p-ISSN 1607-3274 Радіоелектроніка, інформатика, управління. 2026. № 1
e-ISSN 2313-688X Radio Electronics, Computer Science, Control. 2026. № 1

ПРОГРЕСИВНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

PROGRESSIVE INFORMATION TECHNOLOGIES

UDC 004.41

A DESIGN PATTERN FOR ENABLING FUNCTIONAL STABILITY IN SOFTWARE SYSTEMS

Bychkov O. S. – Dr. Sc. (Engin.), Professor, Head of the Department of Software Systems and Technologies, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine. ROR: https://ror.org/02aaqv166. ORCID: https://orcid.org/0000-0002-9378-9535.

Moroz M. V. – Post-graduate student of the Department of Software Systems and Technologies, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine. ROR: https://ror.org/02aaqv166. ORCID: https://orcid.org/0000-0001-6953-683X.

ABSTRACT

Context. Modern software systems operate in dynamic and harsh environments where internal and external failures, unexpected disturbances, direct attacks, and resource constraints challenge the consistent provision of core functionalities. In these contexts, ensuring functional stability – where the quality of each system function remains within a predetermined stable range despite failures or environmental anomalies – is critical, especially for safety-critical and high-availability applications.

Objective. The primary objective of this work is to develop and justify an enabling design pattern that provides the architectural backbone for achieving functional stability in software systems. The main focus is to provide a flexible solution that facilitates dynamic adaptation while maintaining robust system behavior.

Method. We propose a novel pattern that combines the dynamic strategy selection capabilities with the loose coupling between components afforded by an event-driven approach. This enabling pattern decouples system components by enforcing communication solely through standardized event types and allows each module to select an appropriate adaptation strategy based on its current context. The described pattern was used to build a design of a real-life example that aims to implement stable object tracking functionality for autonomous quad-platforms. The proposed design was evaluated using design-level metrics alongside qualitative comparisons with existing adaptive approaches.

Results. Our analysis shows that the enabling pattern achieves significant modularity and adaptability. Key object-oriented metrics indicate minimal interdependencies among modules and a clear separation of concerns. The design proposal demonstrates that the pattern supports dynamic behavior adjustment through flexible strategy selection and serves as an enabler for functional stability by providing a robust architectural backbone for software systems.

Conclusions. The scientific novelty of this work is twofold: firstly, the novel pattern is obtained in our study, providing dynamic adaptation through context-aware strategy selection; secondly, functional stability received further development in the area of software architecture. The proposed pattern offers a robust, scalable, and maintainable architectural solution, with significant practical implications for the design of adaptive, resilient software systems.

KEYWORDS: software design patterns, functional stability, event processing, adaptive behavior, autonomous systems.

NOMENCLATURE

C is a set of software components (modules) that form the system SW;

C_i is a specific component of the system SW, responsible for executing one or more functions;

c_i is an internal state or context of component C_i, which affects how the strategy selection function σ behaves;

E is a situation space representing the full range of environmental and operational conditions in which the system may operate;

$E_{expected}$ is a subset of situation space E that contains situations anticipated by the system designer;

$E \backslash E_{expected}$ is a set of unexpected or unhandled situations that are not explicitly anticipated during system design;

F is a set of system functions provided by the system SW;

f_i is a particular function from the set F, implemented by a specific component C_i;

M is a mapping function that defines the system's behavior;

$O(f)$ is a set of all possible output states of the component that provides function f;

$o(f)$ is an output state of the component that provides function f at time t;

P is an enabling architectural software design pattern;

s is a specific situation from space E that may affect the system's functioning at runtime;

$S_{stable}^{(f)}$ is a set of output states considered stable for function f;

ST is a set of strategies available for adaptation within the system;

st is a specific adaptation strategy from the set ST that a component may use to react to a detected event;

DOI 10.15588/1607-3274-2026-1-9

SW is a target software system under analysis, composed of interacting components that together implement the system's core functionality;

T is a transient interval after which the output state of function *f* is expected to reside within the stable set $S_{stable}^{(f)}$;

σ is a strategy selection function that determines the most suitable strategy for a component based on the current situation and internal context.

INTRODUCTION

Modern software systems are increasingly deployed in environments characterized by rapid changes, complex dependencies, unexpected disturbances, and resource limitations. These conditions – ranging from internal failures and external attacks to unanticipated operational anomalies – pose significant challenges to maintaining consistent, reliable functionality. The concept of functional stability [1], defined as the ability of a system to preserve its core functions within predetermined stable boundaries despite adverse conditions, has traditionally been applied in mechanical, physical, and decentralized systems. However, its application to software systems and software architectures is less explored, creating a gap in both scientific research and practical implementations.

The current state of research in adaptive and resilient software architecture emphasizes the use of classical design patterns alongside approaches for dynamic reconfiguration and self-adaptation. These methods provide valuable mechanisms for enabling systems to adjust their behavior in response to changing operational conditions. However, while they offer important insights into dynamic adaptation and fault tolerance, many of these approaches address only isolated aspects of system resilience. Consequently, a comprehensive architectural solution that unifies dynamic adaptation with long-term functional stability remains lacking. This gap motivates our investigation into an enabling design pattern that not only supports flexible strategy selection and decoupled communication between components but also provides a robust framework for maintaining stable system behavior in complex, dynamic environments.

The object of the study is the adaptive process within software systems that encounter a diverse range of operational situations – including both expected and unforeseen disturbances.

The subject of the study is the enabling design pattern, an architectural backbone that integrates dynamic strategy selection with decoupled, event-driven communication among components. This pattern is intended to provide the structural support necessary for systems to maintain stable functionality over time, even under conditions of stress or failure.

The purpose of the work is to develop and justify a pattern-based solution that provides the necessary structural support for achieving functional stability in complex software systems. To achieve this aim, we have set the following tasks: to analyze existing approaches and identify their limitations with respect to functional stability; to design a pattern that is able to fulfill the mentioned requirements; to evaluate the proposed pattern.

1 PROBLEM STATEMENT

Let *SW* be a software system composed of a set of components $C=\{C_1, C_2, ..., C_n\}$ that collectively implement a set of functions $F=\{f_1, f_2, ..., f_r\}$. The system *SW* operates in an environment characterized by a situation space *E*, where each situation $s \in E$ is an element of $\square^k$. These situations represent various conditions – including internal and external failures, disturbances, and other operational anomalies – that the system may encounter.

The system is modeled by a mapping function M: $E \rightarrow O(f)$, where $O(f)=\{ o_1^{(f)}, o_2^{(f)}, ..., o_m^{(f)} \}$ represents the set of output states or responses of a component that provides function *f* after encountering a situation *s*. In response to a situation, a component may either change its state or maintain its current state.

The overall goal is to achieve functional stability for each function $f \in F$ under situation space *E*. Formally, for every function *f*, there exists a subset of stable states $S_{stable}^{(f)}$ such that after a transient interval *T*, the state $o_t^{(f)}$ corresponding to function *f* satisfies formula (1):

$$o_t^{(f)} \in S_{stable}^{(f)}, \forall t > T. \quad (1)$$

To address these challenges, the problem targeted by this article is to develop an enabling software design pattern *P* that provides the architectural backbone for achieving functional stability in dynamic environments. This pattern by its design considerations should support the system *SW* with the ability to satisfy formula (1) despite failures, disturbances, or unanticipated operational conditions. In details, the target pattern *P* should be responsible for next aspects.

– Enabling adaptability. Allowing each component C_i to select a strategy from a set *ST* based on the current situation $s \in E$ and its internal state (context) c_i. This selection is formalized by a function $\sigma:(s,c_i) \rightarrow st \in ST$. By adjusting a strategy component could transition into a stable state, satisfying the formula (1).

– Handling expected situations. It is expected that a system designer defines a subset $E_{expected} \subset E$ that covers the range of anticipated conditions. Recognizing that $E \backslash E_{expected}$ may include unexpected situations, *P* should support default or fallback strategies to mitigate these unhandled cases.

– Supporting extensibility and scalability. Ensuring that the pattern *P* is designed in a modular way to allow easy maintenance, extension, and scaling of both the strategies *ST* and the overall system functions *F*.

