring living conditions, such as temperature and energy consumption, and construction site monitoring to ensure that building materials remain dry. We are currently planning to add a network monitoring classroom circumstance in food processing education. Later more sources of measurements will be added.

Figure 1: Serving data from a single endpoint.

According to our feasibility studies, using serverless approach in our case would lead to a structure with overlapping responsibilities and scattered data management. We have implemented a dedicated middleware for gathering and transferring measurements to further analysis and permanent storage, because we merge sensor data from multiple sensor networks, each of potentially using different messaging protocols and hardware assemblies. Using a comprehensive IoT architecture from a single provider, such as Microsoft Azure, would not be

straightforward, when we need to combine data from diverging sources. Integration process would most probably require additional middleware within those systems in any case and could even lead to interoperability issues. For real-time interaction, performance would require special attention. In the large, generic-purpose cloud services, network latencies may emerge when dealing with large amounts of data. Finally, need for specialized components, high performance and robust network connections, may lead to sharply increased operating costs in these platforms. The overall architecture model of our sensor network platform is depicted in figure 2.

Storing sensor measurements from varying sources in consistent way and making the data uniformly accessible from a single endpoint presents several challenges. Protocol heterogeneity requires that the architecture support various communication standards and translation mechanisms. Parsing and normalization procedures must be flexible to ensure consistency of data. The high volume of data necessitates efficient, scalable and fault tolerant processing capabilities.

Scalability, maintainability and extensibility are the main quality goals that we aim to achieve in our approach. In the first phase, the solution will integrate measurements from 1-3 separate sensor networks, but the number of networks is expected to grow quickly. Ideally, students participating in projects involving sensor networks will start using the platform for storing and retrieving sensor data, as well as ongoing research projects within the organization. In addition, we need to establish and access point for selected data collections for the use of our collaborative organizations. Thus, the endpoint should authenticate users, and user access rights should be defined in the metadata.

Because of the continuous expansion, the architecture will incorporate networks that are in diverse phases of their lifecycles. This further emphasizes the importance of modularity and flexibility the architecture must possess.

Sensors and actuators are depicted at the bottom of the figure 2. There either is a gateway device gathering the data from the sensor network, or a single sensor can send data directly to the system. The gateway performs the first filtering and aggregation tasks for the data. So far, we have implemented two networks, one consisting of several sensors and a Home Assistant acting as a gateway, and another consisting of individual sensors that deliver their data through LoRaWAN. Data formats, sending frequencies and transfer protocols are completely different in these two cases. The idea is to enable easy addition of new networks later.

Adding a gateway makes it possible to perform more comprehensive filtering before sending data over the network. If the hardware is capable enough, the gateway can even analyse data with AI algorithms, and significantly reduce network load by forwarding only important updates to the cloud.

Message broker serves as common interaction interface for data transfer. MQTT protocol simplifies the data transmission and eliminates need for other messaging protocols after this point, but data coming from different sensor networks still has incoherent formats.

Next, the cloud middleware validates the origin of the incoming data. Even though a specific device has been provisioned for a network, it can be temporarily disabled for some reason. Authorization details and other metadata concerning devices and networks is stored in a database running on SQL Server. In addition to validation, metadata is used to enrich the raw measurement data with useful information, such as human-readable name or location of the device.

Filtered and supplemented data are stored into time series database, which in our case is Influx DB. The database schema is kept as simple as possible, defining measurements just in key-value pairs. Any attached additional information is saved as tags. Simplified schema makes subsequent data management more straightforward.

