

## Article

# UML Profile for Messaging Patterns in Service-Oriented Architecture, Microservices, and Internet of Things

Tomasz Górski 

Institute of Computer Science, University of Gdańsk, Wita Stwosza 57, 80-308 Gdańsk, Poland;  
tomasz.gorski@ug.edu.pl

**Abstract:** The exchange of information among information technology (IT) systems is inevitable. Service fulfillment often involves sending and receiving messages. The article presents a set of messaging patterns for service-oriented architecture, microservices, and messaging protocols for the Internet of Things. The paper describes selected patterns that are the result of current research work. In addition, patterns introduced in open-source frameworks such as ZeroMQ have also been included. Moreover, the set includes Enterprise Integration Patterns. All considered messaging patterns have been described using the stereotype extensibility mechanism of the Unified Modeling Language (UML), and their complete set has been included in the new *UML Profile for Messaging Patterns*. The paper also shows the manner of integration flow modeling. In the illustrative examples, both the integration flow modeling diagram and the profile have been used to describe the communication in the context of the *Integrated services* view of the 1+5 architectural views model. The profile has been designed in the visual paradigm tool and revealed in a public repository for the community.

**Keywords:** interoperability; service-oriented architecture; microservices; Internet of Things; Unified Modeling Language; 1+5 architectural views model



**Citation:** Górski, T. UML Profile for Messaging Patterns in Service-Oriented Architecture, Microservices, and Internet of Things. *Appl. Sci.* **2022**, *12*, 12790. <https://doi.org/10.3390/app122412790>

Academic Editors: José María Álvarez Rodríguez and Javier García-Heras Carretero

Received: 20 November 2022

Accepted: 10 December 2022

Published: 13 December 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Service provisioning and execution more and more often requires cooperation and thus the exchange of information among IT systems. Various integration styles are used. Among the most widely known are the following: file transfer, shared database, remote procedure invocation, and messaging. The most flexible and loosely coupled is the last one, and, thus, the paper concentrates on that style. Integration flows enable communication between systems using various data formats or communication protocols. The set of Enterprise Integration Patterns (EIP) for integration flows was defined by Hohpe and Woolf [1] and is widely known in the community. However, there are differences in the implementation and naming of particular patterns for different message brokers. For example, the Apache Camel integration framework introduces the Change Data Capture pattern that can be used as the Messaging Bridge pattern [2]. In addition, there are new, specialized versions of EIPs for the currently used message brokers, especially in lightweight architectures such as microservices. Widely used is an open-source messaging queue ZeroMQ [3]. That framework defines new specializations of two EIPs: Request–Retry and Publish–Subscribe. Moreover, messaging patterns are still a topic of research studies. Recently, the Saga pattern was enhanced for distributed transactions in microservices architecture by Daraghmi et al. [4]. In addition, Martinez et al. [5] showed an implementation of the Publish–Subscribe pattern in microservices architecture that employs container orchestration in Kubernetes. Additionally, Aziz et al. [6] emphasized in their literature review that integration solutions using Enterprise Service Bus are mainly realized in service-oriented architecture, and more research is required on the Internet of Things (IoT) and cloud-based systems.

When designing the exchange of messages between systems, it is important to be able to model integration flows in a unified manner. Practitioners commonly use the Unified

Modeling Language for software architecture descriptions. Ozkaya et al. [7] surveyed practitioners as to discover the usage of UML diagrams and architectural views. Informational (99% of asked professionals) and functional (96% of surveyed experts) views are the most popular. The work revealed that the UML class diagram is the most commonly used (85% of inquired specialists) for data structure modeling, whereas the UML deployment diagram is usually adopted for infrastructure modeling (71% of questioned professionals). It should be also noted that the UML activity diagram is quite frequently used for data flow modeling (65% of queried professionals). UML offers stereotypes as extensibility mechanisms to provide new semantics. Stereotypes are grouped into logically consistent sets called profiles. In recent work, Petrasch and Petrasch [8] presented Data Integration Patterns strictly connected with information systems interoperability. However, those patterns focus on the data, and UML has been used to model data integration tasks.

There is a gap between the originally defined set of Enterprise Integration Patterns and the ones that are currently being developed. Therefore, there is a necessity to gather a set of messaging patterns that also include academic patterns and those designed in open-source frameworks. For modeling purposes, there is also a need for structural elements that are responsible for realizing actions representing messaging patterns. Last but not least is the modeling method of integration flows situated in the architectural views model.

The contribution of the paper comprises the new *UML Profile for Messaging Patterns* with stereotypes for patterns resulting from the research carried out: Daraghmi et al. [4], Martinez et al. [5], Zhong et al. [9], and Livaja et al. [10]. The profile includes new patterns in the context of the open-source messaging queue ZeroMQ [3]. The profile also includes the Enterprise Integration Patterns for message flows defined by Hohpe and Woolf [1]. The profile has been designed using the Standard version of the Visual Paradigm tool. The profile file *UMLProfile4MessagingPatterns.vpp* is stored under Git version control and is published as a public GitHub repository [11]. In the 1+5 architectural views model, the *Integrated services* view is dedicated to modeling the communication among cooperating parties [12]. It should be emphasized that this is a unique and specially separated architectural view for the aspect of communication between services or cooperating systems. Therefore, the article focuses on the presentation of this architectural view. In the context of the view, the article depicts a method of modeling integration flows with the use of UML component diagrams and specialized UML activity diagrams, called *integration flows diagrams*.

Figure 1 presents the overview of the architectural description of integration flows. The figure emphasizes the *Integrated services* view and both contexts: structural with cooperating components and dynamic with modeled message flow. The UML component diagram identifies the components involved in the message exchange and the interfaces of those components. The integration flows diagram shows the flow of the message, taking into account the responsibility of the components for performing individual actions in the flow.

