Term Paper

Strategy Pattern

as part of the lecture **Advanced Software Engineering**

of the **Computer Science** study program at the **Baden-Württemberg Cooperative State University Stuttgart**

from

Ognjen Jovanovic (Matr.-Nr.: 5635633)

Simon Chasi (Matr.-Nr.: 2695170)

Course TINF23CE

Table of Contents

List of Figures

1 Introduction

"The essence of strategy is choosing what not to do." — Michael E. Porter

Fundamentally, strategy is the practice of making deliberate decisions. Effective strategy is defined not only by the selection of actions aligned with a desired outcome, but also by the intentional exclusion of alternate courses of action. This exclusionary notion, succinctly put in Porter's observation, reveals a key insight into what strategy aims to facilitate: navigating complexity.

In the face of today's rapid digitalization and technological progress, modern software systems are required to provide more than isolated functionality. Systems are expected to exhibit dynamic behaviour, adapting to context and extending or modifying functionality without undermining the integrity of the system as a whole. In software engineering, this demand for interchangeability increases the complexity posed by each problem: determining *how* to approach a problem has become equally as important as finding a solution.

It is in traversing this complexity that the Strategy Pattern, as formulated by the Gang of Four (GoF), demonstrates its relevance. The pattern outlines a way to make competing or distinct implementations of an algorithm or functionality interchangeable, thereby rendering a system flexible in the face of required adaptability.

Importantly, the Strategy Pattern is intertwined with central and widely held software engineering principles, such as the Open/Closed Principle and the Dependency Inversion Principle of SOLID, promoting modularity, loose coupling of components, and long-term maintainability.

As a result, the pattern remains highly relevant today, with its influence evident in a range of contemporary contexts, including dependency-injection frameworks, flexible payment processing systems, and modular machine-learning pipelines.

This paper aims to showcase the Strategy Pattern from both a theoretical and practical perspective. It will begin by examining the conceptual and historical foundations of the pattern, situating it within the broader GoF design pattern framework, as well as outlining its contemporary relevance. This will be followed by a practical demonstration in the form of a console-based Java application that implements the Strategy Pattern in the domain of maze traversal. The application allows the user to select among different maze-solving algorithms at runtime, illustrating how the pattern enables dynamic selection of behaviour while maintaining a clear separation between the problem domain and the algorithms that operate within it.

2 Background & Literature Review

2.1 Justification of the Primary Source

Design Patterns: Elements of Reusable Object-Oriented Software was originally published in 1994 by the Gang of Four (GoF) – Gamma, Helm, Johnson, and Vlissides – and is widely regarded as the authoritative source on software design patterns.

Its continued relevance today lies in its balance between theoretical grounding and practical applicability: the book not only introduces a systematic vocabulary for recurring design challenges but also provides a language-agnostic framework for implementing design patterns in object-oriented software. As a result, the patterns described in the book continue to support the development of modular, maintainable, and adaptable systems in contemporary software engineering.

Because the Strategy Pattern was originally introduced, defined, and exemplified in this work, it is both theoretically sound and practically appropriate to adopt the GoF description as the primary reference in this paper. Furthermore, the structured format provided by the GoF text offers a clear framework for the maze-traversal implementation presented later, illustrating the alignment between conceptual analysis and concrete demonstration.

With this foundation established, the following subsection introduces the framework used to describe design patterns by the GoF.

2.2 Establishing Key Terms

Design patterns provide reusable solutions for recurring problems in object-oriented software design. They consolidate proven approaches to abstract problems structuring the collaboration and communication between objects and classes, making them both broadly applicable and suitable to the requirements of specific contexts [1, p. 3].

In this way, design patterns formalise the mechanism that often separates experienced developers from beginners: the ability to draw on established solutions to similar design challenges encountered in the past [1, p. 2]. A pattern is typically defined by four elements—its name and classification, the problem and context in which it should be applied, a description of how it solves that problem, and the consequences and trade-offs associated with its use [1, p. 3].

Each pattern in the GoF catalogue is described in a structured format, following a template including its name, intent, known aliases, motivation, applicability, structural diagram, key participants, collaborations, sample code, known uses, and related patterns [1, p. 6-7]. This structured representation supports consistent communication and shared understanding of design patterns, which in turn promotes their adoption across diverse problem domains.

