Graduate Research Project Final Paper: A Comparative Exploration of Behavioral Design Patterns in OOP

Katherine Deegan, Yifei Zhang
Katherine.Deegan@colorado.edu
Yifei.Zhang@colorado.edu

Abstract

This research proposes a comparative study of Behavioral Design Patterns in object-oriented programming (OOP). Specifically, it examines the trade-offs among commonly used patterns, analyzing their strengths, weaknesses, and ideal use cases. By investigating how these patterns affect the readability, extensibility, and performance of code, this project aims to provide deeper insight into their practical applications within modern software development.

Introduction

Design patterns in Object-Oriented programming emerged as a strategy to improve the structure, behavior, and relationships of entities, and are typically categorized into three groups: Creational, Structural, and Behavioral Patterns (O., 2022). While each category addresses a distinct set of challenges that can arise when designing software at scale, they all share the same goals of making code more understandable, extensible, and reusable. This research aims to focus specifically on Behavioral Design Patterns, and the trade-offs that exist between popular paradigms within this category.

Behavioral Design Patterns focus on algorithms that enable better interaction between objects. These patterns often promote loose coupling between entities, encapsulation of algorithms within objects, and aim to enable extensibility of code without modification (Sharma, 2025). These principles will be discussed in greater detail in subsequent sections, along with examples of how various design patterns implement them.

In 1994, the highly influential book *Design Patterns: Elements of Reusable Object-Oriented Software* documented what became known as the "Gang of Four" (GoF) Design Patterns, referring to a set of 23 classic software patterns. This text identifies 11 key Behavioral Design Patterns, including the Template Method, Mediator, Chain of Responsibility, Observer, Strategy, Command, State, Visitor, Interpreter, iterator and Memento (Gamma et al., 1994).

These patterns form the basis for our comparative study, which explores similarities and distinctions between patterns, and provides guidance on how developers can make informed decisions when selecting the Behavioral Design Pattern best suited for their particular use case.

Literature Review

Research on Behavioral Design Patterns has been widely discussed since the publication of Design Patterns: Elements of Reusable Object-Oriented Software by Gamma et al. (1994), which established the foundational taxonomy of Creational, Structural, and Behavioral patterns. The Gang of Four (GoF) patterns continue to influence contemporary software design, emphasizing reusability, decoupling, and scalability across large systems.

Subsequent studies expanded the GoF framework to evaluate its practical relevance and evolution. Aversano et al. (2007) and Wedyan and Abufakher (2020) conducted empirical analyses on how design patterns affect software quality, highlighting measurable improvements in maintainability and cohesion when patterns are applied consistently. Similarly, Kumar and Saxena (2012) emphasized the correlation between coupling, cohesion, and system complexity, reinforcing that Behavioral Patterns support low coupling and high cohesion through encapsulated interactions.

More recent works explore pattern selection and comparative methodologies. Hasheminejad and Jalili (2012) proposed automated techniques for selecting suitable patterns based on software requirements, while Naghdipour, Hasheminejad, and Barmaki (2023) performed a systematic review of pattern selection approaches, identifying extensibility, scalability, and performance as key decision criteria. Complementary discussions by Manchana (2019) and Sharma (2025) focus on modern OOP implementations, examining

how Behavioral Patterns enhance communication and algorithmic flexibility among interacting components.

Practical educational resources such as Refactoring.Guru (2025) and Freeman and Robson (2021) have contributed to making these patterns accessible to practitioners, illustrating their real-world use in modern frameworks. Collectively, the literature reveals that while Behavioral Patterns share common goals (flexibility, maintainability, and separation of concerns). They vary significantly in structure, runtime cost, and suitability across contexts.

This research builds upon these studies by providing a comparative exploration of all major Behavioral Patterns, combining structural analysis with empirical performance evaluation to clarify their respective trade-offs and ideal use cases.

Methodology

This research employs a mixed-method approach combining qualitative analysis and quantitative evaluation to examine the trade-offs among Behavioral Design Patterns. Eleven patterns identified by the Gang of Four (Gamma et al., 1994) were selected for analysis.