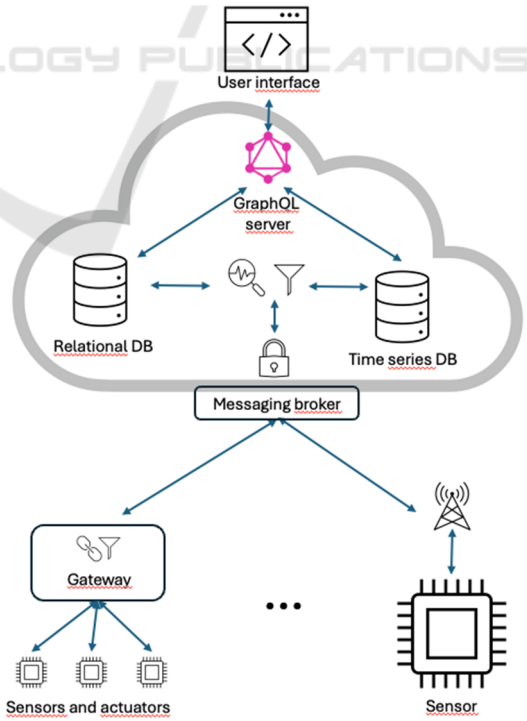

Figure 2: Overall architecture of our solution.

Finally, the data can be queried in a flexible way from a single endpoint provided by a GraphQL server. The GraphQL schema defines objects that are available on the server and provides information on how to retrieve data. This approach provides self-explanatory interface for the clients and obviates dependencies between them and the cloud service. Resolvers that retrieve the time series data can also make use of metadata, enriching query results according to the clients' specific needs. Different views on the information can be implemented by updating the schema, without need to modify the middleware implementation.

The middleware has been implemented in Go language. It is well suited for event-driven architectures dealing with time series data, since it's good performance and robust networking library. It also has built-in support for concurrency, enabling efficient processing of data from multiple sources at the same time (Pike, 2012). Go's concurrency model and performance enable building scalable software that can be expanded when the number of connected devices increases, data formats change, or new features needs to be introduced.

Go produces statically linked binaries, simplifying deployment in variety of IoT hardware, where devices may have limited resources and divergent operating systems. The Go community is growing rapidly, introducing versatile libraries and frameworks that support IoT implementations. There are tools for robotics, networking, microcontroller boards and IoT devices, for example.

While Go is not a traditional object-oriented language, it provides structures, such as interfaces and visibility mechanism, that support creating pattern-oriented implementations (Raiturkar, 2018). Design patterns are not exclusive to object-oriented languages, but general solutions to common problems (Beck et al. 1996).

3 DESIGN PATTERNS UTILIZED

In our middleware implementation, several design patterns were considered useful. First, the software must be able to receive data from heterogenous sources. Currently, the system can adapt to two different types of networks: LoRaWAN and WiFi. ThingPark X IoT Flow is an additional mediator layer that enables forwarding incoming sensor data to a variety of destinations, such as Azure IoT Hub, Amazon Web Services or an MQTT broker (Actility, 2024).

By using the **API Gateway** design pattern, we can simplify the interaction between the heterogenous set of devices (IoT sensors, local sensor networks and gateways), and the cloud services used for data storage and analysis. The principal idea of the API Gateway pattern is to provide a unified interface for the components, hiding the complexity of subsystems (Amazon 2024). The pattern is very similar to the Façade pattern (Gamma et al. 1994) but is applied at subsystem level. In our case, we introduced a MQTT broker as between the actual sensor networks and the middleware. Thus, from the middleware's viewpoint, sensor networks using divergent protocols and data formats appear uniform. A high-level overview of the solution is depicted in figure 3. Instead of each sensor network delivering their data directly to the cloud, all data, regardless of underlying network technology or protocols, is first handed to the broker. For LoRa communication, this is achieved with ThingPark X Flow driver, and in case of Home Assistant, data export plugin is installed in the gateway. Gateway devices are responsible for aggregating and filtering the raw data so that measures are sent to the broker in sensible intervals.

The cloud service is decoupled from the specifics of the individual gateways and devices, making the system more modular and easier to maintain. Furthermore, scaling up the system to manage multiple sensor networks of various types becomes easier.