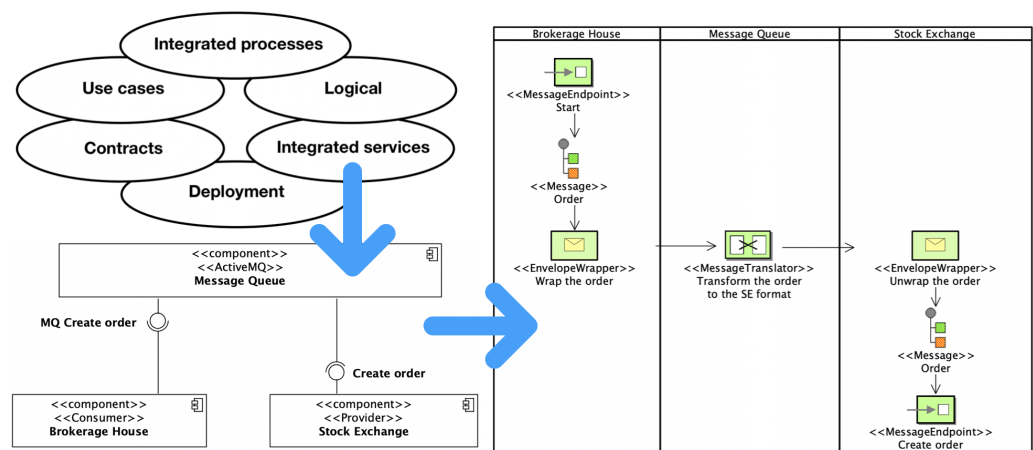


Figure 1. Scheme of using the profile in designing the integration flow.

The remaining part of this paper has the following structure. Section 2 presents related work. Section 3 introduces stereotypes declared in the *UML Profile for Messaging Patterns*. The section describes stereotypes declared for academic patterns, Enterprise Integration Patterns, and their specialized versions in open-source frameworks. Section 4 shows the modeling method of integration flows in the context of the *Integrated services* view in two examples. Section 5 encompasses a discussion on the pros and cons of the profile and method. Section 6 summarizes the work done and indicates further research directions.

## 2. Related Work

When designing the exchange of messages between systems, it is important to be able to model integration flows in a unified manner. Modeling of software and IT systems architecture is commonly done using UML [7]. The language itself does not provide mechanisms for modeling messaging patterns. However, stereotypes as extensibility mechanisms of the UML can be applied to provide new semantics for patterns [13]. Stereotypes are grouped into logically consistent sets called profiles. Generally, constructing UML profiles is a simple and flexible way to provide semantically new modeling elements for use in various fields. The Internet of Things allows for acquiring real-time data from multiple spatially distributed devices. In that field, Thramboulidis and Christoulakis [14] showed a UML profile that supports the transformation of interfaces in the IoT environment to representational state transfer (REST) type ones. In addition, Marouane et al. [15] introduced a profile with concepts related to real-time databases and integrated with the Object Constraint Language (OCL) to enforce the variation points consistency. The STS4IoT UML profile was introduced by Plazas et al. [16]. The profile simplifies the definition of IoT applications and their integration into other information systems, such as stream data warehouses. Moreover, digital twin bridges the physical and virtual spaces. Wang et al. [17] proposed a profile that uses Systems Modeling Language (SysML) stereotypes to represent the system design of a digital twin in a unified manner. The profile includes system design digital models for the following layers: virtual space, system services, relationships, and digital twin data. SysML itself is defined as an extension of a subset of the Unified Modeling Language using a profile mechanism. Furthermore, OPC unified architecture (UA) is a platform-independent standard for message-based communication between clients and servers on various types of networks used to facilitate information exchange. OPC UA has been adopted in various domains, such as power grids, building automation, and smart devices, to support the interoperability of involved systems. These domains also use the Unified Modeling Language as the standard notation for data or system modeling. The use of various notations in the same domain causes compatibility issues. Lee et al. [18] addressed this issue by presenting an approach for transforming OPC UA to UML to improve their compatibility and integration. In the approach, the authors rigorously analyzed the semantics of OPC UA elements and established a mapping between OPC UA and UML modeling means. The converse approach was proposed by Pauker et al. [19]. They proposed an automatic transformation from UML class diagrams to OPC UA information models by extending UML to guarantee the transformation preserves all relevant information. Analyzing the current literature on the subject, it can be seen that UML profiles are used in many areas. However, there is no up-to-date profile for modeling integration flows.

The design of integration flows requires more than just stereotypes. A recent review by Kirpitsas and Pachidis [20] showed the evolution of software development methods. They emphasized that in parallel significant advancements occurred in the field of the software architecture description and pointed out the 1+5 architectural views model [12]. Indeed, recently published research papers show a trend of applying a broader architectural perspective to the design of diverse IT systems. The communication aspect gains importance also in blockchain applications. For example, Han et al. [21] present the deployment view of their solution and use UML deployment diagrams. Although they have not formally constructed a new profile, they have proposed specific stereotypes for IoT, Software Guard Extensions (SGX), and blockchain nodes, services, and protocols, e.g.: <<IoTDeviceNode>>.

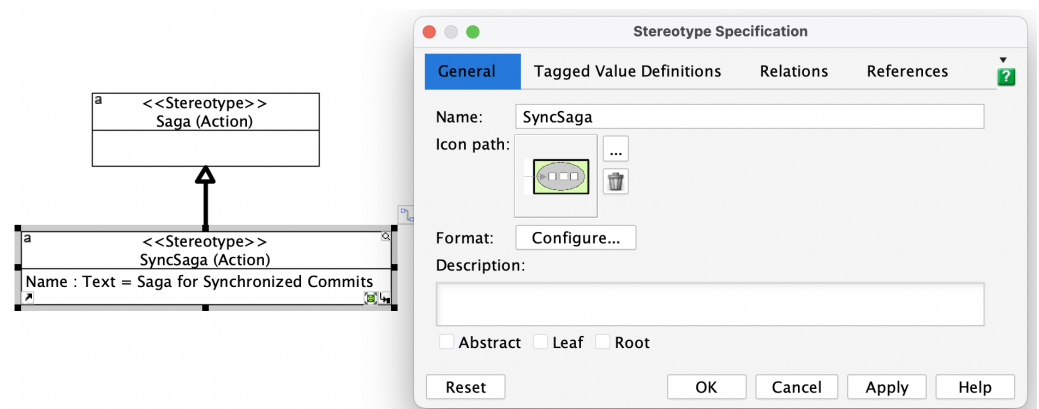
<<IoTGatewayNode>>, <<SGXServerNode>>, <<distributedStorageService>>, and <<blockchainService>>. Moreover, Ahmed et al. [22] proposed an incentive trust model based on blockchain with a privacy-preserving threshold ring signature scheme for vehicular ad hoc networks (VANETs). They focused on *Logical* and *Deployment* views to help comprehend the information exchange between systems. For architectural description, they used four UML diagram types: class, sequence, communication, and deployment. They also proposed specific UML stereotypes for VANETs, e.g.: <<VehicleNode>> for a vehicle node and <<RSUNode>> for a roadside unit. In both cases, the authors used selected views of the 1+5 model to present the architecture of their solutions in the context of communication among nodes. In addition, Akhilesh et al. [23] used the UML activity diagram in the *Use cases* view, UML component diagram in the *Logical* view, and UML deployment diagram in the *Deployment* view to show the design of the automated penetration testing framework for IoT devices. This work has underlined the need for wider and more precise architectural description giving an example of the 1+5 model.

