Amrita Vishwa Vidyapeetham
TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

Deepthi J

21st January, 2024

Paradigm 1: Functional Programming

Principles and Concepts of Functional Programming:

0.1 What is functional programming?

- Functional programming is a declarative programming paradigm style where one applies pure functions in sequence to solve complex problems.
- Functions take an input value and produce an output value without being affected by the program.
- Functional programming mainly focuses on “what to solve” in contrast to an imperative style where the main focus is “how to solve”.
- It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.
- Functional programming excels mostly at mathematical functions where the values don’t have any correlation and don’t make use of concepts like shared state and mutable data used in object-oriented programming.

0.2 Functional Programming Concepts:

1. **Pure Functions:**

    - Pure functions form the foundation of functional programming and have two major properties:
        (a) They produce the same output if the given input is the same.
        (b) They have no side effects i.e. they do not modify any arguments or local/global variables or input/output streams.
    - The pure function’s only result is the value it returns. They are deterministic.

- Programs done using functional programming are easy to debug because pure functions have no side effects or hidden I/O.
- Pure functions also make it easier to write parallel/concurrent applications. When the code is written in this style, a smart compiler can do many things – it can parallelize the instructions, wait to evaluate results when needing them, and memorize the results since the results never change as long as the input doesn't change.
- Pure functions work well with immutable values as they describe how inputs relate to outputs in declarative programs. Because pure functions are independent this means that they are reusable, easy to organize, and debug, making programs flexible and adaptable to changes.
- Another advantage of using pure functions is memoization. This is when we cache and reuse the results after computing the outputs from the given inputs.

2. **Recursion:**
   - Unlike object-oriented programming, functional programming doesn't make use of "while" or "for" loops or "if-else" statements. Functional programs avoid constructions that create different outputs on every execution.
   - Iteration in functional languages is implemented through recursion. Recursive Functions call themselves repeatedly until they reach the desired state or solution known as the base case.
3. **Immutability:**
   - In functional programming, we can't modify a variable after being created. The reason for this is that we would want to maintain the program's state throughout the runtime of the program.
   - It is best practice to program each function to produce the same result irrespective of the program's state. This means that when we create a variable and assign a value, we can run the program with ease fully knowing that the value of the variables will remain constant and can never change.
4. **First-Class Functions and High-Order Functions:**
   - First-class functions in functional programming are treated as data type variables and can be used like any other variables. These first-class variables can be passed to functions as parameters, or stored in data structures
   - A function that accepts other functions as parameters or returns functions as outputs is called a high-order function. This process applies a function to its parameters at each iteration while returning a new function that accepts the next parameter.

Language for Paradigm 1: Clojure

Characteristics and Features of Clojure associated with Functional Programming:

- Clojure is a dynamic, general-purpose programming language, that combines the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming.

- Clojure is predominantly a functional programming language and features a rich set of immutable, persistent data structures. When a mutable state is needed, Clojure offers a software transactional memory system and reactive Agent system that ensure clean, correct, multithreaded designs.
- Clojure provides the tools to avoid mutable states, provides functions as first-class objects, and emphasizes recursive iteration instead of side-effect-based looping. Clojure is impure, yet stands behind the philosophy that more functional programs are more robust.

1. **Immutable Data Structures:**
    - Clojure encourages the use of immutable data structures. Once created, data structures cannot be modified, and any operation that seems to modify them returns a new structure.
    - **Example:**
      ```
      ;; Creating an immutable vector
      (def my-vector [1 2 3])
      ;; Adding an element creates a new vector
      (def new-vector (conj my-vector 4))
      ;; my-vector is still unchanged
      ```
2. **First-Class Functions:**
    - Functions are first-class citizens in Clojure, meaning they can be assigned to variables, passed as arguments, and returned as values.
    - **Example:**
      ```
      ;; Defining a function
      (defn square [x] (* x x))
      ;; Assigning a function to a variable
      (def my-func square)
      ;; Using the function as an argument
      (def result (my-func 5)) ; Returns 25
      ```
3. **Higher-Order Functions:**
    - Clojure supports higher-order functions, allowing functions to take other functions as arguments or return functions as results.
    - **Example:**
      ```
      (defn apply-twice [fn x] (fn (fn x)))
      ;; Using the higher-order function
      (def square-twice (apply-twice square))
      (square-twice 3) ; Returns 81
      ```
4. **Anonymous Function (Lambdas):**
    - Clojure supports the creation of anonymous functions using the fn keyword.