2 REVIEW OF THE LITERATURE

Functional stability [1] is defined as the property of an object to preserve the execution of its primary functions over a specified time, within limits set by normative

DOI 10.15588/1607-3274-2026-1-9

requirements, even when exposed to counteractive influences and streams of failures, malfunctions, or errors. This concept is particularly valued in contexts where uninterrupted performance is critical, such as in safety-critical systems or environments prone to frequent disturbances. The methodological approach to achieving functional stability can be divided into four compact stages [2]:

– detection of an abnormal situation associated with degradation in the quality of functioning due to the influence of destabilizing factors;

– identification of an abnormal situation;

– making a decision on restoring the functioning process;

– restoring functioning by redistributing functions and tasks between undamaged elements.

Historically, the concept of functional stability has been applied extensively to mechanical and physical systems – such as for example onboard aircraft systems [3, 4], control and navigation systems [5, 6], and distributed information networks [7, 8] – where reliability under failure conditions is paramount. However, this concept remains poorly defined and underexplored in the domain of software engineering, particularly at the software architecture and design levels. But there are some connected terms that are represented in the area of software architecture: in some academic circles, “functional stability” is a common term, while Western literature often employs related concepts such as self-adaptive, robust, or resilient system design. Further we provide brief definitions of each term and related literature review.

According to the source [9] self-adaptive software is identified as software that possesses the capability to autonomously modify its behavior in response to changes in its operating environment or internal state. Such systems continuously monitor their context and, upon detecting deviations from desired behavior or performance goals, reconfigure themselves automatically to satisfy both functional and non-functional requirements.

The standard [10] defines robustness as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

The CNSS Glossary [11] determines resilience as the ability to prepare for and adapt to changing conditions and withstand and recover rapidly from disruptions. Resilience includes the ability to withstand and recover from deliberate attacks, accidents, or naturally occurring threats or incidents.

These definitions, while distinct in their emphasis, collectively highlight the system’s capacity to maintain and restore core functionality under adverse conditions and are interrelated in that they contribute to the overall goal of designing systems that can endure, adapt, and recover in harsh dynamic environments.

A first notable foundation for achieving adaptiveness in software design is provided by the work of the GoF [12], which shaped modern software architecture. The GoF introduced a catalog of design patterns that offer proven solutions to recurring problems in software design. Among these, the Observer and Strategy patterns stand out as particularly relevant for enhancing system adaptiveness. The Observer pattern decouples the subject from its observers, allowing components to react in real time to state changes. It enables systems to dynamically adjust their behavior in response to environmental shifts, thereby supporting continuous functional performance. By encapsulating interchangeable algorithms, the Strategy pattern empowers systems to select the most appropriate behavior at runtime. This flexibility is essential for maintaining adaptive responses under varying operational conditions.

The thesis [13] which is summarized in the work [14] presents a catalog of design patterns specifically aimed at enabling software systems to adjust their behavior dynamically at runtime. The work systematically categorizes design patterns that support dynamic adaptiveness, outlining how various patterns address different aspects of runtime adaptation. The patterns are organized into clear categories based on their roles within the adaptive process:

– monitoring and analysis patterns focus on continuously observing system behavior and environmental conditions, providing essential data for triggering adaptation;

– planning and decision patterns encapsulate strategies for selecting among multiple behavioral alternatives; it leverages concepts from control theory and optimization to guide adaptive decision-making;

– execution and reconfiguration patterns responsible for the implementation of adaptive changes at runtime, such as dynamic component replacement and state migration, ensuring that the system can reconfigure itself seamlessly.

The thesis focuses on dynamic adaptiveness without addressing the concept of functional stability, so there is an opportunity to integrate these concepts. The works [15, 16] introduce innovative autonomic design patterns that complement the taxonomy of dynamically adaptive systems presented in the thesis. The work [15] proposes an event-based pattern for web services, triggering adaptive responses via system events, while the study [16] presents an adaptive reconfiguration compliance pattern that enables systems to adjust configurations while meeting compliance standards.

The work [17] introduces a comprehensive taxonomy of design patterns aimed at enabling runtime reconfiguration in software systems. It emphasizes the importance of decoupling configuration logic from core business functionality, thereby allowing components to adapt dynamically without disrupting overall operations. Key ideas include mechanisms for dynamic state transfer, policy-driven reconfiguration, and runtime component replacement – each facilitating seamless adaptations in response to changing environments or internal conditions. Overall, this source provides a structured framework for

DOI 10.15588/1607-3274-2026-1-9

understanding how pattern-based approaches can support software adaptiveness.

In the more recent study [18] authors present a framework for dynamic software adaptation that leverages runtime architectural models to manage both planned and unplanned changes. The paper categorizes adaptation into three types – algorithmic, configuration, and architectural – and introduces state machine-based adaptation patterns that explicitly govern component transitions from active to quiescent states.

The article [19] presents a system designed to automatically detect, diagnose, and repair faults in sensor networks during runtime. The framework is built upon the MAPE-K model [20] – an established autonomic computing paradigm which structures the self-healing process into five distinct phases:

– monitor – the system continuously gathers sensor data and other relevant metrics;

– analyze – collected data are processed to detect anomalies or deviations that may indicate sensor faults;

– plan – upon detecting an issue, the system formulates a remediation strategy;

– execute – the planned corrective actions are applied to restore proper operation;

– knowledge – a shared repository is maintained to store historical data, learned patterns, and decision-making rules for future incidents.

By leveraging this model, the framework not only addresses transient errors in sensor data but also supports dynamic adaptation to evolving fault conditions, thereby enhancing the overall reliability and availability of sensor-based systems. Although the framework addresses self-healing, it does not explicitly integrate the broader notion of functional stability into its design. There is an opportunity to combine the self-healing approach with functional stability principles. The framework is tailored to sensor networks, focusing primarily on data acquisition and fault remediation in that context.

In summary, while the reviewed literature proposes many adaptive and self-healing solutions, these approaches predominantly address how systems can alter their behavior in response to change. However, there remains a gap: the concept of functional stability is underexplored at the software design and architectural levels. This gap underscores the need for novel design patterns and methodologies that explicitly integrate functional stability into software architectures, thereby fostering systems that are not only adaptive but also inherently resilient and robust.

3 MATERIALS AND METHODS

Our approach for ensuring functional stability in software systems is based on the assumption that a software component can achieve functional stability if it is capable of dynamically adapting to changes in its operational environment. Specifically, if each software component C_i can detect events that signify changes in the situation space E and then select an appropriate adaptation strategy from a set ST – using a strategy selection function σ – then software system can maintain its output state within a predetermined stable set $S_{stable}^{(f)}$. In this approach, the presence of multiple strategies, including a fallback strategy for handling unexpected events and critical failures, enables the module to adjust its behavior in real time. Consequently, by dynamically switching among these strategies as counteractions to detected disturbances, a software component can preserve the quality of its primary functions and, in turn, contribute to the overall functional stability of the system.

In this section, we introduce our novel design pattern that aims to enable functional stability in software systems while expanding the adaptive patterns collection. Our pattern is born from the idea of combining the flexibility of the Strategy pattern [12] with the dynamic notification features of the Observer pattern [12]. The Strategy pattern excels at allowing interchangeable algorithms, and the Observer pattern provides the ability to trigger actions in response to incoming events. Below, you'll find the UML diagrams and a detailed description of our pattern. These materials will walk you through how the pattern is structured, how each component interacts, and the rationale behind our design choices.

One key aspect of our approach is evolving from the traditional Observer pattern to a more flexible Publish-Subscribe model [21]. In the classic Observer pattern, subscribers must know the Observer (publisher) directly, which can create tight coupling between components. With Publish-Subscribe subscribers don't need to know the publisher. Instead, they simply subscribe to specific event types. This decoupling enhances flexibility and makes it easier to manage complex, dynamic systems.

We should also mention that event handling is a core element of our pattern, serving as the engine for adaptivity and a one of key contributors to functional stability. By centering our design around event handling, we enable a decoupled, dynamic interaction between components, where events trigger specific responses without requiring direct connections between publishers and subscribers. At the same time, it is important to note that we will not be focusing on how events are generated or detected within the system. We assume that events are produced by various entities through different approaches – whether for example by physical sensors, monitoring tools, or through the specialized "monitoring and analysis patterns" [13, 14, 20]. Our primary concern here is how these events are handled to adapt the behavior of individual components and the overall system.