With the above in mind, the author decided to prepare the *UML Profile for Messaging Patterns* that takes into account a wider and up-to-date set of messaging patterns. The profile has been used in the integration flows modeling method placed in the integrated services view of the 1+5 architectural views model.

### 3. UML Profile for Messaging Patterns

Stereotypes extend the standard UML model element types [13]. For the message pattern, the *UML Action* model element type has been assumed as a base type for stereotypes. It applies to Enterprise Integration Patterns, patterns for ZeroMQ, and specialized academic patterns. In addition, for the structural elements, the *UML Component* model element type has been applied as a base type for stereotypes. It applies to message queues and enterprise services buses. Finally, for the messaging protocols, the *UML Control Flow* model element type has been used as a base type for stereotypes. Among others, it applies to the following protocols: Message Queuing Telemetry Transport, Advanced Message Queuing Protocol, and Constrained Application Protocol. Stereotypes are defined in UML profile diagrams. Apart from the declaration of the new stereotype, there can be defined tagged values as attributes associated with that stereotype.

Figure 2 presents the Stereotype Specification window for the academic <<SyncSaga>> stereotype. As can be seen, the *Name* tagged value has been declared for that stereotype, which stores its full name. Stereotype specification is also a place for adding a unique icon for the stereotype.



**Figure 2.** Stereotype Specification window for the <<SyncSaga>>.

Proposed stereotypes do not change the set of UML etamodel (M2) elements available for use in modeling integration flows. However, at the UML Model (M1) level we can employ UML base types and dynamically modify their semantics by applying specific stereotypes.



### 3.1. Stereotypes for Academic Patterns

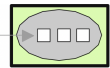
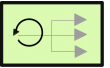


This article takes into account four patterns resulting from research work. All are Enterprise Integration Patterns extensions: Saga and Publish–Subscribe Channel. The Publish–Subscribe Channel pattern has attracted the most attention of scientists. Three new versions of the pattern have been introduced. In addition, the Saga pattern has been elevated on a higher level of transaction integrity.

The specialized version of the Saga (Software Automation, Generation, and Administration) pattern was proposed by Daraghmi et al. [4]. The pattern was proposed to maintain data consistency across distributed microservices transactions. However, it lacks isolation, which means that reading and writing data from an incomplete transaction is allowed. The specialized version uses the quota cache and the commit-sync service. In case a microservice fails to be completed, the other microservices run rollback transactions, and the changes only affect the cache layer. No wrong commit occurs in the database layer. Database commit is executed when all transactions are completed successfully. The introduced enhancement for the Saga pattern seems important enough to define a new version of the pattern. Therefore, the author named the new pattern Saga for Synchronized Commits and proposed the new stereotype <<SyncSaga>> and an icon associated with it.

The specialized version of the Publish–Subscribe Channel pattern in microservices architecture was proposed by Martinez et al. [5]. The idea relies on the deployment of communication components within containers of the Kubernetes (K8s) orchestration cluster. The component implements a publish/subscribe pattern of communication among microservices. The cluster provides self-healing and auto-scaling services. As the introduced enhancement for the Publish–Subscribe Channel pattern seems important, the author named the new version of the pattern Resilient Publish–Subscribe and proposed the new stereotype <<ResilientPS>>, and the icon associated with it. Zhong et al. [9] extended the same EIP pattern, but for geo-textual data. In their approach, subscribers get geo-textual object pairs rather than individual objects, which leads to avoiding duplicates. The author named the new version the Pairwise Publish–Subscribe pattern and proposed the new stereotype <<PairwisePS>>. Livaja et al. [10] proposed a cluster-based distributed geospatial publish–subscribe system. The cluster matches incoming geospatial objects in real-time with a set of stored subscriptions. The author named the new version the Separate Publish–Subscribe pattern and proposed the new stereotype <<SeparatePS>>.

Table 1 shows the summary of the four patterns resulting from research work: Saga for Synchronized Commits, Resilient Publish–Subscribe, Pairwise Publish–Subscribe, and Separate Publish–Subscribe.

**Table 1.** New stereotypes for the patterns resulting from research work.

Name	Stereotype	Base Type	Icon
Saga for Synchronized Commits	<<SyncSaga>>	UML Action	
Resilient Publish–Subscribe	<<ResilientPS>>	UML Action	
Pairwise Publish–Subscribe	<<PairwisePS>>	UML Action	
Separate Publish–Subscribe	<<SeparatePS>>	UML Action	

New and unique icons have been designed, in a color scheme of the Enterprise Integration Patterns, for the academic patterns. A prefix characterizing the distinguishing property is included in the names of stereotypes that expand the Publish–Subscribe Channel. The suffix is the same.

### 3.2. Stereotypes for Enterprise Integration Patterns

Messaging makes applications loosely coupled by communicating asynchronously. It makes communication more reliable because the two applications do not have to be running at the same time. The messaging system is responsible for transferring data from one application to another, so the applications can focus on what data they need to share but not worry so much about how to share them. Standard actions within such communication have been enclosed in Enterprise Integration Patterns and can be divided into the following categories: messaging systems, messaging channels, message construction, message routing, message transformation, messaging endpoints, and system management.



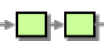
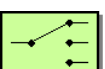
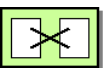
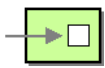
According to the Enterprise Integration Patterns website, there are 61 patterns [24]. For each of them, the corresponding stereotype is declared. For the majority of them, the icon is also attached to the stereotype in the profile. Some of the patterns do not have declared icons at all, e.g., Format Indicator, Scatter–Gather, or Canonical Data Model. The defined stereotypes for selected Enterprise Integration Patterns are presented below. A brief description of the purpose of each of the shown patterns is provided.