Qualitative Analysis

Each pattern was analyzed based on its structure, design intent, and trade-offs, focusing on three key OOP principles: encapsulation, decoupling, and extensibility. Class and interaction diagrams were used to illustrate relationships between components. The analysis drew from established sources (Sharma, 2025; Naghdipour, Hasheminejad, and Barmaki, 2023) to ensure theoretical grounding and to highlight where patterns overlap or diverge in purpose and structure.

Quantitative Evaluation

To complement the qualitative findings, each pattern was implemented in Java 22 / Java 24 within a controlled environment. Performance metrics were measured using representative scenarios. For example, the Observer and Command patterns were compared by increasing the number of dependents, and the Chain of Responsibility pattern was tested across varying chain lengths. Results were summarized through descriptive statistics to highlight practical performance trends.

Case Studies

Simplified case studies were developed to demonstrate each pattern's real-world behavior, including event-driven updates (Observer), lifecycle transitions (State), and dynamic algorithm selection (Strategy). Peer code reviews were conducted to assess readability and maintainability, validating theoretical insights through practical implementation.

Key Principles of Behavioral Patterns

Behavioral Design Patterns build upon key object-oriented principles that define how objects collaborate and share responsibilities. These principles aim to improve clarity, maintainability, and adaptability in complex systems. This section outline three core ideas: encapsulation of behavior, loose coupling, and flexibility and extensibility, which form the foundation of many behavioral patterns.

Encapsulation of Behavior

The foundation of object-oriented design is packaging data, and the methods operating on that data, into an object. Client requests are the only way to execute these operations, and operations are the only way to change the object's internal data. When the object's internal representation is hidden in this way, the object's internal state is said to be encapsulated (Gamma et al., 1994). Many Behavioral Design Patterns strive to encapsulate the interactions between subsystems by creating objects that act as communication facilitators, encapsulating the protocols and algorithms that govern how objects interact, while keeping the interacting parties independent of each other. When complex behaviors such as these are successfully decomposed into objects, the result is often a much more readable, easy-to-follow codebase (Sharma, 2025).

Loose Coupling

In software design, coupling (the interlinking of classes) and cohesion (the strength of the individual classes) are closely correlated metrics controlling the complexity of a system. The optimal object-oriented designs are based on achieving low coupling and high cohesion levels (Kumar and Saxena, 2012). Behavioral patterns reinforce this principle by ensuring that each object is aware only of the actions it must perform within the broader system operation, without becoming tightly coupled to other participating objects (Sharma, 2025).

Flexibility and Extensibility

Encapsulating highly cohesive, loosely coupled objects enables designers to construct complex systems with extensive component interactions, while maintaining the flexibility to modify or extend functionality at runtime without altering existing code (Sharma, 2025).

GoF Behavioral Design Patterns

This section provides a brief overview of each of the 11 Behavioral Design Patterns previously mentioned, focusing on the unique characteristics that distinguish each one. Code examples for each pattern can be found in our project's GitHub repository.

Chain of Responsibility

Intent: The Chain of Responsibility Pattern ensures loose coupling between a request sender and receiver by establishing a succession of receiving objects that will

handle the incoming request or pass it along to the next object in the chain (Gamma et al., 1994).

Structure: The pattern defines a *Handler* interface which contains a method for handling requests. *Concrete Handlers* implement this abstraction, and have access to their successor (another *Concrete Handler*), and will forward the request down the chain if they are unable to address it themselves. A *Client* initiates the request to a *Concrete Handler* in the chain but has no knowledge of who will ultimately fulfill their request. (Refactoring.Guru, 2025)

Command

Intent: The Command Pattern focuses on encapsulating a request as an object, enabling clients to parameterize objects with different requests at runtime and decoupling the object that invokes the operation from the one that knows how to perform it (Gamma et al., 1994).

Structure: This pattern specified a *Command* interface, typically with just a single method to handle the execution of an operation. This abstraction is then implemented by *Concrete Command* objects (Gamma et al., 1994). *Client* code creates and configures *Concrete Command* objects, specifying a *Receiver* (request handler) instance in the process. One or multiple *Invoker* classes may be configured, each with a strong reference to this *Command* object, and these classes are responsible for actually initiating requests. Once an *Invoker* object triggers a request, the *Command* object invokes the corresponding operations on the *Receiver* object to fulfill the request (Refactoring.Guru, 2025).