Our pattern is expected to be applied for software components (modules), and under the term "software component" we understand any self-contained, logically cohesive unit that encapsulates a distinct functionality within a system. Each module should be designed to operate independently, featuring well-defined interfaces that facilitate interaction via our event-driven approach. Such modular structure supports the decoupling and dynamic adaptivity of our design. In addition, our pattern acknowledges that software components can exist in

DOI 10.15588/1607-3274-2026-1-9

hierarchical structures. Some modules may function as submodules of larger modules, forming dependency relationships that need careful handling. Our approach is designed to accommodate these nested configurations, ensuring that even when modules depend on one another, the event-driven mechanism maintains robust decoupling and clear communication channels.

With the foundational concepts in place, let's now dive into the details of our pattern. The pattern is a behavioral design approach that enables software components to dynamically adjust their behavior in response to events that arise in a system. By merging the strategic decision-making capabilities of the Strategy pattern with the loose coupling of the Publish-Subscribe model, this pattern allows components to select the most appropriate operational strategy when specific events occur.

When an event is triggered, a dedicated handler within the component is able to evaluate the event context (the event may contain some valuable information, such as reason, creation time, criticality etc) along with a system state and provide a decision about activating the best-suited strategy from a set of available options. For instance, strategy changes may involve initiating a recovery process, modifying the execution algorithm to maintain the module's core functionality, switching to backup resources, switching to a default strategy that degrades the functionality in case of unexpected and unknown events, adapting to new environmental conditions, or adjusting task processing priorities, among other possibilities.

It's worth mentioning that in our approach, the event handler is not required to change the strategy for every event. Certain events may fall outside a module's ability to resolve on its own. When a module experiences an unknown or critical error that impedes its normal operation, it can notify its dependent modules about the malfunction. These dependent modules are then expected to adjust their strategies accordingly, effectively shifting the responsibility for resolving the event to a higher hierarchical level. Taking this into account, it can be seen that our pattern allows each module to play a dual role – acting both as a publisher and as a subscriber. This means that a module can react to received events while also generating new events in response to its current state. This cascading approach to event processing supports multi-level and hierarchical error management, ultimately enhancing the overall reliability and adaptability of the system. In summary, our design supports adaptation of individual components by allowing them to change their operational strategies (for example, by switching algorithms) and by reconfiguring their submodules.

An example of such hierarchical structure is introduced in Fig. 1. Here, the highest-level module functions as the main component that orchestrates the system, yet its overall functionality depends on the services provided by the lower-level modules. In this view, the downward pointing dependency arrows effectively show that while the high-level module is in a position to steer the configuration and behavior of the lower modules, it remains fundamentally dependent on them to supply the necessary functionality.

Furthermore, the pattern encourages developers to clearly identify and define critical events and to design targeted response strategies. Developers are urged to thoroughly analyze the range of potential events – both anticipated and unforeseen – that might influence system behavior. For each identified event, a corresponding handling strategy should be pedantically crafted. Such careful planning and targeted strategy development ensure that each module can gracefully adapt to changing conditions while remaining isolated from unintended side effects in other parts of the system.

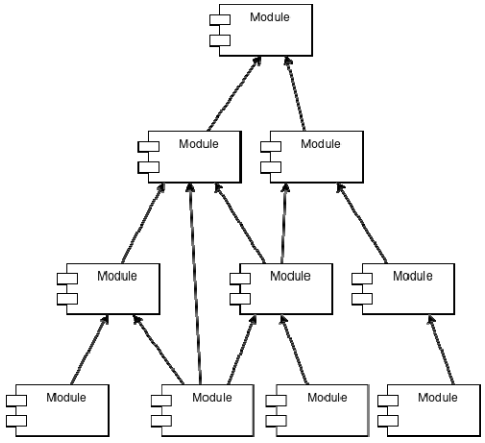

Figure 1 – An example of hierarchical module structure

The UML class diagram of the pattern, which is presented on Fig. 2, illustrates the key components and their relationships within the pattern, further is the description of every item.

– IPublisher. This interface defines the contract for any entity that is responsible for publishing events. It declares the method for event publication, ensuring that any implementing class can deliver events to the system's central coordinator.

– IEventSubscriptionService. This interface outlines the operations required for managing event subscriptions. It includes methods for subscribing and unsubscribing components, ensuring that events are delivered only to those subscribers that have expressed interest.

– IEventSubscriber. The IEventSubscriber interface specifies the method(s) a component must implement to handle incoming events. It standardizes event processing so that every subscriber can react appropriately when notified of an event.

– IStrategy. This interface defines the operational algorithm or behavior that a subscriber may adopt when processing an event. It encapsulates the logic for adapting the module's functionality in response to system or environment changes.

– IStrategySelector. The IStrategySelector interface establishes the contract for selecting an appropriate strategy based on the current context. This context may include for example information provided by an event or

DOI 10.15588/1607-3274-2026-1-9

system's state, allowing the selector to choose the best-fit strategy for the situation.

– Event. An Event represents any change, notification, or trigger that might be significant to system components. Each event is associated with a predefined type (EventType) and can carry various details (such as a timestamp, publisher identity, cause, or description) that subscribers might require for processing the event.

– EventType. Defined as an enumeration, EventType specifies the different categories of events that the system can handle. It serves as a key for mapping events to their corresponding subscribers and facilitates efficient event routing within the system.

– AnEventProducer. This abstract class encapsulates the logic for generating events. AnEventProducer aggregates an instance of the IPublisher interface, which it uses to publish events. By decoupling event generation from event distribution, it ensures that modules remain independent of the specific mechanisms used to deliver events.

– EventManager. Serving as the central coordinator of the pattern, the EventManager implements both the IPublisher and IEventSubscriptionService interfaces. It is responsible for managing subscriptions and for delivering events to the appropriate subscribers. To accomplish this, it maintains an internal mapping (subscribers map) that associates each event type with a list of interested subscribers, ensuring that events are routed correctly.

– ConcreteModule. A ConcreteModule plays the dual role of event subscriber and, optionally, event producer. It implements the IEventSubscriber interface to process incoming events and, in some cases, extends AnEventProducer to generate events as well. Upon receiving an event, the ConcreteModule may leverage a strategy selection mechanism to dynamically choose an appropriate response strategy, thereby adapting its behavior to maintain system stability.

The UML sequence diagram is illustrated on Fig. 3. This diagram conveys the dynamic interplay between the