2.3 The Strategy Pattern

2.3.1 Intent

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” [1, p. 315]

This is how the Strategy Pattern, sometimes referred to as Policy [1, p. 315], is canonically defined in the GoF text. It enables the selection of different yet substitutable algorithms as solutions to a given problem while keeping the client code unchanged, allowing behaviour to adapt dynamically.

2.3.2 Motivation

Clients often depend on functionality whose concrete implementation may vary, with multiple algorithms capable of solving the same problem. To maintain modularity and loose coupling

between clients and these varying implementations, it is preferable to insulate the client from managing the selection and coordination of specific algorithms [1, p. 315]. This separation of concerns supports flexibility, ease of modification, and improved testability.

The Strategy Pattern achieves insulation of the client code by introducing a common interface for a family of algorithms. Following the Strategy Pattern, clients depend exclusively on this abstraction, referred to as a strategy. In this way, a client can access the various implementations of the strategy, allowing for dynamic runtime behaviour, while being insulated from the complexity of managing multiple implementations. [1, p. 316]

2.3.3 Applicability

The applicability of the Strategy Pattern is best understood by examining relationships between classes, particularly where behavioural variation can be isolated within a common abstraction.

Following this logic, the pattern is appropriate when multiple classes differ only in their behaviour but fulfil the same conceptual role. In such cases, the differing behaviours can be extracted and represented as separate implementations of a core strategy, sharing a common interface. This promotes modularity and allows behaviour to vary independently of the code that uses it. [1, p. 316]

Strategy is also useful when the details of an algorithm's implementation should be concealed from its clients. A strategy abstraction enforces these knowledge boundaries by encapsulating implementation details within the concrete strategies themselves. [1, p. 316]

Furthermore, the pattern is suitable when multiple versions of the same algorithm are required to meet distinct functional or non-functional requirements. Each version can be realised as a separate strategy, thereby avoiding duplication in client logic and enabling interchangeability. [1, p. 316]

From a programmatic perspective, Strategy can be employed as a principled alternative to large conditional statements and branching logic. Behaviour that would otherwise be controlled by conditional flow within a single class can be distributed across multiple strategy classes, improving readability and enforcing the separation of concerns. [1, p. 316]

2.3.4 Examples of Applicability

Example 1 — Encryption (knowledge boundaries)

A common application of the Strategy Pattern can be found in cryptographic algorithms such as AES, RSA, and Blowfish. These algorithms differ in performance characteristics, security guarantees, and underlying implementation. Most often, however, clients require only the functionality to encrypt, while the specific implementation details are irrelevant to their use. By following the Strategy Pattern, these implementation details remain encapsulated within the concrete strategies, enforcing knowledge boundaries and supporting secure, modular system design.

Example 2 — Payment processing (functional variation)

A second example can be found in modern payment processing systems, where providing multiple payment options has become a requirement within the digital marketplace. Credit card transactions, PayPal transfers, and cryptocurrency payments each achieve the same

overarching goal—executing a financial transaction—but differ in latency, fees, geographic availability, and regulatory constraints. Implementing each payment method as a separate strategy allows clients to vary payment behaviour dynamically without altering the code that initiates the transaction. This promotes extensibility, as new payment providers can be introduced by adding new strategy implementations, rather than modifying existing client logic.

2.3.5 Structure

To illustrate the structure of the Strategy Pattern in practice, this section presents the design of the maze traversal application, showing how the pattern's core participants are realised in concrete components and how they collaborate to produce dynamic traversal behaviour.

The diagram below depicts the relationships between the context, the strategy abstraction, and the concrete strategy implementations used in the application.