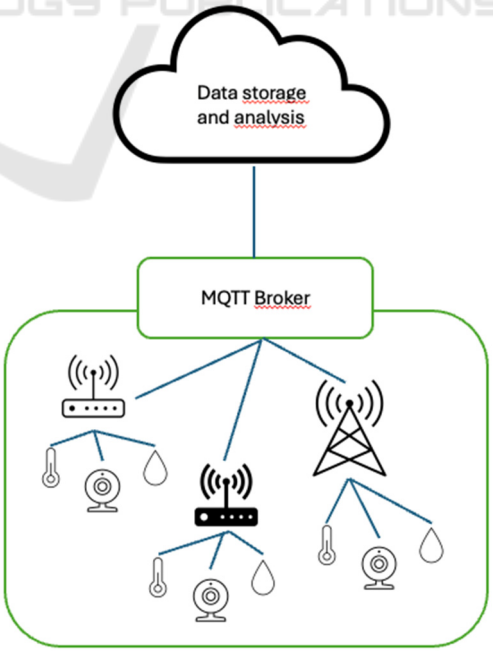

Figure 3: MQTT broker as API Gateway.

The MQTT protocol itself implements the **Publish/Subscribe** design pattern. It is a behavioural pattern where an object maintains a list of its dependents and notifies them of any state changes. This dramatically reduces unnecessary interactions between the objects, since the subscriber objects do not have to poll the publisher object (Hohpe and Woolf, 2003). In MQTT, the broker acts as a subject maintaining the list of subscribers for each topic. When clients register their interest in specific topics, they start receiving updates (messages) from the broker whenever there is new data published to the topics. Another advantage of the Publish/Subscribe pattern is decoupling the sender of the information from the receivers. Thus, the components do not need to know implementation details of each other.

The cloud middleware listens to the events that the broker triggers. In the cloud, three major operations are performed on the incoming messages: validity check, reformatting and enriching the message with corresponding metadata, and saving the data to the time series database. **Chain of Responsibility** pattern allows requests to be passed along a chain of handlers (Gamma et al. 1994). Each handler can either process the request or pass it to the next handler in the chain. This pattern is useful in situations where multiple actions must be taken on the message and keeping the sender unaware (and independent) of the objects that handle the message. The following simplified code illustrates the implementation of Chain of Responsibility pattern in Go language.

```
type MsgSource int
const (
  loRa MsgSource = iota
  ha
)
type Handler interface {
  SetNext(handler Handler)
  Handle(request string)
}
type BaseHandler struct {
  next Handler
}
func (b *BaseHandler) SetNext(handler
Handler) {
  b.next = handler
}
func (b *BaseHandler) Handle(m
MsgSource) {
  if b.next != nil {
    b.next.Handle(m)
  }
}
type LoRaValidationHandler struct {
    BaseHandler
}
func (h * LoRaValidationHandler)
Handle(m MsgSource) {
  if m == loRa {
    // validate LoRa message fields
  } else {
    BaseHandler.Handle(request)
  }
}
type HaValidationHandler struct {
  BaseHandler
}
func (h * HaValidationHandler) Handle(m
MsgSource) {
  if m == ha {
    //validate json from Home Assistant
  } else {
      h.BaseHandler.Handle(request)
  }
}
func main() {
  handler1 := &LoRaValidationHandler{}
  handler2 := &HaValidationHandler{}
  handler1.SetNext(handler2)
  // Read incoming message into msg
  handler1.Handle(msg)
}
```

The code first defines enumeration for identifying the type of the sensor network message is coming from. Then, the Handler interface and the BaseHandler structure define the mechanism of passing the message along the chain and default implementation for setting up the operation chain. The code fragment demonstrates two concrete handlers, LoRaValidationHandler and HaValidationHandler, which validate the input depending on the message source.

We need to define more concrete handlers for the other operations: metadata inclusion and database entry. These are not shown here. Usage of these functions is demonstrated in main function, where incoming message is handed over to the chain of two handlers. Depending on the type of the message, it is handled either in LoRaValidationHandler or HaValidationHandler. When more network types will be introduced, we must create a new handler for it and attach it to the chain, but we can do that with minimal alterations to existing code.