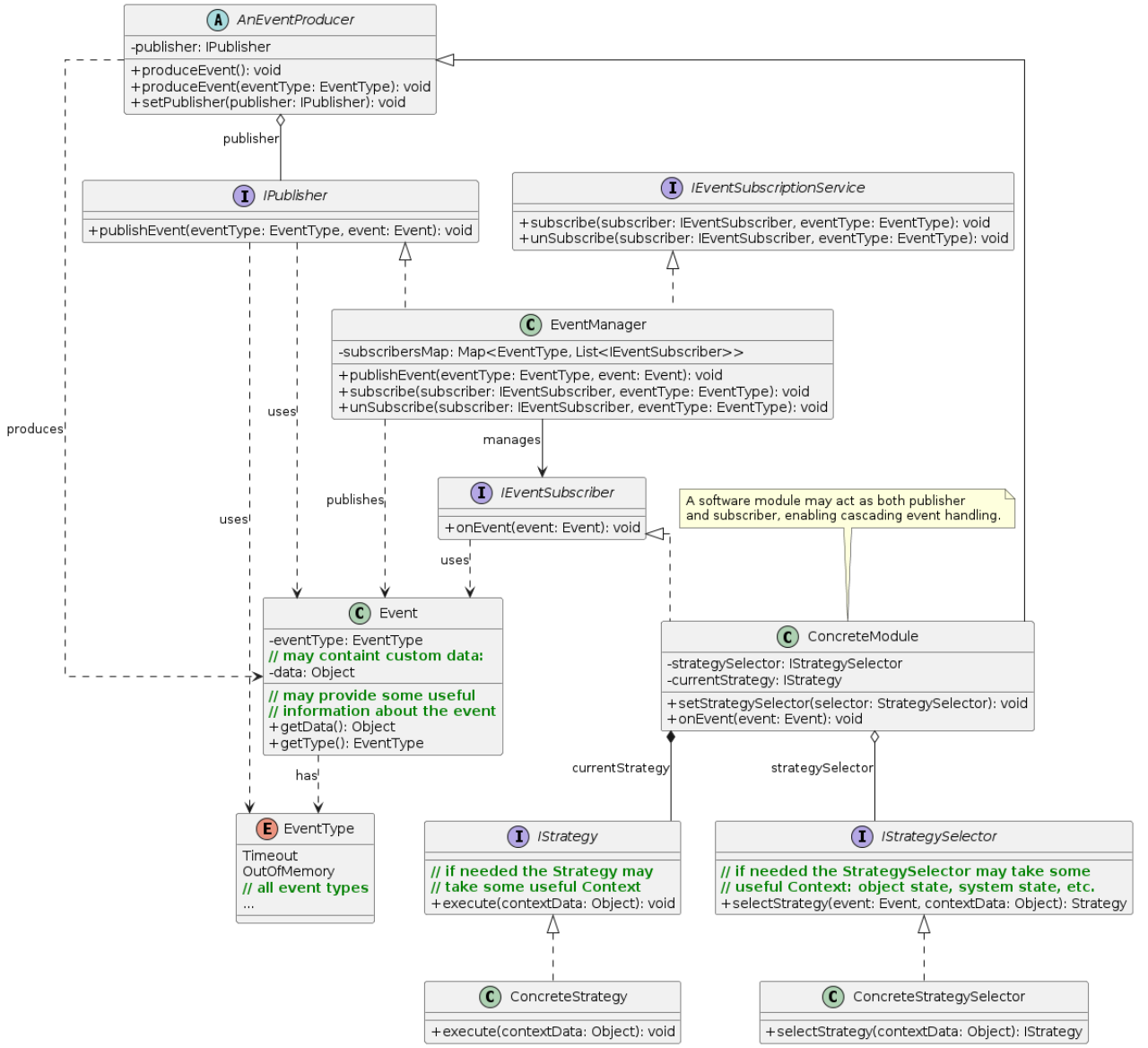

Figure 2 – UML class diagram of the proposed pattern

DOI 10.15588/1607-3274-2026-1-9

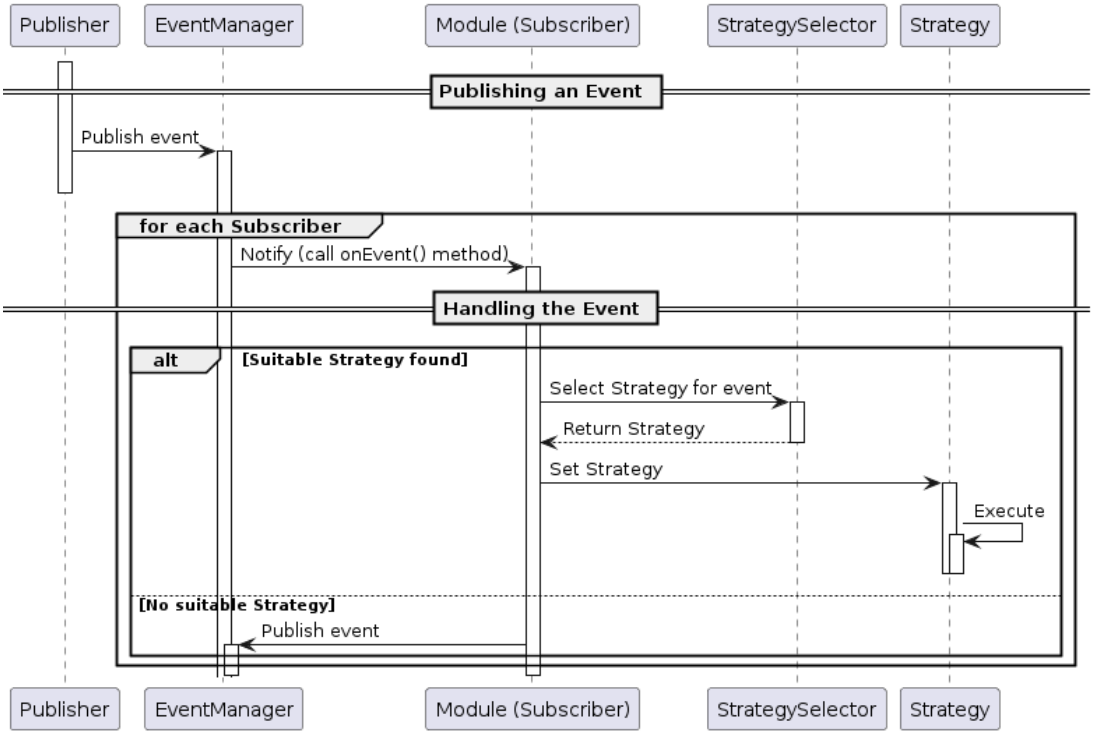

Figure 3 – UML sequence diagram of the proposed pattern

key components in our pattern, showcasing how the system adapts to events through a cascading mechanism. The diagram highlights the inherent flexibility built into the design. Upon receiving the event, each subscriber evaluates it using its StrategySelector. The sequence diagram illustrates that if a subscriber identifies an appropriate response, it will execute the corresponding strategy to adapt its behavior. However, if no suitable strategy is found, the subscriber escalates the situation by publishing a new event, effectively shifting the responsibility to higher-level components. This cascading approach not only underscores the dual role of modules – as both publishers and subscribers – but also reinforces a multi-tiered error management process that enhances the overall reliability and adaptability of the system.

In summary, our proposed pattern fulfills the described approach of providing functional stability for software systems by enabling each software component to maintain multiple strategies – represented via the IStrategy interface – and dynamically react to changes. The pattern ensures that, upon detection of relevant events, a component can select and execute the most appropriate adaptation strategy through its strategy selection interface IStrategySelector. This design not only supports flexible adaptation but also lays the architectural foundation for achieving functional stability across the system.

4 EXPERIMENTS

In our experimental evaluation, we address the challenge of designing a robust software system that can maintain functional stability under dynamic conditions. Rather than focusing on specific code implementations, our experiment demonstrates how the proposed enabling pattern influences the design of an application intended for real-life challenges, such as stable visual object tracking for autonomous platforms.

The solution under design is expected to detect, track, and recover from tracking failures in real time. Initially, the system analyzes the video stream to detect an object based on predefined criteria. Upon detection, an event is generated that triggers the transition from an object detection mode to an object tracking mode. When the target moves an appropriate event occurs and the corresponding module uses a strategy selection mechanism to determine whether to adjust the camera orientation or reposition the platform using its wheels.

The experimental design is considered to support several scenarios.

– Initial detection and transition to tracking. To start the tracking process, the system must first detect the target object. In this initial step, the system employs “find” and “detect” strategies by exploring the camera view and, if necessary, changing its physical location. Once the target is detected, a detection event is published and routed to all interested parties, initiating the tracking process.

– Dynamic tracking with adaptive strategies. As the target moves, a series of events reflecting its motion are generated. The strategy selector evaluates these events and dynamically chooses the appropriate response – adjusting the camera for minor movements or driving the wheels for larger positional changes. This scenario

DOI 10.15588/1607-3274-2026-1-9

demonstrates the pattern's ability to maintain functional stability by adapting to ongoing changes.

– Target loss and recovery. In situations where the target temporarily leaves the field of view or becomes occluded, the system generates a "target lost" event. This event triggers a recovery procedure, wherein the system searches for the target based on the last known tracking data

The pattern's capability to dynamically switch between different strategies ensures that the system can adapt to environmental variability and unexpected disturbances, thereby achieving reliable performance under a wide range of conditions. To present our experimental design we have developed a series of simplified UML class diagrams that detail the architecture of the experimental system. These diagrams illustrate the core infrastructure, along with the domain-specific modules that implement dynamic adaptation. Note that some core components (e.g., EventManager) and interfaces are not present in UML diagrams for better readability, but they are still essential for the design.