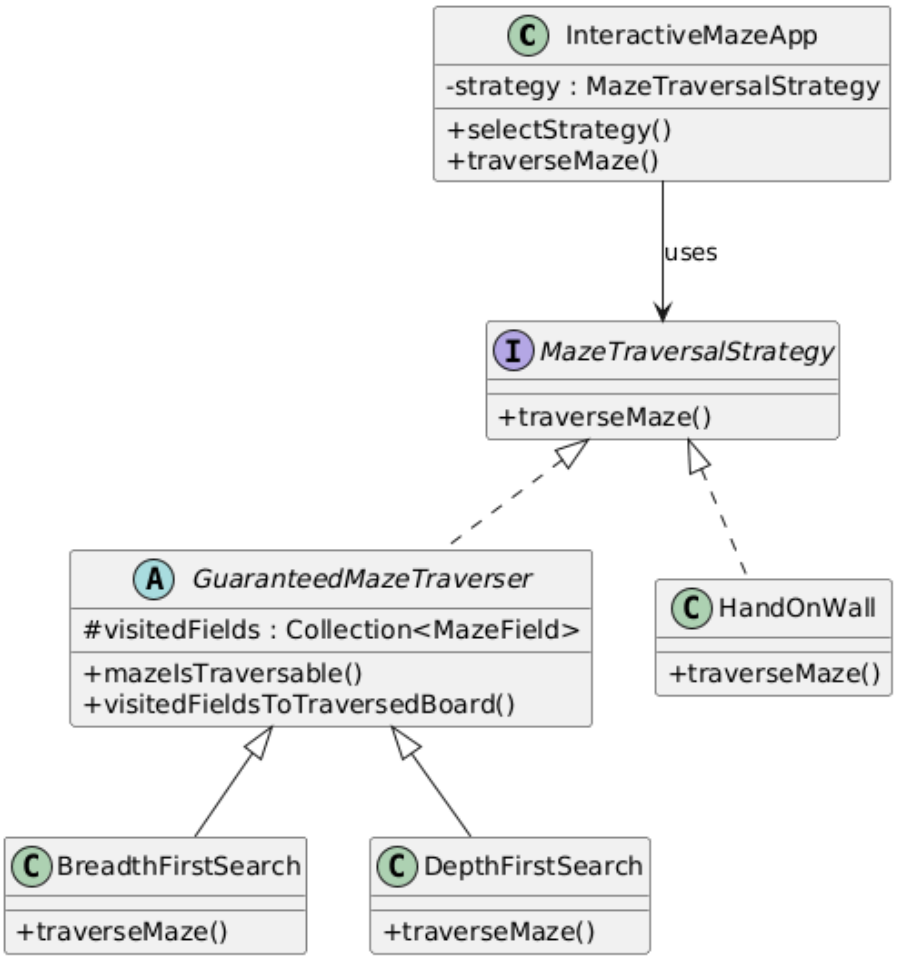

Figure 1: Structure diagram for the maze traversal app

2.3.6 Participants and Collaborations

Context - InteractiveMazeApp

At the highest level, the `InteractiveMazeApp` acts as the context. It maintains a reference to a traversal strategy and delegates traversal requests to that strategy. The context does not require knowledge of how traversal is executed. It is responsible only for selecting a strategy and invoking it.

Strategy - MazeTraversalStrategy

Defines the abstraction common to all traversal algorithms. It specifies the contract for traversal behaviour though the `traverseMaze()` method, which must be implemented by all concrete strategies. By depending only on this interface, the `InteractiveMazeApp` can interchange traversal algorithms at runtime without modifying its own logic.

Abstract Strategy - GuaranteedMazeTraverser

Provides shared behaviour for algorithms that guarantee full maze coverage and termination. It includes a method for reconstructing the traversed maze, as well as logic to evaluate whether a maze is traversable under the given algorithm.

Concrete Strategies - BreadthFirstSearch, DepthFirstSearch, HandOnWall

Implement the `MazeTraversalStrategy` interface:

- `BreadthFirstSearch` and `DepthFirstSearch` extend `GuaranteedMazeTraverser`, inheriting shared traversal functionality.
- `HandOnWall` implements the strategy interface directly, representing a heuristic algorithm without guaranteed coverage.

2.3.7 Consequences

As with any design choice, applying the Strategy Pattern introduces trade-offs. In particular, it affects both design quality and structural complexity. For the maze traversal application, the main costs and benefits are as follows:

+ Extensibility without modification

New traversal algorithms can be introduced by implementing `MazeTraversalStrategy` or extending `GuaranteedMazeTraverser`. Because the context depends only on the abstraction, existing client logic remains unchanged. This supports modular, easily changeable code, aligning with well-established software principles such as the Open/Closed Principle [2, p. 70].