- **Example:**
  ```
  ;; Anonymous function (lambda)
  (def add-one (fn [x] (+ x 1)))
  ;; Using the anonymous function
  (add-one 5) ; Returns 6
  ```

5. **Lazy Squeneces:**
   - Clojure supports lazy sequences, allowing efficient handling of large data sets by computing values only as needed.
   - **Example:**
     ```
     ;; Lazy sequence example
     (defn infinite-sequence [] (iterate inc 0))
     ;; Using the lazy sequence
     (take 5 (infinite-sequence)) ; Returns (0 1 2 3 4)
     ```

6. **Concurrency with Atoms:**
   - Clojure provides concurrency support with constructs like atoms.
   - **Example:**
     ```
     ;; Example of an atom (mutable reference)
     (def counter (atom 0))
     ;; Function to update the atom in a thread-safe way
     (defn increment-counter []
     (swap! counter inc))
     ```

- These examples highlight key functional programming concepts in Clojure, such as immutability, first-class functions, higher-order functions, lazy evaluation, and concurrency support. Clojure's syntax and built-in features make it well-suited for expressing functional programming ideas concisely and clearly.

Paradigm 2: Event-Driven Programming

Principles and Concepts of Event-Driven Programming:

- Event-driven programming is a programming paradigm in which the flow of program execution is determined by events - for example, a user action such as a mouse click, key press, or a message from the operating system or another program.
- An event-driven application is designed to detect events as they occur, and then deal with them using an appropriate event-handling procedure.
- The idea is an extension of interrupt-driven programming of the kind found in early command-line environments such as DOS, and in embedded systems (where the application is implemented as firmware).
- It allows the creation of dynamic applications where the flow of control is determined by the sequence of events, rather than a predetermined order of execution.

- The primary goal of Event Driven Programming is to make the software more responsive to user actions and to simplify the development process by providing a clear separation between event handling and other aspects of the software design.

Key Components of Event Driven Programming:

- Several key components in Event Driven Programming work together to handle and process events efficiently. Understanding these components is essential for creating successful Event Driven applications.

1. **Event Handlers:**
    - Event handlers are the backbone of Event Driven Programming. These are functions or methods that are designed to be triggered when a specific event occurs. For instance, when a user clicks on a button on a graphical user interface, an event handler associated with that button responds by executing the designated code.
    - Event handlers can be further categorized as:
        (a) Synchronous event handlers: Execute the code immediately when an event occurs.
        (b) Asynchronous event handlers: Allow other tasks to continue executing while the code for handling the event is being processed.
2. **Event Loop:**
    - The event loop is a continuous process that runs in the background and checks for any queued events.
    - When an event is detected, the event loop dispatches it to the appropriate event handler for processing. It then moves on to the next event in the queue, ensuring that all events are handled as they occur.
    - The event loop is responsible for managing the event queue and maintaining the responsiveness of the application.
3. **Event Queue:**
    - The event queue is a data structure that holds events waiting to be processed by their associated handlers. Events are added to the queue as they occur, and they are removed and dispatched to the corresponding event handlers by the event loop.
    - The event queue ensures that events are handled in the order in which they are received, and they also help manage concurrency and synchronization issues in Event-driven applications.
    - In Event-Driven Programming, it is essential to carefully manage the event queue to prevent bottlenecks and maintain the responsiveness of the application.
    - Properly handling events in the event queue ensures that the software can continue to process user input and other tasks while still remaining responsive to new events as they arise.

Background

- Before the arrival of object-oriented programming languages, event handlers would have been implemented as subroutines within a procedural program. The flow of program execution was determined by the programmer, and controlled from within the application's main routine. The complexity of the logic involved required the implementation of a highly structured program. All of the program's code would be written by the programmer, including the code required to ensure that events and exceptions were handled, as well as the code required to manage the flow of program execution.
- In a typical modern event-driven program, there is no discernible flow of control. The main routine is an event-loop that waits for an event to occur, and then invokes the appropriate event-handling routine. Since the code for this event loop is usually provided by the event-driven development environment or framework, and largely invisible to the programmer, the programmer's perception of the application is that of a collection of event handling routines. Programmers used to working with procedural programming languages sometimes find that the transition to an event-driven environment requires a considerable mental adjustment.
- The change in emphasis from procedural to event-driven programming has been accelerated by the introduction of the Graphical User Interface (GUI) which has been widely adopted for use in operating systems and end-user applications. It really began, however, with the introduction of object-oriented (OO) programming languages and development methodologies in the late 1970s. By the 1990's, object-oriented technologies had largely supplanted the procedural programming languages and structured development methods that were popular during the 70s and 80s.
- One of the drivers behind the object oriented approach to programming that emerged during this era was the speed with which database technology developed and was adopted for commercial use. Information system designers increasingly saw the database itself, rather than the software that was used to access it, as the central component of a computerized information system. The software simply provided a user interface to the database, and a set of event-handling procedures to deal with database queries and updates.
- One of the fundamental ideas behind object-oriented programming is that of representing programmable entities as objects. An entity in this context could be anything with which the application is concerned. A program dealing with tracking the progress of customer orders for a manufacturing company, for example, might involve objects such as "customer", "order", and "order item".
- An object encompasses both the data (attributes) that can be stored about an entity, the actions (methods) that can be used to access or modify the entity's attributes, and the events that can cause the entity's methods to be invoked. The basic structure of an object, and its relationship to the application to which it belongs, is illustrated in the diagram below.