For messaging systems patterns the following stereotypes have been declared:

- `<<MessageChannel>>`—the source application adds the information to a particular message channel and the target application retrieves the information from that channel;
- `<<Message>>`—data must be converted into one or more messages and then sent through messaging channels;
- `<<PipesAndFilters>>`—divides a larger processing task into a sequence of processing steps (Filters) that are connected by channels (Pipes);
- `<<MessageRouter>>`—receives a Message from the Message Channel and sends it to one of the alternative channels due to conditions fulfillment;
- `<<MessageTranslator>>`—translates one data format into another;
- `<<MessageEndpoint>>`—a client of the messaging system that can be used to send or receive messages.

Table 2 shows stereotypes and icons for the patterns in the messaging system category.

**Table 2.** Stereotypes with icons for the patterns in the messaging system category.

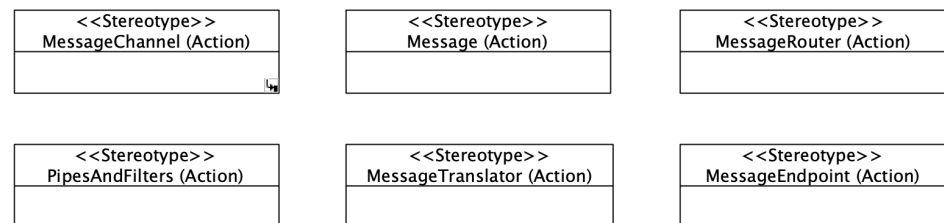
Name	Stereotype	Base Type	Icon
Message Channel	<code>&lt;&lt;MessageChannel&gt;&gt;</code>	UML Action	
Message	<code>&lt;&lt;Message&gt;&gt;</code>	UML Action	
Pipes and Filters	<code>&lt;&lt;PipesAndFilters&gt;&gt;</code>	UML Action	
Message Router	<code>&lt;&lt;MessageRouter&gt;&gt;</code>	UML Action	
Message Translator	<code>&lt;&lt;MessageTranslator&gt;&gt;</code>	UML Action	
MessageEndpoint	<code>&lt;&lt;MessageEndpoint&gt;&gt;</code>	UML Action	

The stereotypes for selected patterns from particular categories are presented below:

- `<<PublishSubscribeChannel>>`—one input channel splits into multiple output channels. When an event occurs, a copy of the message is delivered to each of the subscribers' channels;
- `<<ChannelAdapter>>`—operates as a messaging client to the messaging system and invokes functions via an application programming interface;

- `<<RequestReply>>`—sends a pair of messages, one to the target application and the second back to the source application, in separate channels;
- `<<Saga>>`—provides a way to define a series of related actions in a route that should be either completed successfully or not executed;
- `<<DurableSubscriber>>`—makes the messaging system save published messages while the subscriber is disconnected;
- `<<EnvelopeWrapper>>`—wraps application data inside an envelope that is compliant with the messaging infrastructure. The message should be unwrapped when it arrives at the destination;
- `<<ContentEnricher>>`—accesses an external data source so as to augment a message with missing information.

Figure 3 presents the UML profile diagram with stereotypes for messaging systems.



**Figure 3.** UML profile diagram with declared stereotypes for messaging systems.

Table 3 shows stereotypes and icons for selected patterns from messaging channels, message construction, message routing, and message transformation categories.

**Table 3.** Stereotypes for selected messaging patterns.

Name	Stereotype	Base type	Icon
Publish–Subscribe Channel	<code>&lt;&lt;PublishSubscribeChannel&gt;&gt;</code>	UML Action	
Channel Adapter	<code>&lt;&lt;ChannelAdapter&gt;&gt;</code>	UML Action	
Request–Reply	<code>&lt;&lt;RequestReply&gt;&gt;</code>	UML Action	
Saga	<code>&lt;&lt;Saga&gt;&gt;</code>	UML Action	
Durable Subscriber	<code>&lt;&lt;DurableSubscriber&gt;&gt;</code>	UML Action	
Envelope Wrapper	<code>&lt;&lt;EnvelopeWrapper&gt;&gt;</code>	UML Action	
Content Enricher	<code>&lt;&lt;ContentEnricher&gt;&gt;</code>	UML Action	

### 3.3. Stereotypes for Structural Components and Messaging Protocols

Two trends in communication can be observed. The first is derived from corporate solutions, and communication is related to the realization of a business process. In the second one, communication most often occurs between two services. The first type employs enterprise service buses, and the second type is intended for lighter message brokers. Therefore, stereotypes for both environments have been included in the profile. The abstract `<<EnterpriseServiceBus>>` stereotype has been proposed. The rest are concrete stereotypes that inherit from this abstract stereotype and represent actual frameworks.

Figure 4 depicts the UML profile diagram with an inheritance hierarchy of stereotypes for structural components representing enterprise service buses.

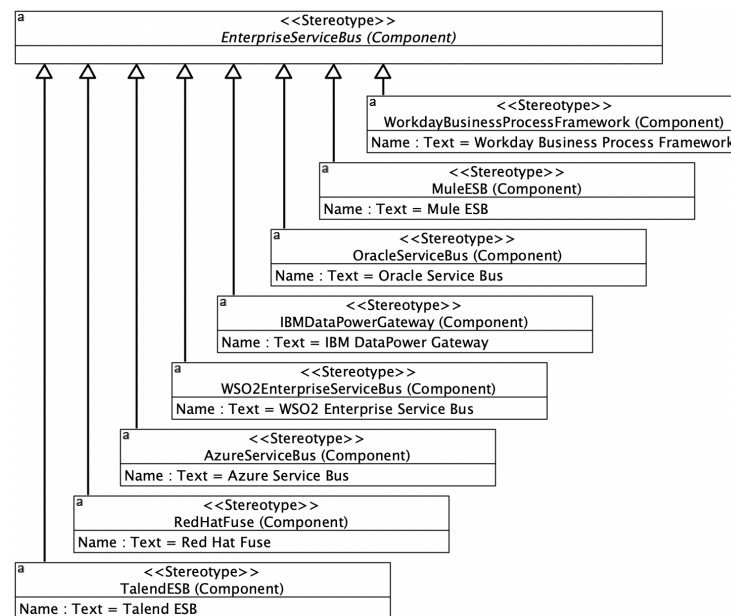


Figure 4. Stereotypes for enterprise service buses.

Similarly, for message queues, one abstract `<<MessageQueue>>` stereotype has been declared. Three concrete stereotypes stand for actual message broker frameworks: `<<ActiveMQ>>`, `<<RabbitMQ>>`, and `<<ZeroMQ>>`. Table 4 shows stereotypes declared for message queues.

Table 4. Stereotypes for message queues.