Fig. 4 presents the components related to the camera functionality. The CameraModule, responsible for capturing frames and reorienting the camera, extends the common AnEventProducer and implements the IEventSubscriber interface. It interacts with a dedicated CameraStrategySelector that dynamically selects between the CameraTrackingStrategy and CameraSearchStrategy based on incoming events.

The Movement Domain diagram presented in Fig. 5 details the architecture for platform repositioning. The MovementModule, which manages the physical adjustments of the system, similarly extends AnEventProducer and implements IEventSubscriber. It utilizes a MovementStrategySelector to choose between strategies such as WheelTrackingStrategy and WheelSearchStrategy. This enables responsive adjustments to the platform's position in reaction to target movement or loss.

In the diagram on Fig. 6, the Image Analysis Domain is depicted. The ImageAnalysisModule, tasked with processing captured images and analyzing target characteristics, also inherits from AnEventProducer and implements IEventSubscriber. It employs an ImageAnalysisStrategySelector to dynamically select among strategies such as FindObjectStrategy, TrackObjectStrategy, and RecoverFindStrategy. This modular approach facilitates robust analysis and quick recovery when tracking is disrupted.

By focusing on architectural design rather than implementation specifics, our experiment demonstrates that the proposed enabling pattern can serve as a robust architectural backbone for adaptive systems. The pattern's ability to support both dynamic strategy selection and component reconfiguration highlights its potential to enhance system functional stability in real-world applications.