We store metadata about the sensors and the networks they belong to in a relational database. It is more efficient to store the static data in the cloud service than send it with the actual measurement data, which would only increase the network load. Metadata is needed especially for efficient system maintenance. When issues arise, having detailed metadata about devices enables quicker identification and resolution of problems, minimizing downtime

and enhancing the reliability of the system. Furthermore, additional information enhances data clarity and management. By providing context, such as the location, type, and purpose of each sensor, metadata helps in organizing and correlating measurements from different sources, enabling more comprehensive and accurate insights. Furthermore, when items are labelled with descriptive names, it becomes easier to understand and interpret the information, which in turn reduce the risk of errors in data analysis and improves the overall usability of the system.

Enriching the original message with metadata from the server can be achieved with the **Decorator** pattern (Gamma et al. 1994). With the Decorator pattern, additional responsibilities or characteristics can be added dynamically to the object – or data structure in the case of Go language. The following code fragment demonstrates the implementation of the Decorator pattern.

```
type SensorData interface {
  GetValue() string
}
type LoRaSensorData struct {
  devEUI string
  measurement string
  value float64
}
func (o *LoRaSensorData) GetValue()
string {
  return o.value
}
type MetaDataDecorator struct {
  sensorData SensorData
}
func (d *MetaDataDecorator) GetValue()
string {
  return d.sensorData.GetValue()
}
type LocationDecorator struct {
    MetaDataDecorator
    location string
}
func (l *LocationDecorator) GetValue()
string {
  // use l.sensorData.devEUI to fetch
  // location information from DB
  return l.location
}
```

The SensorData interface will be implemented by all sensor data types. In this example, it declares function GetValue to demonstrate retrieving of measurement values. LoRaSensorData is the concrete sensor data struct. MetaDataDecorator struct is the base decorator, which the actual decorators are based on. LocationDecorator is the concrete decorator that enhances the basic sensor data with location information. In real implementation, we would fetch all relevant additional information about the device in addition to location.

The parser function for incoming MQTT messages in the cloud is implemented using the **Strategy Pattern**. This pattern allows us to define a family of algorithms, encapsulate each one, and make them interchangeable (Gamma et al. 1994). When the system needs to be upgraded to manage new message formats, we can implement a new parser function that implements the common interface. The following simplified code shows the basic structure of the parsing strategy.

```
type PayloadParser interface {
  Parse(msg mqtt.Message)
}
type HAPayloadParser struct{}
func (p *HAPayloadParser) Parse(msg
mqtt.Message) {
  // Parsing logic for Home Assistant
  // aggregated payload
}
type BinaryPayloadParser struct{}
func (p *LoRaPayloadParser) Parse(msg
mqtt.Message) {
  // Parsing logic for LoRa payload
}
func incomingMqttMessageHandler(msg
mqtt.Message) {
    var parser PayloadParser
    switch {
    case strings.HasPrefix(msg.Topic(),
"homeassistant"):
        parser = &HAPayloadParser{}
    case strings.HasPrefix(msg.Topic(),
"thingpark"):
        parser = &LoRaPayloadParser{}
    default:
        // Handle unknown topic
        return
    }
    parser.Parse(msg)
}
```

Using GraphQL server helps to reduce couplings between our cloud service and data consumers by allowing clients to request exactly the data they need, no more and no less. Clients are not dependent on specific data structures, enabling evolving the API without breaking clients. GraphQL schema provide a clear contract between the server and the clients and enables creating different views to data. GraphQL also simplifies data fetching by aggregating multiple data sources into a single endpoint, thus reducing the need for consumers to make sequential API calls, further decoupling the data provider from the consumer.