Name	Stereotype	Base Type
Apache Camel ActiveMQ	<code>&lt;&lt;ActiveMQ&gt;&gt;</code>	UML Component
RabbitMQ	<code>&lt;&lt;RabbitMQ&gt;&gt;</code>	UML Component
ZeroMQ	<code>&lt;&lt;ZeroMQ&gt;&gt;</code>	UML Component

In addition, there are various communication protocols, especially in the IoT area. That is why the stereotypes have been declared for the most commonly used IoT messaging protocols [25]. Table 5 shows stereotypes declared for messaging protocols.

Table 5. Stereotypes for messaging protocols.

Name	Stereotype	Base Type
Message Queuing Telemetry Transport	<code>&lt;&lt;MQTT&gt;&gt;</code>	UML Control Flow
Advanced Message Queuing Protocol	<code>&lt;&lt;AMQP&gt;&gt;</code>	UML Control Flow
Constrained Application Protocol	<code>&lt;&lt;CoAP&gt;&gt;</code>	UML Control Flow
HyperText Transfer Protocol	<code>&lt;&lt;HTTP&gt;&gt;</code>	UML Control Flow
Extensible Messaging and Presence Protocol	<code>&lt;&lt;XMPP&gt;&gt;</code>	UML Control Flow
Data Distribution Service	<code>&lt;&lt;DDS&gt;&gt;</code>	UML Control Flow



There are no icons declared for structural components, as components can be applied to various stereotypes. The icon would restrain its use in one context. Stereotypes for protocols mark connections between two modeling elements. In the paper, the *UML Control Flow* type element connects two *UML Actions* type constructs. Connections have a standard appearance in UML.

### 3.4. Stereotypes for ZeroMQ Framework Patterns

ZeroMQ defines specialized versions of two EIP patterns: the Publish–Subscribe Channel and Request–Reply. The built-in core ZeroMQ Request–Reply patterns are:

- `<<LazyPirate>>`—polls the socket and receives from it only when it is sure a reply has arrived. If no reply has arrived within a timeout period, the pattern resends a request. The transaction is abandoned if there is no reply after several requests;
- `<<SimplePirate>>`—extends the Lazy Pirate pattern with a queue proxy that allows for transparent communication with multiple servers;
- `<<ParanoidPirate>>`—allows for robust reliable queuing that allows sending and receiving messages at any time but requires its own envelope management;
- `<<Majordomo>>`—adds a service name to requests that the client sends and asks servers to register for specific services. Adding service names turns the Paranoid Pirate pattern from the queue into a service-oriented broker;
- `<<Titanic>>`—stores messages in the message broker to ensure they never get lost;
- `<<BinaryStar>>`—puts two servers in a primary–secondary high-availability pair. At any given time, the active server accepts connections from client applications. The passive server is idle, but the servers monitor each other. If the active server stops working, the passive one takes over as active;
- `<<Freelance>>`—creates a pool of name servers so if one stops working, clients can connect to another. In this architecture, a large set of clients connect to a few servers in a pool directly. The clients connect to the pool, which is the opposite of a broker-based approach such as Majordomo, where clients connect to the broker.

Figure 5 shows, in the UML Profile diagram, the ZeroMQ stereotypes inheritance tree for specializations of the Request–Reply pattern. The `<<Titanic>>` stereotype stands for the most specialized version of the Request–Reply pattern, whereas the `<<Freelance>>` stereotype refers to its most flexible variant.

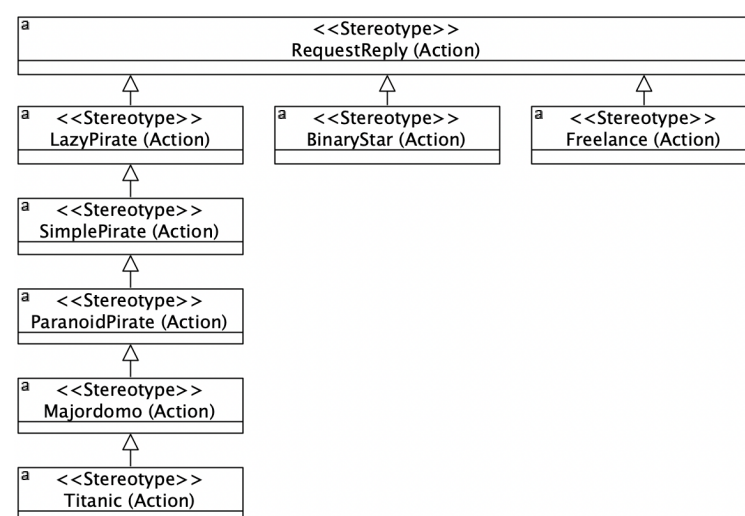

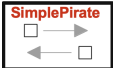







Figure 5. ZeroMQ stereotypes for patterns that inherit from the EIP Request–Reply pattern.

Table 6 shows a few ZeroMQ stereotypes with proposed icons for the Request–Reply pattern specializations. The original icon for the Request–Reply pattern has been modified and a black-red-white color schema has been adopted for icons proposed for stereotypes related to ZeroMQ patterns.

**Table 6.** Selected stereotypes for ZeroMQ specializations of the Request–Reply pattern.

Name	Stereotype	Base Type	Icon
Lazy Pirate	<<LazyPirate>>	UML Action	
Simple Pirate	<<SimplePirate>>	UML Action	
Paranoid Pirate	<<ParanoidPirate>>	UML Action	
Majordomo	<<Majordomo>>	UML Action	
Titanic	<<Titanic>>	UML Action	
Binary Star	<<BinaryStar>>	UML Action	
Freelance	<<Freelance>>	UML Action	





The built-in core ZeroMQ Publish–Subscribe Channel patterns are:

- <<SuicidalSnail>>—allows for the detection of slow subscribers. Devoted to subscribers having service-level agreements to guarantee specific maximum latency;
- <<BlackBox>>—allows for detection and dealing with high-speed subscribers. The pattern breaks subscriber design into a multithreaded one so that sending and reading messages are in separate sets of threads;
- <<Espresso>>—allows for monitoring a publish–subscribe network and works by creating a listener thread that reads a socket and prints anything it gets;
- <<Clone>>—builds a shared key-value store. The pattern allows for updating a shared state across a set of clients. To achieve that, the pattern uses key-value pairs, which represent atomic units of change in the shared state.

Each of these four patterns presented is a direct specialization of the Publish–Subscribe Channel pattern. Similarly here, the original icon for the Publish–Subscribe Channel pattern has been modified and a black-red-white color schema has been adopted for icons proposed for stereotypes related to ZeroMQ patterns.