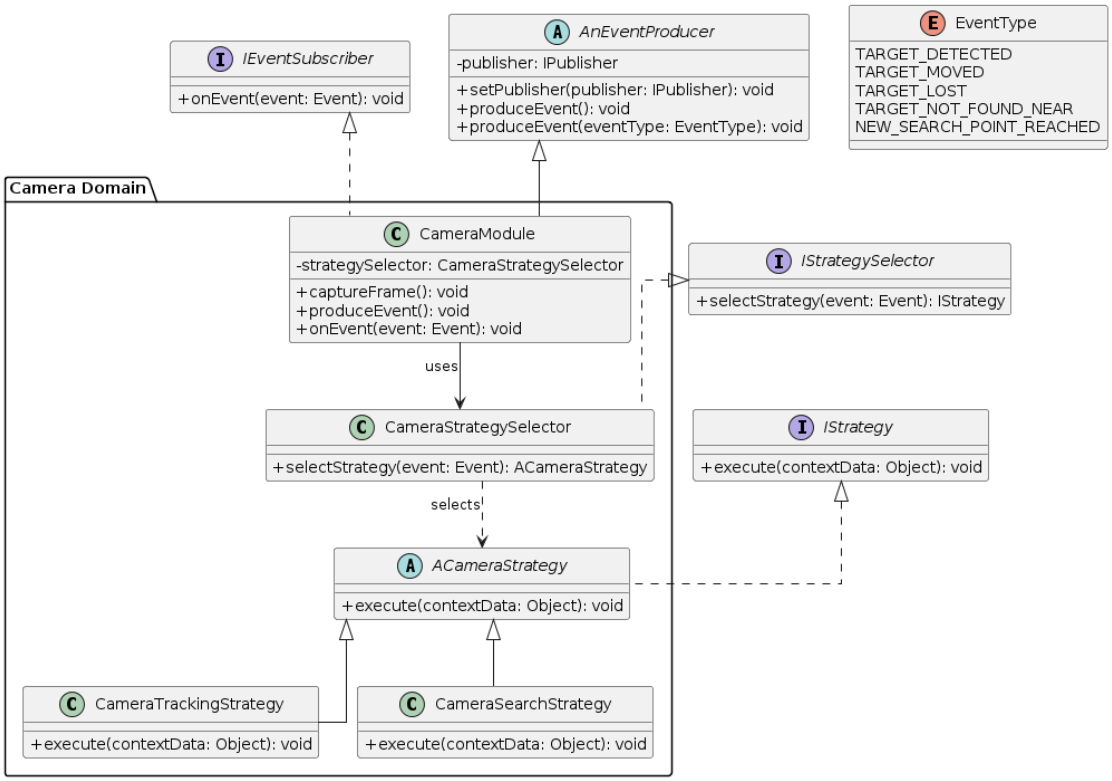

Figure 4 – The UML class diagram for Camera domain

DOI 10.15588/1607-3274-2026-1-9

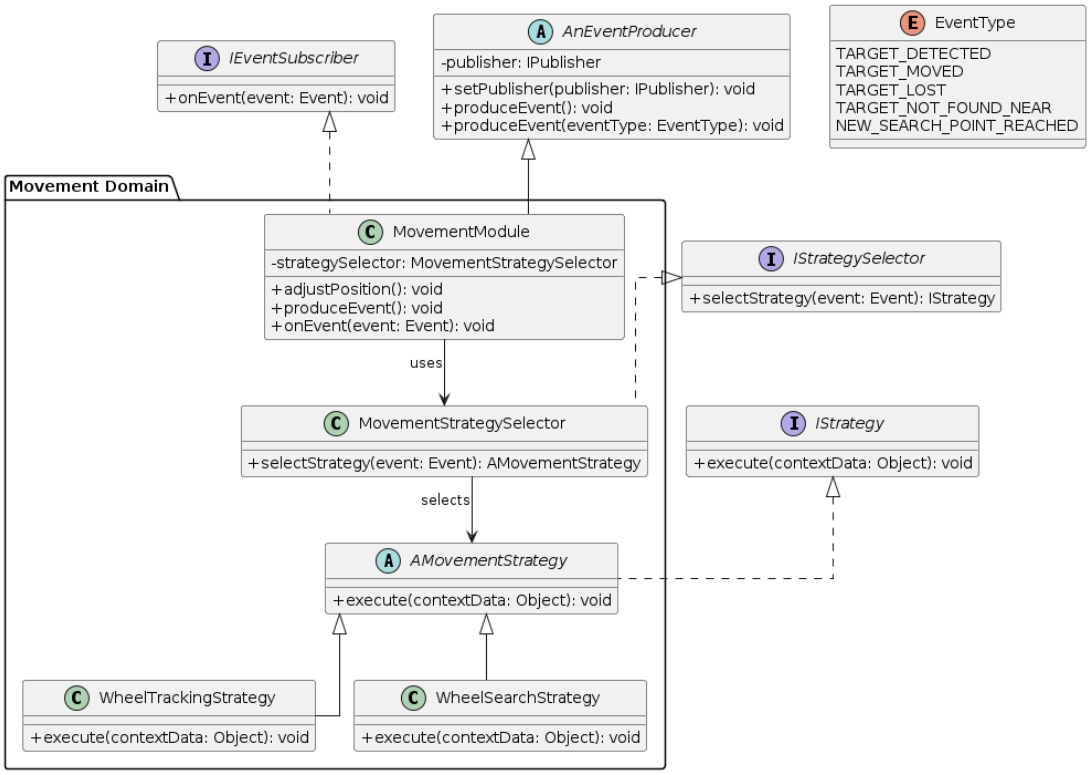

Figure 5 – The UML class diagram for Movement domain

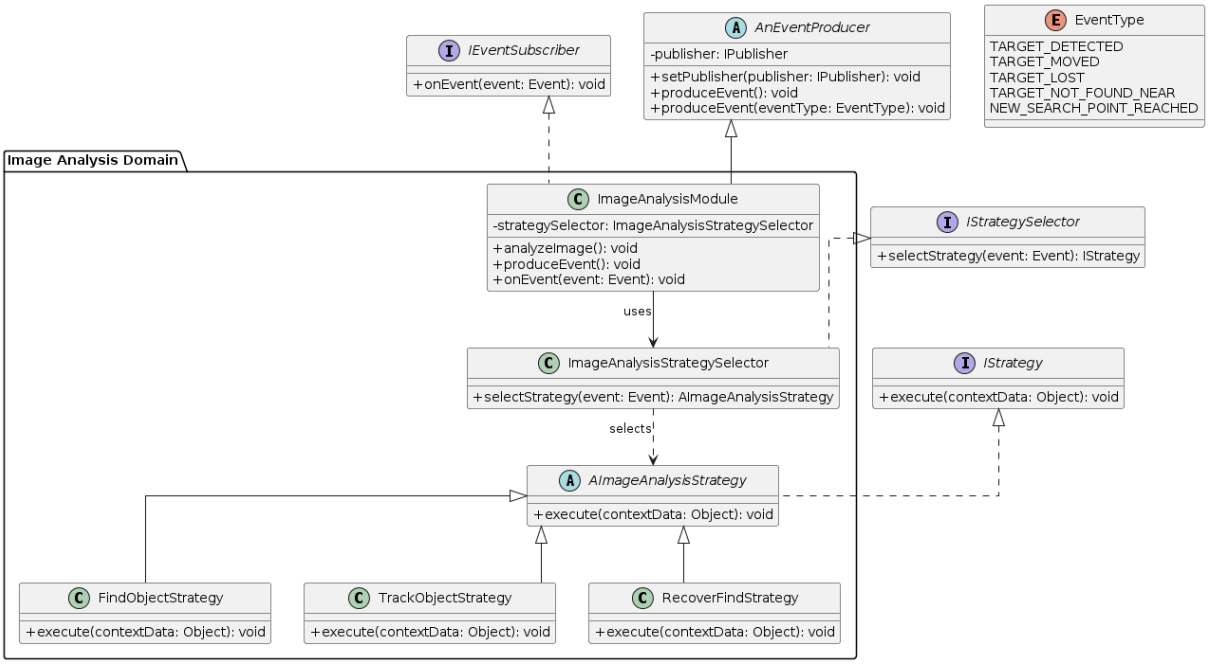

Figure 6 – The UML class diagram for Image Analysis domain

5 RESULTS

The proposed design pattern was evaluated from the perspective of structural quality, modularity, and readiness for adaptation in dynamic software systems. Rather than benchmarking low-level algorithmic performance, the assessment focuses on design-level metrics and scenario-based maintainability analysis.

To quantify structural quality, object-oriented design metrics were calculated based on the UML architecture presented in the Experimental section. Their values are shown in Table 1.

DOI 10.15588/1607-3274-2026-1-9

Table 1 – Object-oriented metrics for the proposed design

Metric	Value	Interpretation
CBO (Coupling Between Objects) [22]	4	Low coupling; components interact solely via event types
LCOM (Lack of Cohesion in Methods, LCOM4) [22]	0.19	High cohesion; modules encapsulate tightly related functionality
RFC (Response for Class) [22]	36	Moderate response surface; limited and testable method complexity
DIT (Depth of Inheritance Tree) [22]	3	Moderate abstraction depth due to use of interfaces and base classes

These metrics confirm the pattern's architectural strength: decoupled modules, focused responsibilities, and extensibility through clearly defined interfaces.

Beyond metric-based evaluation, the design was assessed according to established SOLID [23] principles. In general, our design satisfies the SOLID principles:

– it demonstrates high cohesion and a clear separation of responsibilities (SRP);

– it is open to extension through subclassing while remaining stable (OCP);

– it supports substitutability of components without loss of functionality (LSP);

– it employs client-specific interfaces to prevent unnecessary dependencies (ISP);

– it inverts dependencies so that high-level modules rely on abstractions (DIP).

Qualitatively, the architecture shows:

– high modularity – by decoupling modules via events, each component can be developed and maintained independently, which facilitates scalability and easier integration of new features;

– dynamic adaptability – the use of dedicated strategy selectors within each domain allows the system to respond adaptively to dynamic conditions, thereby enhancing functional stability;

– maintainability – clear separation of core and domain-specific components reduces complexity and makes the system easier to understand and maintain.

These qualities collectively validate the robustness, adaptability, and maintainability of our proposed pattern.

Although the study does not rely on a specific implementation, an approximate estimation of runtime overhead was performed based on architectural assumptions. The event loop involves four stages: event creation, dispatch, strategy selection, and strategy execution. Table 2 outlines estimated average durations for each step, assuming single-process deployment with in-memory event dispatch.

Table 2 – Estimated runtime overhead per event

Operation	Estimated duration (μs)
Event creation and enqueueing	5–10
Dispatch to subscribers	10–15
Strategy selection	3–5
Strategy switch and execution	4–6
Total estimated per event	22–36

Given that typical processing intervals in control or video-tracking applications range from 20 to 50 milliseconds, even handling multiple events per frame results in overhead well under 1–2% of total cycle time. These figures validate that the design is lightweight and suitable for real-time reactive systems.

To further analyse maintainability, several common evolution scenarios were considered. For each, we assessed the required changes and the relative complexity involved in extending the system using the proposed pattern. Table 3 presents these findings.

Table 3 – Change scenarios and their impact on the proposed design

Scenario	Required changes	Relative impact
Adding a new event type	Define a new event type, register subscribers, implement matching strategy	Low
Replacing tracking algorithm	Add new strategy and register it; no changes to other modules	Low
Adding a notification module for alerting	Implement a subscriber, subscribe to target events	Low
Introducing criticality-based prioritisation of events	Extend event metadata, modify dispatch logic or event manager policies	Medium
Changing event routing policy (e.g., filtering, batching)	Modify event manager internals or insert pre-processors	Medium
Adding a dynamic strategy selector that uses ML models	Extend selector logic, integrate model loading, handle decision fallback	Medium

The analysis shows that typical functional extensions involve localized changes and benefit from the modular structure. Scenarios that modify global coordination logic (e.g., routing policies) introduce higher complexity, which is expected due to their system-wide implications.

6 DISCUSSION

Our evaluation confirms that the proposed software design pattern successfully addresses the stated problem, described at the beginning of the article. Specifically, the pattern meets the following requirements:

– enabling adaptability – each component dynamically selects a strategy based on the current situation and internal state;

– handling expected and unforeseen situations – the inclusion of fallback strategies ensures that even when unexpected conditions arise, the system maintains its stability;

– supporting extensibility and scalability – the modular architecture, with clearly decoupled components that communicate solely via event types, facilitates maintenance, extension, and scaling.

These points are supported by our design-level metrics and qualitative assessments, which together demonstrate that the pattern provides a robust architectural backbone for enabling functional stability.

Our proposed pattern distinguishes itself from reviewed solutions in several ways.

DOI 10.15588/1607-3274-2026-1-9

– Integrated Adaptability Through Strategy Selection. Whereas traditional patterns like Observer and Strategy provide individual benefits – event notification and algorithmic interchangeability – our pattern fuses these concepts together. By dynamically selecting among domain-specific strategies, our pattern not only reacts to events but does so in a way that supports continuous functional stability.

– Decoupling and Modularity. Several approaches in the literature [13–16, 19, 20] emphasize reconfiguration or centralized fault handling. Our pattern achieves adaptability through a decoupled, event-driven architecture, ensuring that components interact solely through standardized event types. This decoupling minimizes interdependencies and supports a highly modular design. Such modularity enables incremental extension and easier maintenance.

– Architectural Focus on Functional Stability. While many studies address adaptiveness [13–16, 20] or self-healing [19] – typically measured in terms of recovery time or system throughput – our work explicitly targets functional stability. Our pattern ensures that, despite disturbances, the output state of each system function remains within an acceptable stable range after a transient period. This focus on functional stability fills a gap identified in the literature where adaptive techniques are rarely evaluated against the criteria of long-term functional stability.

– Combination of Strategy and Reconfiguration Approaches. Some literature [13–16] contrasts strategy-based adaptation with dynamic reconfiguration. Our pattern uniquely combines these perspectives: while components may change their operational strategies at runtime, these strategy changes can incorporate subcomponent reconfigurations as needed. This hybrid approach not only provides flexibility but also ensures that the system maintains its essential functionality even as underlying configurations evolve.

Despite its strengths, our design is not without challenges. The event-driven architecture introduces inherent complexity in managing event flows, which can complicate debugging and performance analysis. Additionally, the overhead of dynamic strategy selection may impact performance under extremely high event loads. These trade-offs are considered acceptable given the substantial gains in adaptability and modularity; however, they warrant further empirical investigation to optimize system performance in large-scale deployments.

The proposed pattern is particularly well-suited for applications requiring high adaptability, such as autonomous systems, real-time monitoring, and fault-tolerant computing environments. Its modular nature enables integration with existing systems and supports iterative enhancement.

CONCLUSIONS

This work addresses the scientific problem of ensuring functional stability in dynamic, resource-constrained software systems by developing an enabling design pattern. Our proposed pattern provides an architectural backbone that decouples system components through event-driven interactions and supports dynamic strategy selection.