Interpreter

Intent: The Interpreter Pattern is used to model the representation for the grammar of a specific language as an abstract syntax tree, and use that representation to interpret sentences in the language (Gamma et al., 1994).

Structure: This pattern involves an *Expression* abstraction, which is common to all nodes in the syntax tree and specifies an *interpret()* method. Both *Terminal Expression* and *Nonterminal Expression* objects populate the syntax tree and provide concrete implementations for *interpret()*.

Iterator

Intent: The Iterator Pattern provides a way to access elements of an aggregate object sequentially, without exposing the underlying structure (Gamma et al., 1994).

Structure: This pattern is implemented with an *Iterator* interface, which declares methods required for traversing a collection (i.e. fetching the next element, retrieving the current position, etc.), and an *Iterable Collection* interface, which declares a method to return an *Iterator* object (i.e. *createIterator()*). *Concrete Iterators* implement specific algorithms for traversing a collection, and *Concrete Collections* return a new instance of an *Iterator* object each time a client requests one (Refactoring.Guru, 2025).

Mediator

Intent: The Mediator Pattern encapsulates how a set of objects interact, promoting loose coupling between objects by preventing direct communication between them (Gamma et al., 1994).

Structure: This pattern includes *Component* classes, which vary in type but all contain a reference to a *Mediator* object. The *Mediator* defines an interface for communicating between components, typically via a single *notify()* method. *Concrete Mediators* implement behavior for interaction between components, often by maintaining references to all the components they manage (Refactoring.Guru, 2025).

Memento

Intent: Without breaking encapsulation, the Memento pattern captures and externalizes the internal state of an object, enabling it to be restored to this state at a later time (Gamma et al., 1994).

Structure: This pattern involves an *Originator* object that creates and sets *Memento* objects to save its internal state. A *Caretaker* object manages and stores *Mementos* without modifying or examining their contents (Gamma et al., 1994).

Observer

Intent: The Observer pattern defines a one-to-many relationship between one subject object and its dependent (observer) objects, so when the subject changes state, its dependents are notified and updated accordingly (Gamma et al., 1994).

Structure: This pattern involves a *Subject* interface, which maintains a list of observers and methods for attaching and detaching *Observer* objects. The *Observer* interface defines a method for receiving updates when the *Subject* changes state and is implemented by one or multiple *Concrete Observer* objects. *Concrete Subject* objects store a state of interest and notify observers upon changes (Gamma et al., 1994).

State

Intent: The State Patterns enables an object to alter its behavior in response to changes with its internal state, making it appear as if the object changed classes (Gamma et al., 1994).

Structure: This pattern includes a *Context* object, which maintains an instance of a *State* object, which represents its current condition. The *State* interface encapsulates a behavior linked to a specific state of the

Context via a *handle()* method, and *Concrete State* objects project implementations for this method (Gamma et al., 1994).

Strategy

Intent: The Strategy Pattern encapsulates a family of algorithms, allowing algorithms to be interchangeable and vary independently from the clients that use them (Gamma et al., 1994).

Structure: This pattern consists of a *Context* that contains a strong reference to a *Strategy* object. A *Strategy* interface is common among *Concrete Strategy* objects. The concrete strategies implement different variations of an algorithm (Refactoring.Guru, 2025).

Template Method

Intent: The Template Method focuses on encapsulating the steps of an algorithm by defining the skeleton of an operation and deferring the implementation of some steps to subclasses (Gamma et al., 1994).

Structure: This method consists of an *Abstract Class* that defines a *template method* which specifies the sequence of algorithm steps. This method typically includes abstract operations or hooks that must be implemented by *Concrete Class* objects (the subclasses) (Gamma et al., 1994).

Visitor

Intent: The Visitor pattern allows you to encapsulate algorithms from the objects on which they operate, allowing you to define new functionality for classes without modifying the existing class code (Musch, 2023).

Structure: This patterns defines a *Visitor* interface, which declares a set of visiting methods, and an *Element* interface, which declares a method for accepting visitors. The *visit()* methods defined in the *Visitor* interface each accept a specific type of *Concrete Element* as an argument, and the *Concrete Visitor* objects implement several versions of the same behavior, tailored to the specific element types. *Client* code typically manages a collection of *Concrete Elements*, and invokes the *Concrete Visitor's* functionality by calling a *Concrete Element's accept()* methods (Refactoring.Guru, 2025).