+ Improved separation of concerns

Traversal logic is isolated from user interaction and program flow. The `InteractiveMazeApp` selects and invokes a strategy, while each strategy encapsulates its own traversal process. This ensures that the `InteractiveMazeApp` and the traversal strategies have distinct reasons to change, simplifying reasoning and unit testing. This aligns with established software engineering principles, such as the Single Responsibility Principle [2, p. 62] and the Common Closure Principle [2, p. 105].

+ Reuse of shared functionality

`GuaranteedMazeTraverser` centralises logic common to guaranteed algorithms, including path reconstruction and feasibility checking. This reduces duplication [1, p. 317] across breadth-

first and depth-first implementations, aligning with established principles of component cohesion [2, p. 103-110] and DRY, the general software principle of avoiding repetition [3, p. 27].

+ Algorithm choice based on functional requirements

Different strategies offer different guarantees and behaviours [1, p. 318]. Heuristic algorithms such as `HandOnWall` can be selected for simplicity, while searching algorithms ensure complete traversal at possible performance costs.

+/- Increased structural complexity

The Strategy Pattern introduces an additional interface and an extra level of indirection between the client and the concrete algorithms. This can increase the initial structural complexity of a design and add some architectural overhead [1, p. 318]. However, this same indirection improves modularity by cleanly separating the responsibilities of the client and the strategy, ultimately supporting a clearer and more maintainable architecture.

+/- Enforced uniform interface

All strategies expose the same traversal method signature. This enables interchangeability but may require adaptation when algorithm behaviour does not naturally fit the shared interface.

3 Implementation of the Maze Traversal App

3.1 Realisation of the Strategy Pattern

The maze traversal application (MTA) integrates the Strategy Pattern into its core runtime flow. The application maintains a collection of traversal strategies implementing the `MazeTraversalStrategy` interface. Users select a traversal algorithm dynamically, and the `InteractiveMazeApp` invokes it polymorphically. Because the application depends only on the interface, not on concrete implementations, traversal behaviour is loosely coupled to the rest of the system, allowing strategies to be interchanged at runtime.

The MTA employs two distinct categories of traversal strategies:

- **Guaranteed strategies** ensure full maze coverage and termination. These extend an abstract base class, `GuaranteedMazeTraverser`, which provides shared functionality such as reconstructing the traversed board and evaluating maze traverse-ability.
- **Heuristic strategies**, such as `HandOnWall`, implement `MazeTraversalStrategy` directly and not inherit shared guarantees.

Concrete guaranteed strategies (BFS, DFS) reuse shared utilities while implementing their own traversal logic. Heuristic strategies bypass the base class, reflecting differences in both implementation and expected outcomes.

Polymorphism ensures that the choice of algorithm is a runtime concern rather than a static decision. New traversal strategies can be added without modifying existing application logic, demonstrating adherence to the Open/Closed Principle.

The diagram in Figure 1 summarizes these relationships.

3.2 Structural Overview

The MTA is organised around a small set of core components that together implement the Strategy Pattern. These classes separate user interaction, algorithm selection, and traversal execution, supporting extensibility and modular reasoning about system behaviour.

3.2.1 InteractiveMazeApp (Context)

Coordinates application flow. It:

- Presents available strategies to the user
- Stores the selected strategy as the active `MazeTraversalStrategy`
- Invokes `traverseMaze()` on the active strategy

The context does not need knowledge of the specific algorithm, ensuring loose coupling between UI logic and traversal logic.

3.2.2 MazeTraversalStrategy (Strategy Interface)

Defines a uniform contract:

```
boolean[][] traverseMaze(Maze maze) throws MazeNotTraversableException;
```

Any class implementing this interface provides traversal behaviour. Because the context depends only on this abstraction, strategies can vary independently of the application code.