The scientific novelty of this work is twofold. Firstly, our novel pattern supports dynamic adaptation by allowing each module to select an appropriate strategy based on the current operational context. This novel pattern is firstly obtained in our study and is substantiated by quantitative design-level metrics – such as low coupling and inferred high cohesion – which demonstrate the robustness and maintainability of the design. Secondly, the functional stability received further development by bridging the gap in the software architecture viewpoint. In this regard, our approach ensures that system functions remain within predetermined stable states despite disturbances, effectively uniting theoretical concepts with practical architectural design.

The practical significance of the proposed pattern is that it is applicable to a wide range of adaptive systems, including autonomous platforms, real-time monitoring, and fault-tolerant computing environments. Its modularity and clear separation of concerns facilitate easier integration, maintenance, and future extension. As a recommendation, practitioners can adopt this pattern as a foundational element in designing resilient architectures where adaptability and stability are critical.

Prospects for further research are to focus on advancing the pattern by incorporating additional features, such as prioritization of events and multi-threading support, which could further enhance the pattern's scalability and performance in high-demand environments. In parallel, the development of specialized tools and methodologies for streamlined pattern testing, debugging, and integration is highly recommended.

DECLARATIONS

Conflict of interest: The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship, or otherwise, that could affect the research and its results presented in this paper.

Authors' contributions: Oleksii Bychkov: pattern conceptualization, writing – review & editing; Mykola Moroz: searching and reviewing the literature, pattern design and evaluation, writing – original draft.

Data availability: The manuscript has no associated data.

Software availability: The manuscript has no associated software.

Use of artificial intelligence tools: The authors used artificial intelligence technologies in creating the submitted work: X-GPT-4 model was used in order to perform style, grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

DOI 10.15588/1607-3274-2026-1-9

REFERENCES

1. Barabash O. V., Sobchuk V. V., Musienko A. P. et al. System analysis and method of ensuring functional sustainability of the information system of a critical infrastructure object, *System analysis and artificial intelligence*. Cham, Springer, 2023, Vol. 1107, pp. 177–192. DOI: 10.1007/978-3-031-37450-0_11
2. Barabash O. V., Svynchuk O. V., Salanda I. P. et al. Ensuring the functional stability of the information system of the power plant on the basis of monitoring the parameters of the working condition of computer devices, *Advanced Information Systems*, 2024, Vol. 8, № 2, pp. 107–117. DOI: 10.20998/2522-9052.2024.2.12
3. Kalashnyk G. A., Kalashnyk-Rybalko M. A. Strategy for provision of the functional stability of integrated complexes of modern and advanced aircraft onboard equipment, *Perspective trajectory of scientific research in technical sciences*. Riga, Baltija Publishing, 2021, Section 10, pp. 186–202. DOI: 10.30525/978-9934-26-085-8-10
4. Kalashnyk G. A., Kalashnyk-Rybalko M. A. Methodology for ensuring the functional stability of aircraft integrated modular avionics complex, *Science and Technology of the Air Force of Ukraine*, 2024, № 4 (53), pp. 30–40. DOI: 10.30748/nitps.2023.53.04
5. Firsov S. N., Pishchukhina O. A. Intelligent support of multilevel functional stability of control and navigation systems, *Radio Electronics, Computer Science, Control*, 2018, № 2, pp. 177–183. DOI: 10.15588/1607-3274-2018-2-20
6. Barabash O. V., Tverdenko H. M., Sobchuk V. V.et al. The assessment of the quality of functional stability of the automated control system with hierarchic structure, *Proceedings of the 2nd International Conference on System Analysis & Intelligent Computing*, 2020, pp. 1–4. DOI: 10.1109/SAIC51296.2020.9239122
7. Sobchuk V. V., Barabash O. V., Musienko A. P. et al. Analysis of the main approaches and stages for providing the properties of the functional stability of the information systems of the enterprise, *Sciences of Europe,* 2019, Vol. 1, № 42, pp. 41–44.
8. Barabash O. V., Sobchuk V. V., Musienko A. P. et al. System analysis and method of ensuring functional sustainability of the information system of a critical infrastructure object, *System Analysis and Artificial Intelligence.* Cham, Springer, 2023, Section 11, pp. 177–192. DOI: 10.1007/978-3-031-37450-0_11
9. Salehie M., Tahvildari L. Self-adaptive software, *ACM Transactions on Autonomous and Adaptive Systems,* 2009, Vol. 4, № 2, pp. 1–42. DOI: 10.1145/1516533.1516538
10. IEEE Standard Glossary of Software Engineering Terminology : IEEE Std 610.12-1990. [Effective from 1990-12-31]. New York, IEEE, 1990, 84 p.
11. Committee on National Security Systems Glossary : CNSSI 4009. [Effective from 2022-03]. Fort Meade, CNSS, 2022, 252 p.
12. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston, Addison-Wesley, 1995, 395 p.
13. Ramirez A. J. Design patterns for developing dynamically adaptive systems : thesis … master of science in computer science. East Lansing, Michigan State University, 2008, 244 p. DOI: 10.25335/ttfn-qx40
14. Ramirez A. J., Cheng B. H. C. Design patterns for developing dynamically adaptive systems, *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems,* 2010, pp. 49–58. DOI: 10.1145/1808984.1808990
15. Mannava V., Ramesh T. A novel event based autonomic design pattern for management of webservices, *Advances in Computing and Information,* 2011, pp. 142–151. DOI: 10.1007/978-3-642-22555-0_16
16. Mannava V., Ramesh T. A novel adaptive re-configuration compliance design pattern for autonomic computing systems, *Procedia Engineering*, 2012, Vol. 30, pp. 1129–1137. DOI: 10.1016/j.proeng.2012.01.972
17. Gomaa H., Hussein M. Software reconfiguration patterns for dynamic evolution of software architectures, *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, 2004, pp. 79–88. DOI: 10.1109/wicsa.2004.1310692
18. Gomaa H., Albassam E. Run-time software architectural models for adaptation, recovery and evolution, *Proceedings of the MODELS 2017 Satellite Event*, 2017, pp. 193–200.
19. Nguyen T. A., Aiello M., Yonezawa T. et al. A self-healing framework for online sensor data, *2015 IEEE International Conference on Autonomic Computing (ICAC),* 2015, pp. 295–300. DOI: 10.1109/icac.2015.61
20. An architectural blueprint for autonomic computing : white paper, IBM Corporation. Armonk, NY, 2006, 37 p.
21. Schmidt D. C., Buschmann F., Henney K. Pattern-oriented software architecture. Hoboken, NJ, John Wiley & Sons, 2007, pp. 28–29.
22. Chidamber S. R., Kemerer C. F. A metrics suite for object oriented design, *IEEE Transactions on Software Engineering,* 1994, Vol. 20, № 6, pp. 476–493. DOI: 10.1109/32.295895
23. Martin R. Clean architecture: A craftsman's guide to software structure and design. Boston, Prentice Hall, 2018, 420 p.

Received 17.03.2025.
Accepted 14.01.2026.
Published 27.03.2026.

УДК 004.41

ПАТЕРН ПРОЄКТУВАННЯ ДЛЯ ЗАБЕЗПЕЧЕННЯ ФУНКЦІОНАЛЬНОЇ СТІЙКОСТІ В ПРОГРАМНИХ СИСТЕМАХ

Бичков О. С. – д-р техн. наук, професор, завідувач кафедри програмних систем та технологій Київського національного університету імені Тараса Шевченка, Київ, Україна. ROR: https://ror.org/02aaqv166. ORCID: https://orcid.org/0000-0002-9378-9535.

Мороз М. В. – аспірант кафедри програмних систем та технологій Київського національного університету імені Тараса Шевченка, Київ, Україна. ROR: https://ror.org/02aaqv166. ORCID: https://orcid.org/0000-0001-6953-683X.

АНОТАЦІЯ

Актуальність. Сучасні програмні системи працюють у динамічних та жорстких умовах, де внутрішні та зовнішні збої, непередбачувані збурення, прямі атаки та обмеження ресурсів ускладнюють стабільне надання основних функцій. У таких умовах забезпечення функціональної стійкості, коли якість кожної функції системи залишається в межах заздалегідь визначеного стабільного діапазону, незважаючи на збої чи аномалії середовища, є критично важливим, особливо для систем, де безпека та висока доступність мають першорядне значення.