This approach conforms to both the "traditional" **Façade** pattern, GraphQL server component acting as a facade by providing a unified interface to a set of network-specific measurements, and the microservice-oriented API Gateway pattern, which also aims at providing clients a single access point for fetching data from multiple sources. GraphQL schema can also be updated to accommodate new measurement types without need to update the client applications.

Since we want to the system be extensible with new sensors and networks, even with device types that do not yet necessarily exist, we must prepare for data payloads whose details are currently unknown. To deal with future extensions, we have structured the code base into core modules and drivers that handle the different types of sensors. Using drivers to modularize code conforms to the **Adapter** pattern. Each driver acts as an adapter, providing a consistent interface to deal with with different data sources. Thus, the Adapter approach allows future extensions to our application without changing its core logic. Each driver is implemented as a separate file.

Table 1 summarizes the previously described pattern examples in our solution. Some of the patterns are utilized at architectural level, meaning that the pattern is principally realized with off-the-shelf components using only configurative modifications. Some patterns, in turn, have been implemented in the code components from the scratch.

Table 1: Summary of design patterns utilized.

Pattern	Usage
Publish/ Subscribe	Used at architectural level to reduce network load and subsystem dependencies.
API Gateway	Used at architectural level to organize communication between components.
Strategy	Used at component level to decouple modules to enable parsing varying data formats.
Chain of Responsibility	Used at component level to establish flexible processing pipeline for data.
Decorator	Used at component level to supplement the raw data.
Façade	Used at component level to integrate data from multiple sources into single access point.
Adapter	Used at project level to enable modular increments to the system.

Applying patterns at architectural level provides a high-level blueprint for organizing system components and their interaction. Patterns help at creating a robust, scalable, and flexible IoT architecture that can easily adapt to evolving technological requirements and integrate new components with minimal disruption.

4 LESSONS LEARNED

Removing Unnecessary Couplings

Ensuring usefulness and clarity of data is particularly important in complex IoT systems consisting of numerous sensors or even multiple sensor networks. Enriching IoT measurements with metadata not only improves understandability but also enhances data management, collaboration, and system maintenance. One major risk regarding this is tight coupling between the data and metadata, which can lead to inflexibility. It must be ensured that a change in metadata does not necessitate extensive modifications across the entire system, which would increase maintenance efforts. Additional processing required by the metadata integration may also cause security vulnerabilities or performance issues. Therefore, careful planning is essential when designing the integration components.

Improving Scalability

Interoperability issues arise from the heterogeneity of devices and protocols used in IoT ecosystems. Manufacturers often use proprietary standards, making seamless communication of devices from different manufacturers difficult. Lack of standardization can result in fragmented systems where data integration becomes complex and error prone. Additionally, compatibility issues between old and new devices can hinder system upgrades and expansions. Bad interoperability also means bad scalability.

As the number of devices within the system increases, exponentially more data streams through the system, which can strain network bandwidth, storage, and processing resources. This can lead to performance bottlenecks and increased latency, affecting negatively in real-time data processing and decision-making.

Improving Communication

Making use of patterns, whole set of implementation structures and subsystem components can be communicated quickly between developers. Patterns also help in comparing and justifying design

decisions. Since IoT applications tend to be complex, improving the understandability of the system architecture may help future maintenance efforts significantly. Design patterns offer a shared vocabulary, making it easier to discuss and document system components and interactions.

Adding to Security and Robustness
Reusing proven solutions contributes to the system robustness. Patterns aid designers in recognizing situations where design reuse is possible or advisable. Several well-tested solutions to common architectural challenges are described in patterns, including managing hardware failures or isolating critical components to prevent system-wide breakdown. By incorporating design patterns, developers can build IoT systems that can maintain high availability under varying conditions.

5 CONCLUSIONS

The IoT field is still in exponential growth. New companies, technologies, devices, protocols, platforms and versions emerge continuously and in large quantity. Integration and maintenance challenges will keep troubling IoT system development in foreseeable future. Design patterns help in setting clear design goals for the system.