Table 7 shows ZeroMQ stereotypes with proposed icons for the Publish–Subscribe Channel pattern specializations.

**Table 7.** Stereotypes for ZeroMQ specializations of the Publish–Subscribe Channel pattern.

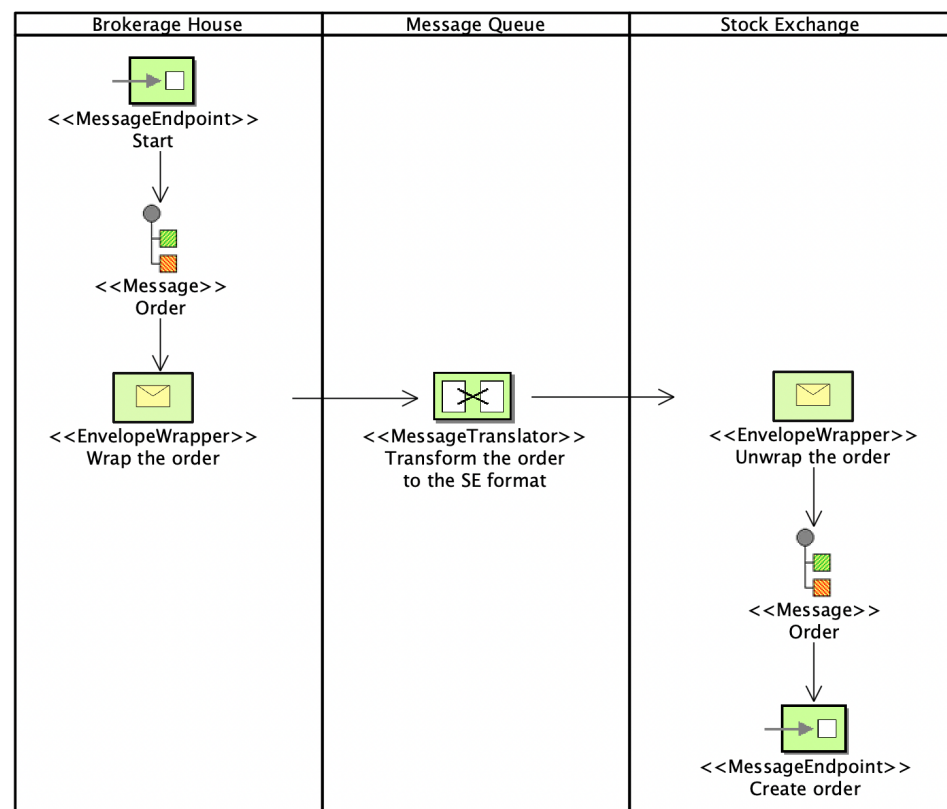
Name	Stereotype	Base Type	Icon
Suicidal Snail	<<SuicidalSnail>>	UML Action	
Black Box	<<BlackBox>>	UML Action	
Espresso	<<Espresso>>	UML Action	
Clone	<<Clone>>	UML Action	

There are two more patterns in the ZeroMQ framework: Client–server and Radio–dish. The client–server pattern allows a single ZeroMQ server to communicate with one or more ZeroMQ clients. After starting the conversation with the client, both the server and the client can send messages asynchronously to each other. The Radio–dish pattern is used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan-out manner. However, both of the patterns are still in draft state. Therefore, there have been omitted in the current version of the profile.

#### 4. Modeling Method with Examples of Integration Flows

The *Integrated services* view shows both static and dynamic aspects of communication. The structural aspect is modeled using a UML component diagram and encompasses the components of cooperating IT systems and the message broker or service bus component. Services are modeled using interfaces realized by components. The most important are the interfaces provided by the components representing IT systems. In these components, the functions necessary to perform the integration flow are implemented. The message queue or service bus component performs flow actions but forwards calls for execution to the system component. That component realizes interfaces, which means that it provides the implementation for interface functions. Therefore, the system component realizes calls for services invoked on the interface. All components are crucial for modeling the dynamic aspect of integration. In integration flow diagrams, components are represented as partitions and services are used as endpoints.

In presented examples, integration flows show the context of a single message transfer from the source application to the destination application, service, or device. The first example shows a message flow of sending orders from the Brokerage House (BH) application to the Stock Exchange (SE) system. Figure 6 presents an integration flow diagram (the specialized version of a UML activity diagram) for the *Send order to the Stock Exchange* flow. This example uses both icons and stereotype names for messaging patterns.



**Figure 6.** The *Send order to the Stock Exchange* integration flow.

Figure 7 presents the UML component diagram in the context of the *Create order in the SE* use case.

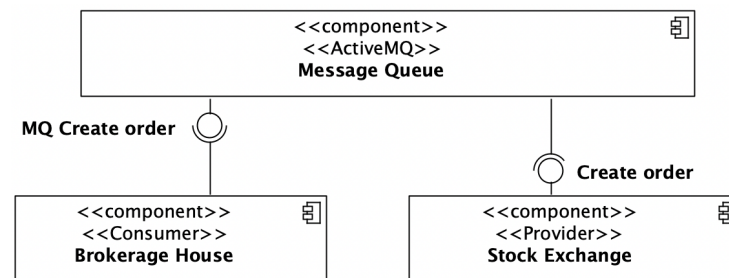


Figure 7. UML Component diagram for BH and SE systems cooperation.

In the UML component diagram, two additional stereotypes from the Service-Oriented Architecture Modeling Language have been used: <<Consumer>> and <<Provider>>. They can be used to clearly denote the difference between the component that realizes the interface (<<Provider>>) and the component that uses the function from the interface (<<Consumer>>). Both components cooperate using a message queue. The SE system enables stock exchange trading. A purchase and sale transaction in the SE system is concluded when two opposite buy and sell orders for the same company appear on the trading market. In addition, in both orders, the buyer agrees to pay the price proposed by the seller.

The second example concerns sending the request for energy produced to IoT devices. Renewable energy prosumer communities exchange energy to optimize usage and cost of energy. Those communities are managed using blockchain networks. The flow illustrates the use of various IoT message protocols.

Figure 8 presents the integration flow diagram for the *Send Request for Energy Produced* integration flow. The second example uses only icons for messaging patterns.