Comparative Analysis of Behavioral Patterns

This section examines the key similarities and differences between the patterns introduced in the previous section, as well as notable trade-offs between similar patterns.

Decoupling Comparison

Both the Observer and Command patterns focus on decoupling dependent objects, however, with different approaches. The Observer pattern decouples subjects and observers (dependents) dynamically, allowing observers to be attached or detached from a subject at runtime, while the Command pattern decouples senders from receivers by encapsulating requests as command objects, allowing senders to remain unaware of who fulfills the request and receivers to have no knowledge of who initiated it (Filho, 2024; Manchana, 2019). Examples of this decoupling behavior can be found in the project repo.

Patterns like Mediator centralize communication, improving modularity with independent Components, but can potentially increase complexity as the mediator itself contains all logic related to communication rules (i.e. message forwarding, filtering, etc.) and is at risk of becoming bloated and violating the Single Responsibility Principle as it accumulates routing logic. Patterns such as the Chain of Responsibility maintains better adherence to the Single Responsibility Principle, with each handler focused on a specific request type.

Encapsulation Comparison

Both the Strategy and State patterns favor composition over inheritance by encapsulating behaviors as objects, allowing behaviors to be swapped at runtime. The Strategy pattern encapsulates interchangeable algorithms, allowing them to change independently of the clients that use them, where as the State pattern encapsulates states-dependent behavior, improving clarity when an object's behavior changes over time (Manchana, 2019; Xiong, Lo, and Li, 2020). Some patterns like Template Method, only partially encapsulate behavior through sub-classing, offering less flexibility but simpler structure. Examples of the variance in behavior encapsulation between these three methods can be found the project repo.

Flexibility Comparison

Patterns such as Strategy and Command are highly flexible because they favor composition over inheritance. Others such as Template Method, rely on inheritance, which can reduce flexibility but simplify reuse (Aversano et al., 2007; Xiong, Lo, and Li, 2020).

Performance Comparison

Certain patterns have notable performance concerns that can arise for systems at scale. For example, the Observer pattern can affect performance when many observers are notified frequently, as the time to notify observers scales linearly with the number of observers. The Observer vs Command comparison code example in this project's repo demonstrates how the time to notify 1 observer vs 1000 observers increases by about 5x.

Similarly, implementing the Chain of Responsibility at scale may require multiple handlers to process a single request, which can slow execution but improves scalability and separation of concerns. Engineers should

balance performance with design clarity when applying these patterns (Wedyan and Abufakher, 2020; Aversano et al., 2007).

Choosing the Right Pattern

This section examines the contexts in which each Behavioral Design Pattern is best suited, and how engineers can select the most appropriate pattern based on their particular design goals and system constraints.

Criteria for Selection:

- **System complexity:** For systems with high behavioral variability, patterns like Strategy or State simplify code organization by encapsulating dynamic behavior changes. The DPSA framework proposes evaluating system behavior variability and adaptability as key criteria for selecting behavioral patterns (Naghdipour, Hasheminejad, and Keyvanpour, 2021).

  Both the Strategy and State patterns enable developers to encapsulate this behavior variance through dedicated objects, so the decision between the two patterns is largely decided by the use case at hand. If there is an algorithm that is best suited for a particular use case for a context object that will not fundamentally change, the Strategy Pattern is likely better suited, as the client typically specifies what strategy object the context is composed of (Freeman and Robson, 2021). An example of this might be selecting between sorting or file compression algorithms based on performance or input size, as shown in the project repo.

  The State Pattern can be a better choice when behavior changes are context state-dependent, and this state is expected to change overtime (Freeman and Robson, 2021). This pattern could be beneficial for order processing systems, as shown in the project repo. For this example, the choice on how to process a particular order (for example, the ability to cancel an order) may depend on if the order is still "pending" (able to cancel) or "shipped" (unable to cancel). In this case, developers likely want objects to be able to manage their own state transitions and adjust dependent behaviors accordingly.

- **Extensibility vs. maintainability:** Command and Observer patterns often enhance extensibility but may increase coupling, affecting maintainability. The systematic review by Naghdipour et al. highlights this trade-off as one of the most frequently cited factors in real-world pattern selection (Naghdipour, Hasheminejad, and Barmaki, 2023).