Scalability and maintainability can be promoted at all levels of system development. Applying patterns in solving or refactoring individual design problems at object or function level helps in creating elegant implementations that can be easily understood and upgraded. Structuring larger components or subsystems with pattern-oriented solutions improves the extensibility and integrability of the system at larger scale. Design patterns give ideas for designing robust IoT software architectures and tackle complexity of large heterogenous systems. In addition to the newer microservices and IoT patterns, the object-oriented patterns can be utilized in IoT context.

This article contributes to the body of knowledge in IoT architectures and design, providing practical guidance for system architects and developers dealing with IoT environments that must manage dynamic workloads, changing requirements and future expansion. We provide concrete examples how design pattern concepts can be applied in real-world scenarios, bridging the gap between abstract ideas and practical implementation. We believe this helps practitioners and researchers to understand the nuances of scalability issues or integration challenges involved in IoT projects. Our project of bringing data from multiple sensor networks together into single middleware hopefully inspires researchers and developers working on similar projects.

REFERENCES

Actility (2024). ThingPark X IoT Flow Overview. *https://docs.thingpark.com/thingpark-x/latest/*.

Amazon (2024). What is Amazon API Gateway? *https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html*.

Bardsley, D., Ryan L. and Howard J. (2018). Serverless Performance and Optimization Strategies. In *IEEE International Conference on Smart Cloud*, pp. 19-26.

Beck K. et al. (1996). Industrial experience with design patterns. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pp. 103-114.

Bloom, G., Alsulami, B., Nwafor E. and Bertolotti, I. C. (2018). Design patterns for the industrial Internet of Things. In *14th IEEE International Workshop on Factory Communication Systems*, pp. 1-10.

Chandra, G. S., and Mahindra, T. (2016). Pattern language for IoT applications. In *PLoP Conference*, pp. 1-8.

Crestani, A., Tetu, R., Douin, J.-M. and Paradinas, P. (2021). Energy Cost of IoT Design Patterns. In *8th International Conference on Future Internet of Things and Cloud*, pp. 383-387.

Eugster, P., Felber, P., Guerraoui, R. and Kermarrec, A. (2003). The many faces of publish/subscribe. In *ACM Computing. Surveys*, vol. 35, 2, pp. 114–131.

Fernandez, E. B., Washizaki, H., Yoshioka, N. and Okubo, T. (2021). The design of secure IoT applications using patterns: State of the art and directions for research. In *Internet of Things*, vol. 15.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, USA.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, USA.

Mateos, C., Crasso, M., Zunino, A. and Campo, M. (2010). Separation of concerns in service-oriented applications based on pervasive design patterns. In *Proceedings of ACM Symposium on Applied Computing)*, pp. 849-853.

Patel, P. and Cassou, D. (2015). Enabling high-level application development for the Internet of Things. In *Journal of Systems and Software*, vol. 103, pp. 62-84.

Pike, R. (2012). Go at Google: Language Design in the Service of Software Engineering. *https://go.dev/talks/*.

Qanbari S. et al. (2016). IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications. In *IEEE First International Conference on Internet-of-Things Design and Implementation*, pp. 277-282.

Raiturkar, J. (2018). *Hands-On Software Architecture with Golang*. Packt Publishing Ltd, UK.

Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F. and Riegg, A. (2016). Internet of things patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, pp. 1-21.

Richardson, C. (2018). *Microservices patterns: with examples in Java*. Simon and Schuster.

Taibi, D., Lenarduzzi, V. and Pahl, C. (2018). Architectural Patterns for Microservices: A Systematic Mapping Study. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, pp. 221-232.

Tounsi, I., Saidi, A., Hadj Kacem, M. and Hadj Kacem, A. (2023). Internet of Things design patterns modelling proven correct by construction: Application to aged care solution, In *Future Generation Computer Systems*, vol. 148, pp. 395-407.

Young, M. (2015). *Implementing Cloud Design Patterns for AWS*. Packt Publishing Ltd, UK.