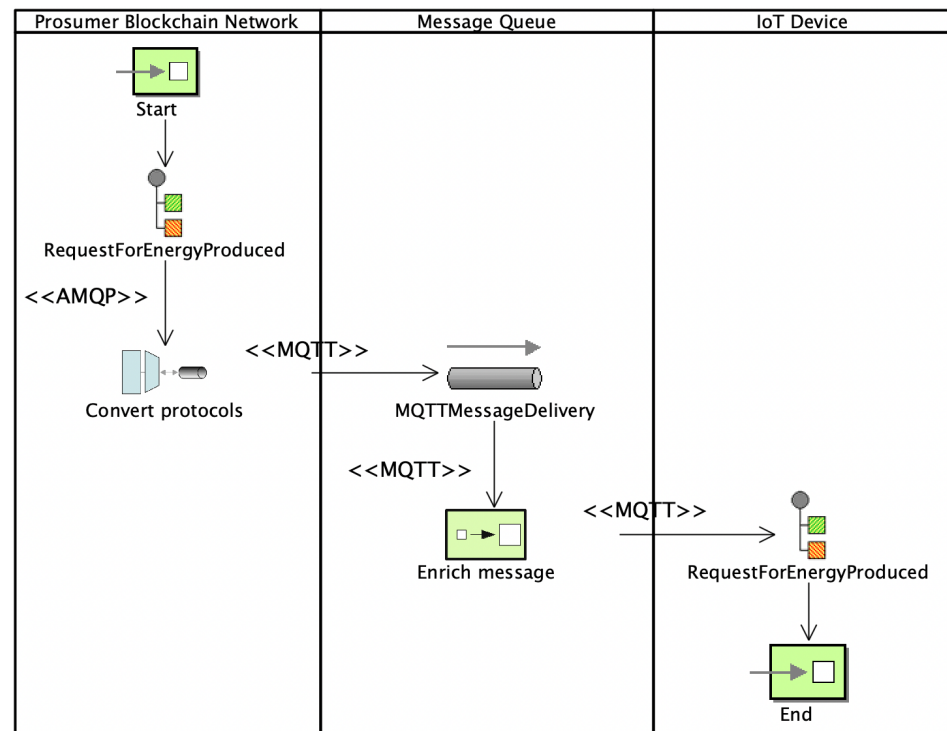


Figure 8. The *Send Request for Energy Produced* integration flow.

## 5. Discussion and Limitations

A wide variety of messaging patterns are included in the profile. Efforts were made to keep the stereotypes names for the patterns brief and concise. However, the full names of patterns, components, or protocols were placed in stereotypes in the form of devoted tagged values. For ease of modeling, the icons for patterns should be unique. That is not fulfilled for all environments. For example, Apache Camel uses the same icon for Messaging Bridge and Change Data Capture patterns. Moreover, another icon is shared also by Threads and Throttle patterns. An important feature of the profile is the uniqueness of the proposed names for patterns and their icons. In the current version of the profile, protocols used in Internet of Things applications are included [25]. For protocols, there are no icons in the profile due to the fact that those stereotypes are attached to directed links between actions, which have standard look in the UML. Instead, the name of the stereotype is used. However, in terms of the Internet of Things, Covert Channels attract more and more researchers' attention. Velinov et al. [26] analyzed that topic, and their work may be a good source of patterns. Currently, they are out of the scope of the profile. The article shows that the area of communication between systems/services is constantly being developed, and more and more flexible patterns are emerging. An interesting is the Freelance pattern introduced in the ZeroMQ framework. It makes the communication mechanism independent of a single intermediary element. Therefore, it eliminates a single point of failure. These types of patterns can become the basis for communication between networks of distributed blockchain nodes. Such exchange of messages may also be organized in a distributed manner. For example, Al-Shaibani et al. [27] proposed a decentralized platform for the stock exchange that is based on blockchain technology. Another vital blockchain application is the managing of energy exchange among prosumer communities [28].

The patterns presented in the work vary in terms of complexity and range from single actions to architecturally complex but clearly defined components. For example, the use of the Resilient Publish–Subscribe pattern can greatly simplify the architectural description of the communication design between cooperating services. That is why the description of the software architecture is so important. The ability to present a design mechanism at a higher level of abstraction enables the definition of patterns that can be reused by the community. The actual limitation is that in the profile there are no detailed descriptions of the patterns. Future work should involve the configuration of constituents of the patterns. It should encompass those standard components provided by the pattern and components that fulfill certain roles that must be delivered by the integration flow designer. The next step is the generation of integration flows using tagged values as configuration parameters of message patterns. This mechanism has been successfully used in the continuous delivery and deployment of complete distributed blockchain applications and node deployment configurations for blockchain networks [29].

The use of the UML profile also positively affects the ease of maintenance of the designed integration models. Model elements in models are independent of stereotypes because they use basic types from the UML language. The tool used connects the profile to integration projects in the form of an external associated file. The profile design can therefore be developed independently, and the changes will be visible in the related integration projects. In addition, updating the profile involves adding more stereotypes or modifying the existing ones. Thanks to this, existing integration projects will be provided with the used set of stereotypes and newly added ones. The author has also intended to prepare the profile in a commonly used and affordable modeling tool, the Visual Paradigm Standard version. The profile has been shared in the public repository and can be used freely.

## 6. Conclusions

The paper introduces the *UML Profile for Messaging Patterns*. The author took a set of Enterprise Integration Patterns as a basis and applied an object-oriented inheritance relationship to model specialized stereotypes for derived patterns. In this way, the consistency of both the set of patterns and stereotypes corresponding to them was achieved.



Only fully developed open-source patterns have been taken into account and presented in the article. The patterns in a draft state have been omitted. During this work, more and more patterns have become visible as a result of scientific work. One of the directions of further work may be the development of a comprehensive library of these patterns with detailed descriptions and architectural support for modeling and generation of integration flows. Specifically, further research patterns for the Internet of Things, containerized environments, and blockchain technology applications will be considered. Moreover, the profile will keenly encompass future open-source proposals.

Using the profile, it is possible to model flows on two levels of abstraction. Abstract stereotypes such as MessageQueue were introduced in the profile. That particular one is suitable for modeling any type of message queue. There is a similar situation with the stereotypes for ZeroMQ. They extend the stereotypes from the Enterprise Integration Patterns set, which is commonly known and used in various frameworks.