- **Performance vs. scalability:** Some patterns introduce overhead due to extra abstraction layers; Chain of Responsibility or Mediator can impact runtime performance but improve scalability by decoupling components. Empirical results show that selection tools and frameworks can model such trade-offs by quantifying execution overheads and scalability impacts (Hasheminejad and Jalili, 2012).

  The Mediator pattern is highly scalable, as the addition of new components or communication rules is centralized in a single location. The Chain of Responsibility pattern also scales easily through easy addition, removal, or reordering of handlers, yet performance may suffer when high-priority requests must traverse the entire handler chain, resulting in O(n) complexity for a chain of n handlers. The project repo demonstrates a Support Ticket example use case, where a critical Support Request is passed through every handler in the chain before being addressed.

  Another performance tradeoff arises when comparing the Observer and Command patterns. At a high level, both patterns allow an action in one object (such as a state change or method call) to trigger behavior in another object, while keeping the two objects loosely coupled and unaware of each other's implementation. From a performance perspective, the Observer Patter execution time is $O(n)$ for n observers, which could be a significant consideration for developers designing a system with many dependents. In this case, encapsulating operations via the Command Pattern could be a better solution, as the execution time for a single Command is $O(1)$ and this pattern allows for operations to be queued (delayed execution) and undone or redone.

- **Team and Project Context:** The choice of pattern should align with the team's familiarity and the project's life cycle stage-rapid prototyping may favor simpler structures, while long-term projects benefit from more formalized design abstractions. Modern selection approaches increasingly incorporate organizational and human factors into the decision process (Naghdipour, Hasheminejad, and Barmaki, 2023).

Practical Guidelines:

Based on the comparative analysis and empirical results presented in this study, several practical guidelines can be derived to assist engineers in selecting the most appropriate Behavioral Design Pattern for their specific context:

- **Match the Pattern to Behavioral Variability:** When system behavior changes dynamically with internal state, the State Pattern offers clearer structure and maintainability. In contrast, when multiple interchangeable algorithms are needed without altering object states, the Strategy Pattern provides greater flexibility and reusability.

- **Balance Extensibility and Maintainability:** Command and Observer patterns enhance extensibility through decoupling, but may introduce higher maintenance overhead as the number of dependent objects or commands increases. These patterns should be applied where scalability outweighs simplicity.

- **Optimize for Scalability versus Performance:** Patterns such as Mediator and Chain of Responsibility scale well by decentralizing control logic or distributing request handling. However, they incur additional message-passing or traversal overheads, making them more suitable for modular systems rather than latency-sensitive applications.
- **Favor Composition for Flexibility:** Empirical comparisons show that composition-based patterns (Strategy, Command, State) achieve higher adaptability than inheritance-based patterns (Template Method), enabling runtime reconfiguration without modifying existing code.
- **Consider Project Context and Team Expertise:** For rapid prototyping or smaller projects, simpler patterns such as Template Method or Observer ensure faster implementation. For long-term or enterprise-scale systems, patterns emphasizing decoupling (Mediator, Strategy) better support evolution and scalability.

These guidelines synthesize the key findings of this study—showing that the optimal pattern depends on the dominant design goal. When performance is critical, lightweight and direct structures perform better; when adaptability and scalability are prioritized, composition-based, decoupled patterns deliver superior long-term benefits.

Key Points

1. **Identify key similarities and differences among the major Behavioral Design Patterns.** This provides a foundation for understanding how patterns overlap or diverge in structure and intent.
   References: (Gamma et al., 1994), (Refactoring.Guru, 2025)
2. **Analyzing the programming contexts in which certain patterns are more suitable for than others.** This highlights practical decision-making strategies for engineers working in varied system environments.
   References: (Refactoring.Guru, 2025)
3. **Evaluate how these patterns address key OOP design principles such as decoupling, encapsulation, and open-closed extensibility.** This connects classical principles with concrete pattern-level implementations.
   References: (Gamma et al., 1994), (Larman, 1998).
4. **Examine certain trade-offs that exist between patterns.** This clarifies how developers can weigh different options when solving equivalent challenges.
   References: (Fowler, 2012)
5. **Identify any noticeable performance differences between design patterns that address similar problems.** While design patterns primarily target maintainability, some patterns may introduce runtime or memory overhead that influences scalability.
6. **Demonstrate concrete code examples that illustrate the application and impact of each pattern.** Real-world examples provide tangible evidence of how theory translates into practice.
   References: (Buschmann, Henney, and Schmidt, 2007), (Musch, 2023)