3.2.3 GuaranteedMazeTraverser (Abstract Base Strategy)

Provides shared behaviour for guaranteed algorithms, including:

- Converting a list of visited `MazeFields` into a traversed board
- Evaluating maze traverse-ability

Concrete guaranteed strategies inherit these utilities, reducing duplication while still implementing unique traversal behaviour.

3.2.4 Concrete Traversal Strategies

Concrete strategies define specific traversal behaviours:

- **BFS / DFS**: Extend `GuaranteedMazeTraverser`, implementing traversal logic while eusing shared utilities.
- **HandOnWall:** Implements `MazeTraversalStrategy` directly. Represents a heuristic algorithm without guaranteed coverage.

3.2.5 Supporting Domain Classes

Traversal strategies utilise supporting domain objects:

- **Maze**: Represents the maze structure, including walls, fields, and adjacency relationships.
- **MazeField**: Represents a position or cell within a maze.

These classes encapsulate maze state and provide the minimal data needed for traversal and post-processing. They remain independent of strategy selection, supporting cohesion and the Single Responsibility Principle.

3.2.6 Interaction Summary

At runtime, the context displays the available strategies, prompts the user for input, and returns the selected strategy:

```
System.out.println("\n✓ Selected: " + strategyNames[userSelectedInput - 1]);

return strategies[userSelectedInput - 1];
```

The selected `MazeTraversalStrategy` is then passed to the method responsible for executing the traversal:

```
boolean[][] traversedBoard = strategy.traverseMaze(maze);
```

The strategy completes its algorithm and returns traversal output, which is displayed to the user. Because all strategies implement the same interface, the context code remains identical regardless of whether BFS, DFS, or `HandOnWall` is selected, demonstrating interface-driven polymorphism and interchangeability of strategies.

3.3 Behaviour and Algorithm Flow

Once a traversal strategy has been selected, the application executes a clearly defined set of interactions between the context, the selected strategy, and the maze domain objects. This section describes the typical runtime flow and highlights where the Strategy Pattern drives the behaviour of the system.

3.3.1 High-level Execution Sequence

1. **Strategy Selection:** User chooses a strategy; the application sets it as active.
2. **Traversal request:** `InteractiveMazeApp` invokes `traverseMaze()` on the selected strategy.
3. **Algorithm execution:**
   - Guaranteed strategies explore the maze completely and track visited fields.
   - Heuristic strategies rely on local decision rules and may not explore the entire maze.
4. **Result return:** The strategy returns traversal output to the `InteractiveMazeApp`, which displays it and asks the user if they would like to request another traversal.

This design keeps traversal a runtime detail, maintaining stability in the logic of the `InteractiveMazeApp`.

3.3.2 Behavioural Diagram

The following sequence diagram illustrates the interactions for a guaranteed strategy (e.g., BFS). Heuristic strategies follow the same pattern but skip post-processing steps.