It is worth emphasizing that the profile also includes stereotypes for structural elements involved in integration flows. This provides the basis for modeling them in an architectural context. The modeling method, presented in the paper, is limited to one architectural view. The *Integrated services* view shows an excerpt from the architectural description of the integration design. The description method clearly identifies all components involved in the integration. In addition, the proposed *integration flows diagram* assigns responsibilities for performed operations to components and determines the sequence of actions in the integration flow. As a result, the method identifies the structural elements and the dynamics of activities between them. For a full perspective, there should be shown the requirements in the *Use cases* view, and the business process in the *Integrated processes* view.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The author declares no conflict of interest.

## References

1. Hohpe, G.; Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*; Addison-Wesley Professional: Boston, MA, USA, 2004.
2. Apache Camel—An Open-Source Integration Framework. Available online: <https://camel.apache.org> (accessed on 20 November 2022).
3. ZeroMQ—An Open-Source Universal Messaging Library. Available online: <https://zeromq.org> (accessed on 20 November 2022).
4. Daraghmi, E.; Zhang, C.-P.; Yuan, S.-M. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. *Appl. Sci.* **2022**, *12*, 6242. [CrossRef]
5. Martinez, H.F.; Mondragon, O.H.; Rubio, H.A.; Marquez, J. Computational and Communication Infrastructure Challenges for Resilient Cloud Services. *Computers* **2022**, *11*, 118. [CrossRef]
6. Aziz, O.; Farooq, M. S.; Abid, A.; Saher, R.; Aslam, N. Research Trends in Enterprise Service Bus (ESB) Applications: A Systematic Mapping Study. *IEEE Access* **2020**, *8*, 31180–31197. [CrossRef]
7. Ozkaya, M.; Erata, F. A survey on the practical use of UML for different software architecture viewpoints. *Inf. Softw. Technol.* **2020**, *121*, 106275. [CrossRef]
8. Petrasch, R.J.; Petrasch, R.R. Data Integration and Interoperability: Towards a Model-Driven and Pattern-Oriented Approach. *Modelling* **2022**, *3*, 105–126. [CrossRef]
9. Zhong, Y.; Zhu, S.; Wang, Y.; Li, J.; Zhang, X.; Shang, J.S. Pairwise Location-Aware Publish/Subscribe for Geo-Textual Data Streams. *IEEE Access* **2020**, *8*, 211704–211713. [CrossRef]
10. Livaja, I.; Pripuzić, K.; Sovilj, S.; Vuković, M. A distributed geospatial publish/subscribe system on Apache Spark. *Future Gener. Comput. Syst.* **2022**, *132*, 282–298. [CrossRef]
11. UML Profile for Messaging Patterns, GitHub Repository. Available online: <https://github.com/drGorski/UMLProfile4MessagingPatterns> (accessed on 20 November 2022).
12. Górski, T. The 1+5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. *Symmetry* **2021**, *13*, 2000. [CrossRef]
13. Pender, T. Customizing UML Using Profiles. In *UML Bible*; Wiley Publishing, Inc.: Indianapolis, IN, USA, 2003; pp. 687–723.

14. Thramboulidis, K.; Christoulakis, F. UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Comput. Ind.* **2016**, *82*, 259–272. [[CrossRef](#)]
15. Marouane, H.; Duvallet, C.; Makni, A.; Bouaziz, R.; Sadeg, B. An UML profile for representing real-time design patterns. *J. King Saud Univ.—Comput. Inf. Sci.* **2018**, *30*, 478–497. [[CrossRef](#)]
16. Plazas, J.E.; Bimonte, S.; Schneider, M.; de Vaulx, C.; Battistoni, P.; Sebillio, M.; Corrales, J.C. Sense, Transform & Send for the Internet of Things (STS4IoT): UML profile for data-centric IoT applications. *Data Knowl. Eng.* **2022**, *139*, 101971. [[CrossRef](#)]
17. Wang, H.; Li, H.; Wen, X.; Luo, G. Unified modeling for digital twin of a knowledge-based system design. *Robot.-Comput.-Integr. Manuf.* **2021**, *68*, 102074. [[CrossRef](#)]
18. Lee, B.; Kim, D.-K.; Yang, H.; Oh, S. Model transformation between OPC UA and UML. *Comput. Stand. Interfaces* **2017**, *50*, 236–250. [[CrossRef](#)]
19. Pauker, F.; Wolny, S.; Fallah, S.M.; Wimmer, M. UML2OPC-UA Transforming UML Class Diagrams to OPC UA Information Models. *Procedia CIRP* **2018**, *67*, 128–133. [[CrossRef](#)]
20. Kirpitsas, I.K.; Pachidis, T.P. Evolution towards Hybrid Software Development Methods and Information Systems Audit Challenges. *Software* **2022**, *1*, 316–363. [[CrossRef](#)]
21. Han, J.; Zhang, Y.; Liu, J.; Li, Z.; Xian, M.; Wang, H.; Mao, F.; Chen, Y. A Blockchain-Based and SGX-Enabled Access Control Framework for IoT. *Electronics* **2022**, *11*, 2710. [[CrossRef](#)]
22. Ahmed, W.; Di, W.; Mukathe, D. A Blockchain-Enabled Incentive Trust Management with Threshold Ring Signature Scheme for Traffic Event Validation in VANETs. *Sensors* **2022**, *22*, 6715. [[CrossRef](#)]
23. Akhilesh, R.; Bills, O.; Chilamkurti, N.; Chowdhury, M.J.M. Automated Penetration Testing Framework for Smart-Home-Based IoT Devices. *Future Internet* **2022**, *14*, 276. [[CrossRef](#)]
24. Messaging Patterns of EIPs. Available online: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html> (accessed on 20 November 2022).
25. Al-Masri, E.; Kalyanam, K.R.; Batts, J.; Kim, J.; Singh, S.; Vo, T.; Yan, C. Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access* **2020**, *8*, 94880–94911. [[CrossRef](#)]
26. Velinov, A.; Mileva, A.; Wendzel, S.; Mazurczyk, W. Covert Channels in the MQTT-Based Internet of Things. *IEEE Access* **2019**, *7*, 161899–161915. [[CrossRef](#)]
27. Al-Shaibani, H.; Lasla, N.; Abdallah, M. Consortium Blockchain-Based Decentralized Stock Exchange Platform. *IEEE Access* **2020**, *8*, 123711–123725. [[CrossRef](#)]
28. Górski, T. Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules. *Appl. Sci.* **2022**, *12*, 5339. [[CrossRef](#)]
29. Górski, T. Towards Continuous Deployment for Blockchain. *Appl. Sci.* **2021**, *11*, 11745. [[CrossRef](#)]