Conclusion and Future Work

This research provides a comparative exploration of Behavioral Design Patterns in object-oriented programming, highlighting their theoretical foundations, practical implementations, and performance trade-offs. Through qualitative and quantitative analysis, this study demonstrated how patterns such as Strategy, State, Observer, and Command differ in their approach to encapsulation, coupling, and flexibility. By contrasting their design intent and performance outcomes, the research contributes to a clearer understanding of when and why particular Behavioral Patterns are most effective in real-world systems.

Contributions

- **Comparative Insights:** A systematic comparison of the eleven GoF Behavioral Design Patterns, clarifying their structural and functional distinctions and the trade-offs involved in applying each pattern.
- **Decision-Making Framework:** Extension of this study to Structural and Creational patterns, and exploration of integration with modern software frameworks.
- **Empirical Validation:** Practical implementations in modern Java environments that demonstrate how theoretical principles of decoupling, encapsulation, and extensibility manifest in executable code.

Future Work

- **Expansion to Other Pattern Categories:** Extending the comparative framework to include Structural and Creational Design Patterns, thereby enabling a holistic cross-category analysis of pattern relationships.
- **Integration with Modern Frameworks:** Investigating how Behavioral Design Patterns can be effectively integrated within contemporary software architectures such as microservices, reactive systems, and AI-driven platforms.
- **Empirical Evaluation at Scale:** Conducting larger-scale experimental studies across industrial

codebases to quantify long-term impacts of pattern selection on maintainability, scalability, and runtime efficiency.

Through these future extensions, this research aims to further bridge the gap between pattern theory and applied software engineering practice, supporting developers and researchers in designing more adaptive, maintainable, and high-performance systems.

References

Aversano, L.; Canfora, G.; Cerulo, L.; Grosso, C. D.; and Penta, M. D. 2007. An empirical study on the evolution of design patterns. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 385–394.

Buschmann, F.; Henney, K.; and Schmidt, D. C. 2007. *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*. John wiley sons.

Filho, N. S. 2024. Comparative analysis of patterns: Distinctions and applications of behavioral, creational, and structural patterns. *Leaders Tec* 1(11).

Fowler, M. 2012. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Freeman, E., and Robson, E. 2021. *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software 2nd Edition*. O'Reilly Media.

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

Hasheminejad, S. M. H., and Jalili, S. 2012. Design patterns selection: An automatic two-phase method. *Journal of Systems and Software* 85(2):408–424.

Kumar, S., and Saxena, V. 2012. Impact of coupling and cohesion in object-oriented technology. *Journal of Software Engineering and Applications* 5(9):671–676.

Larman, C. 1998. *Applying UML and Patterns*. Prentice Hall.

Manchana, R. 2019. Behavioral design patterns: Enhancing software interaction and communication. *International Journal of Science Engineering and Technology* 7:1–18.

Musch, O. 2023. *Design Patterns with Java: An Introduction*. Springer Vieweg Wiesbaden.

Naghdipour, A.; Hasheminejad, S. M. H.; and Barmaki, R. L. 2023. Software design pattern selection approaches: A systematic literature review. *Software: Practice and Experience* 53(4):1091–1122.

Naghdipour, A.; Hasheminejad, S. M. H.; and Keyvanpour, M. R. 2021. Dpsa: A brief review for design pattern selection approaches. In *Proceedings of the 2021 26th International Computer Conference, Computer Society of Iran (CSICC)*, 1–6. IEEE.

O., W. 2022. Gang of four: Fundamental design patterns. Medium blog.

Refactoring.Guru. 2025. Behavioral design patterns. Website.

Sharma, P. 2025. Deep-dive: Behavioral design patterns in java. Medium blog.

Wedyan, F., and Abufakher, S. 2020. Impact of design patterns on software quality: a systematic literature review. *IET Software* 14(1):1–17.

Xiong, R.; Lo, D.; and Li, B. 2020. Distinguishing similar design pattern instances through temporal behavior analysis. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 296–307. IEEE.