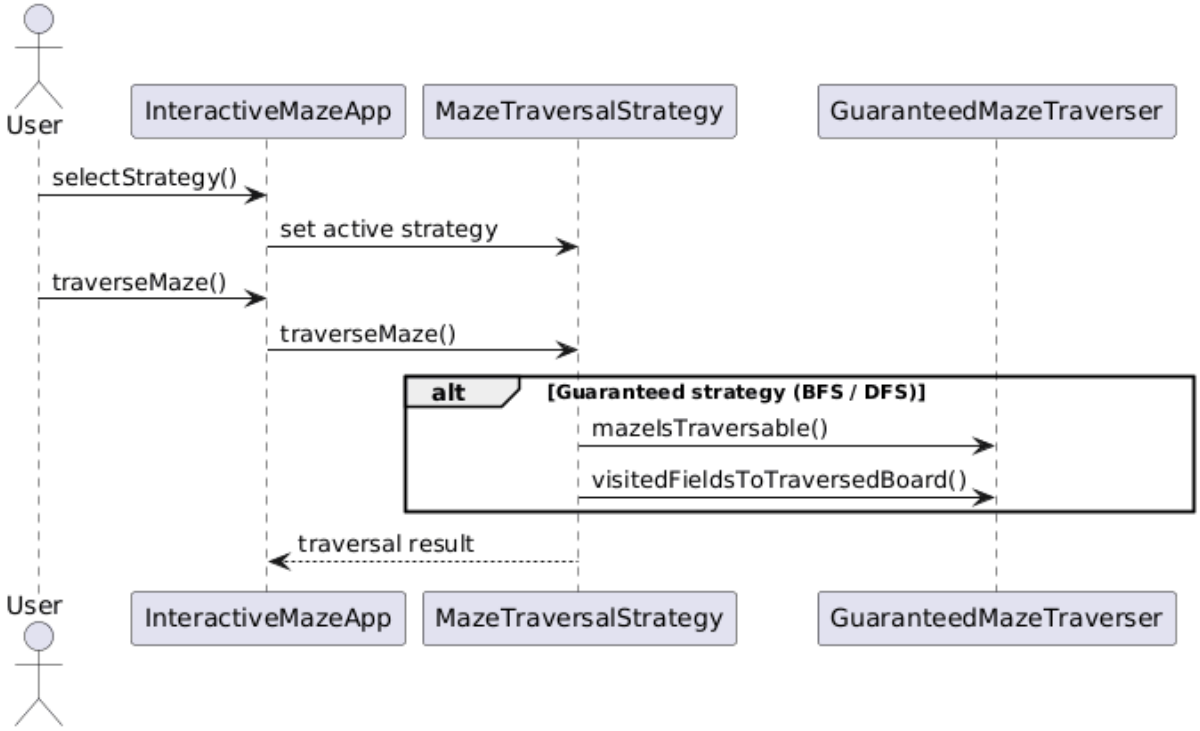

Figure 2: Sequence diagram of maze traversal

3.3.3 Pattern Realisation Observations

The implementation exhibits characteristic behaviours of the Strategy Pattern as defined in established design pattern literature, particularly the GoF text:

Runtime Method dispatch via a shared Interface

The `InteractiveMazeApp` invokes `traverseMaze()` on a reference of type `MazeTraversalStrategy`, and the concrete implementation is selected at runtime. GoF describe this as a core property of the Strategy Pattern, where the context forwards requests to a strategy object without knowing which specific algorithm is in use [1, p. 316 - 317].

Encapsulation of algorithm variability

Differences in traversal behaviour (e.g., queue-based exploration in BFS, stack-based exploration in DFS, or local rule-following in `HandOnWall`) are isolated within the corresponding concrete strategy classes. This reflects the GoF principle that individual algorithms should be encapsulated to allow them to vary independently from the code that uses them [1, p. 315 - 317].

Shared post-processing

Guaranteed strategies inherit feasibility checking and result reconstruction from `GuaranteedMazeTraverser`, reducing duplication and keeping algorithm-specific logic separate from reusable shared operations. This aligns with GoF guidance to avoid conditional logic in the context by introducing helper abstractions when strategies share behaviour [1, p. 316 - 317].

Together, these behavioural characteristics demonstrate that the `MazeTraversalApp` realises the intent of the Strategy Pattern according to established descriptions the literature. They show that algorithm selection is a runtime concern, variability is appropriately encapsulated, and

shared behaviour is structured through inheritance rather than conditional branching in the context.

3.4 Validation and Testing

The applications functionality was validated through unit tests targeting traversal behaviour, feasibility checking, and the methods of the supporting classes. Test cases covered both guaranteed and heuristic algorithms, including failure scenarios where a strategy reports that a maze is not traversable. These tests demonstrate that traversal implementations conform to the shared interface and behave predictably.

4 Related Patterns

The Strategy Pattern is conceptually similar to several other behavioural design patterns but differs in intent and areas of applicability.

4.1 Template Method

Both patterns support varying algorithmic behaviour [1, p. 315, 325]. The Template Method defines a fixed list of abstract primitive methods in a base class with subclasses implementing them to vary their behaviour [1, p. 327]. By contrast, Strategy employs composition as the specific implementation of an algorithm is selected by the context at runtime. Strategy thus supports dynamic substitution, whereas Template Method determines the implementation at compile-time.

4.2 State

Strategy and State share a similar structure with both relying on a common interface with multiple concrete implementations [1, p. 306, 316]. Their difference lies in their intent: State affects object behaviour according to changes in their internal state [1, p. 305], while Strategy models alternative algorithms selected externally by a client [1, p. 315].