Мета роботи – розробка та обґрунтування патерну проєктування, який забезпечує архітектурну основу для досягнення функціональної стійкості в програмних системах. Головний акцент робиться на наданні гнучкого рішення, яке сприяє динамічній адаптації при збереженні надійної роботи системи.

Метод. Запропоновано новий патерн проєктування, який поєднує можливості динамічного вибору стратегії з перевагами слабкого зв'язку між компонентами, який забезпечується моделлю на основі подій. Цей патерн роз'єднує компоненти

DOI 10.15588/1607-3274-2026-1-9

системи, забезпечуючи взаємодію виключно через стандартизовані типи подій, та дозволяє кожному модулю обирати відповідну стратегію адаптації на основі його поточного контексту. Описаний патерн був використаний для побудови дизайну прикладу, спрямованого на впровадження функціональності стабільного трекінгу об'єктів для автономних квадроплатформ. Запропонований дизайн оцінювався за допомогою дизайн-метрик та якісних порівнянь з існуючими адаптивними підходами.

Результати. Проведений аналіз показав, що патерн забезпечує значну модульність та адаптивність. Ключові об'єктно-орієнтовані метрики свідчать про мінімальну взаємозалежність між модулями та чіткий розподіл обов'язків. Запропонований дизайн демонструє, що патерн підтримує динамічну адаптацію поведінки за рахунок гнучкого вибору стратегії і слугує засобом забезпечення функціональної стійкості, створюючи надійну архітектурну основу для програмних систем.

Висновки. Наукова новизна цієї роботи є подвійною: по-перше, у дослідженні отримано новий патерн, що забезпечує динамічну адаптацію через контекстно-орієнтований вибір стратегії; по-друге, функціональна стійкість отримала подальший розвиток в області програмної архітектури шляхом заповнення прогалини між теоретичними концепціями стійкості та практичним проєктуванням систем. Запропонований патерн пропонує надійне, масштабоване та підтримуване архітектурне рішення з вагомими практичними наслідками для розробки адаптивних, стійких програмних систем.

КЛЮЧОВІ СЛОВА: патерни проєктування програмного забезпечення, функціональна стійкість, обробка подій, адаптивна поведінка, автономні системи.

ЛІТЕРАТУРА

1. System analysis and method of ensuring functional sustainability of the information system of a critical infrastructure object / [O. V. Barabash, V. V. Sobchuk, A. P. Musienko et al.] // System analysis and artificial intelligence. – Cham : Springer, 2023. – Vol. 1107. – P. 177–192. DOI: 10.1007/978-3-031-37450-0_11
2. Ensuring the functional stability of the information system of the power plant on the basis of monitoring the parameters of the working condition of computer devices / [O. V. Barabash, O. V. Svynchuk, I. P. Salanda et al.] // Advanced Information Systems. – 2024. – Vol. 8, № 2. – P. 107–117. DOI: 10.20998/2522-9052.2024.2.12
3. Kalashnyk G. A. Strategy for provision of the functional stability of integrated complexes of modern and advanced aircraft onboard equipment / G. A. Kalashnyk, M. A. Kalashnyk-Rybalko // Perspective trajectory of scientific research in technical sciences. – Riga : Baltija Publishing, 2021. – Section 10. – P. 186–202. DOI: 10.30525/978-9934-26-085-8-10
4. Kalashnyk G. A. Methodology for ensuring the functional stability of aircraft integrated modular avionics complex / G. A. Kalashnyk, M. A. Kalashnyk-Rybalko // Science and Technology of the Air Force of Ukraine. – 2024. – № 4 (53). – P. 30–40. DOI: 10.30748/nitps.2023.53.04
5. Firsov S. N. Intelligent support of multilevel functional stability of control and navigation systems / S. N. Firsov, O. A. Pishchukhina // Radio Electronics, Computer Science, Control. – 2018. – № 2. – P. 177–183. DOI: 10.15588/1607-3274-2018-2-20
6. The assessment of the quality of functional stability of the automated control system with hierarchic structure / [O. V. Barabash, H. M. Tverdenko, V. V. Sobchuk et al.] // Proceedings of the 2nd International Conference on System Analysis & Intelligent Computing. – 2020. – P. 1–4. DOI: 10.1109/SAIC51296.2020.9239122
7. Analysis of the main approaches and stages for providing the properties of the functional stability of the information systems of the enterprise / [V. V. Sobchuk, O. V. Barabash, A. P. Musienko et al.] // Sciences of Europe. – 2019. – Vol. 1, № 42. – P. 41–44.
8. System analysis and method of ensuring functional sustainability of the information system of a critical infrastructure object / [O. V. Barabash, V. V. Sobchuk, A. P. Musienko et al.] // System Analysis and Artificial Intelligence. – Cham : Springer, 2023. – Section 11. – P. 177–192. DOI: 10.1007/978-3-031-37450-0_11
9. Salehie M. Self-adaptive software / M. Salehie, L. Tahvildari // ACM Transactions on Autonomous and Adaptive Systems. – 2009. – Vol. 4, № 2. – P. 1–42. DOI: 10.1145/1516533.1516538
10. IEEE Standard Glossary of Software Engineering Terminology : IEEE Std 610.12-1990. – [Effective from 1990-12-31]. – New York : IEEE, 1990. – 84 p.
11. Committee on National Security Systems Glossary : CNSSI 4009. – [Effective from 2022-03]. – Fort Meade : CNSS, 2022. – 252 p.
12. Design Patterns: Elements of Reusable Object-Oriented Software / [E. Gamma, R. Helm, R. Johnson, J. Vlissides]. – Boston : Addison-Wesley, 1995. – 395 p.
13. Ramirez A. J. Design patterns for developing dynamically adaptive systems : thesis … master of science in computer science / Andres J. Ramirez – East Lansing, Michigan State University, 2008. – 244 p. DOI: 10.25335/ttfn-qx40
14. Ramirez A. J. Design patterns for developing dynamically adaptive systems / A. J. Ramirez, B. H. C. Cheng // Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. – 2010. – P. 49–58. DOI: 10.1145/1808984.1808990
15. Mannava V. A novel event based autonomic design pattern for management of webservices / V. Mannava, T. Ramesh // Advances in Computing and Information. – 2011. – P. 142–151. DOI: 10.1007/978-3-642-22555-0_16
16. Mannava V. A novel adaptive re-configuration compliance design pattern for autonomic computing systems / V. Mannava, T. Ramesh // Procedia Engineering. – 2012. – Vol. 30. – P. 1129–1137. DOI: 10.1016/j.proeng.2012.01.972
17. Gomaa H. Software reconfiguration patterns for dynamic evolution of software architectures / H. Gomaa, M. Hussein // Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004). – 2004. – P. 79–88. DOI: 10.1109/wicsa.2004.1310692
18. Gomaa H. Run-time software architectural models for adaptation, recovery and evolution / H. Gomaa, E. Albassam // Proceedings of the MODELS 2017 Satellite Event. – 2017. – P. 193–200.
19. A self-healing framework for online sensor data / [T. A. Nguyen, M. Aiello, T. Yonezawa et al.] // 2015 IEEE International Conference on Autonomic Computing (ICAC). – 2015. – P. 295–300. DOI: 10.1109/icac.2015.61
20. An architectural blueprint for autonomic computing : white paper / IBM Corporation. – Armonk, NY, 2006. – 37 p.
21. Schmidt D. C. Pattern-oriented software architecture / D. C. Schmidt, F. Buschmann, K. Henney. – Hoboken, NJ : John Wiley & Sons, 2007. – P. 28–29.
22. Chidamber S. R. A metrics suite for object oriented design / S. R. Chidamber, C. F. Kemerer // IEEE Transactions on Software Engineering. – 1994. – Vol. 20, № 6. – P. 476–493. DOI: 10.1109/32.295895
23. Martin R. Clean architecture: A craftsman's guide to software structure and design / R. Martin. – Boston : Prentice Hall, 2018. – 420 p.

DOI 10.15588/1607-3274-2026-1-9