4.3 Flyweight

GoF list Flyweight as related to Strategy because strategy objects are typically interchangeable and may be shared in multiple contexts [1, p. 323]. A flyweight separates intrinsic from extrinsic state so that a single object can be shared simultaneously in multiple contexts [1, p. 196]. Since Strategy implementations often have several concurrent implementations, they present intrinsic characteristics that align well with Flyweight usage, making them natural candidates for sharing [1, p. 206].

5 Contemporary Relevance

Despite originating in the mid-1990s, the Strategy Pattern remains highly relevant in modern software systems. This is due to its core intent—encapsulating interchangeable behaviour behind a stable abstraction—being timeless in its application. Modern software systems are expected to be extensible, configurable, and resilient to change, all of which are qualities directly supported by the Strategy Pattern.

This is best observed in user-facing applications, where the Strategy Pattern frequently appears in domains that require flexible business logic. Payment processing systems are a representative example, as discussed previously in the section on the applicability of the Strategy Pattern.

Modern applications routinely support multiple payment providers such as credit cards, digital wallets (e.g., PayPal), and deferred payment services (e.g., Klarna). While the specific details of these payment methods vary, they are commonly integrated behind a uniform abstraction, allowing providers to be added, removed, or substituted without restructuring existing transaction workflows. Similar patterns can be observed in related domains such as tax calculation, discount policies, and shipping cost computation, where behaviour varies by context but must integrate seamlessly into a unified application flow.

Beyond domain-specific use cases, the Strategy Pattern is embedded in modern software frameworks, for example, through dependency injection mechanisms. Frameworks such as Spring encourage applications to depend on interfaces while selecting concrete implementations at runtime via configuration or annotations, following inversion-of-control principles [4]. For example, different authentication strategies—such as database-backed authentication, OAuth2-based identity providers, or token-based mechanisms—can be substituted without changing the consuming components. This design is often not labelled explicitly as "Strategy," but follows the same structural and behavioural principles: algorithms or behaviours are encapsulated behind abstractions and selected dynamically.

In contemporary backend and microservice architectures, the Strategy Pattern is also frequently applied to infrastructure concerns such as persistence. Applications may support relational databases (e.g., PostgreSQL), document stores (e.g., MongoDB), or in-memory caches (e.g., Redis), each implemented behind a common repository or data-access interface. This allows systems to evolve incrementally, replace infrastructure components, or introduce new integrations without widespread refactoring.

Taken together, these examples demonstrate that the Strategy Pattern is not an outdated relic of design practice, but a foundational technique that underpins many contemporary applications, frameworks, and architectural styles. Its continued relevance reflects the enduring importance of modularity, abstraction, and controlled variability in modern software systems.

6 Summary

This paper examined the Strategy Pattern from theoretical and practical perspectives. The pattern enables algorithmic variation to be isolated behind a common abstraction, allowing clients to select behaviour dynamically at runtime while remaining independent of concrete implementations. Strategy supports established software engineering principles, including the Open/Closed Principle, separation of concerns, and modular extensibility.

Real-world applicability was demonstrated through examples such as cryptographic algorithms and payment processing systems. These examples illustrate that Strategy is well suited to domains that require the interchangeability of functionally equivalent behaviours.

The maze traversal application provided a concrete implementation of the pattern. It employed interfaces, inheritance, and runtime strategy selection to support multiple traversal algorithms without modifying client logic. Guaranteed strategies shared common functionality through an abstract base class, while heuristic strategies implemented the core interface directly, illustrating both reuse and variability through encapsulation in practice.

The design met the intent of the Strategy Pattern as described in the GoF and demonstrated practical benefits, including reduced conditional logic, simplified testing of traversal algorithms in isolation, and straightforward extensibility through the addition of new strategies.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA, USA: Pearson Education, 2017.

[3] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA, USA: Addison-Wesley, 1999.

[4] Spring Framework Documentation, "Dependencies and configuration in detail," Broadcom Inc. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html. [Accessed: 21-Dec-2025].