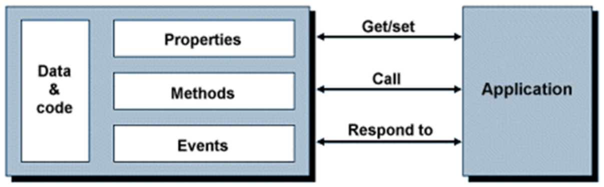

Figure 1: The relationship between an object and an application

- Before going any further, it is worth distinguishing objects and classes. A class, in very general terms, is an object template that defines the attributes, methods, and events that will be implemented in any object created with it. An object is an instance of a class. An analogy would be to say that "Dog" is a class, of which "Fido" is a specific instance. "Fido" has all of the generic characteristics and behaviors of the class "Dog", and responds to the same external stimuli.

- The link between object-oriented programming and event-driven programming is fairly obvious. For example, objects on a Visual Basic form (usually referred to as controls) can be categorized into classes (e.g. "Button", "TextBox" etc.), and many instances of each can appear on a single form. Each class will have attributes (usually referred to as properties) that will be common to all objects of that type (e.g. "BackgroundColour", "Width" etc.), and each class will define a list of events to which an object of that type will respond. The methods (event handlers) to handle specific events are usually provided as templates to which the programmer simply has to add the code that carries out the required action.

How Event-Driven Programming works?

- The central element of an event-driven application is a scheduler that receives a stream of events and passes each event to the relevant event-handler. The scheduler will continue to remain active until it encounters an event (e.g. "EndProgram") that causes it to terminate the application. Under certain circumstances, the scheduler may encounter an event for which it cannot assign an appropriate event handler. Depending on the nature of the event, the scheduler can either ignore it or raise an exception (this is sometimes referred to as "throwing" an exception).

- Within an event-driven programming environment, standard events are usually identified using the ID of the object affected by the event (e.g. the name of a command button on a form), and the event ID (e.g. "left-click"). The information passed to the event-handler may include additional information, such as the x and y coordinates of the mouse pointer at the time the event occurred, or the state of the Shift key (if the event in question is a key-press).

- Events are often actions performed by the user during the execution of a program, but can also be messages generated by the operating system or another application, or an interrupt generated by a peripheral device or system hardware. If the user clicks on a button with the mouse or hits the Enter key, it generates an event. If a file download completes, it generates an event. And if there is a hardware or software error, it generates an event.

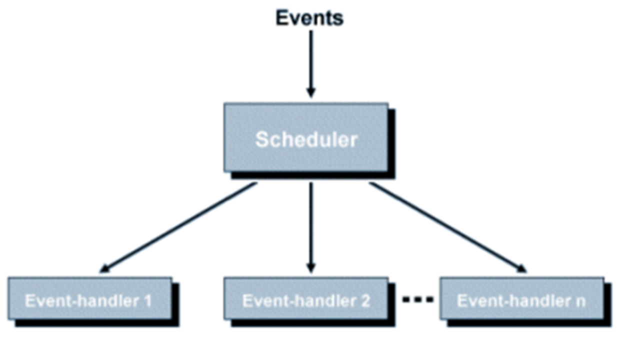

Figure 2: A simple event-driven programming paradigm

- The events are dealt with by a central event-handler (usually called a dispatcher or scheduler) that runs continuously in the background and waits for an even to occur. When an event does occur, the scheduler must determine the type of event and call the appropriate event-handler to deal with it. The information passed to the event handler by the scheduler will vary, but will include sufficient information to allow the event-handler to take any action necessary.
- Event-handlers can be seen as small blocks of procedural code that deal with a very specific occurrence. They will usually produce a visual response to inform or direct the user, and will often change the system's state. The state of the system encompasses both the data used by the system (e.g. the value stored in a database field), and the state of the user interface itself (for example, which on-screen object currently has the focus, or the background colour of a text box).
- An event handler may even trigger another event too occur that will cause a second event-handler to be called (note that care should be taken when writing event handlers that invoke other event-handlers, in order to avoid the possibility of putting the application into an infinite loop). Similarly, an event-handler may cause any queued events to be discarded (for example, when the user clicks on the Quit button to terminate the program).The diagram below illustrates the relationship between events, the scheduler, and the application's event-handlers.

.

Language for Paradigm 2: <Name of Language 2>

Discuss the characteristics and features of the language associated with Paradigm 2.

Analysis

1. **Functional Programming:**

   Features of Functional Programming:

   - Notable features of Functional Programming include (as explained in the previous sections):
     (a) Immutability
     (b) Pure Functions
     (c) First-Class and Higher-Order Functions
     (d) Referential Transparency
     (e) Function Composition
     (f) Recursion
     (g) Pattern Matching
     (h) Declarative Style
     (i) Parallel and Concurrent Programming

   Advantages of Functional Programming:

   (a) **Bugs-Free Code:** Functional programming does not support state, so there are no side-effect results and we can write error-free codes.

   (b) **Efficient Parallel Programming:** Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.

   (c) **Efficiency:** Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.

   (d) **Supports Nested Functions:** Functional programming supports Nested Functions.

   (e) **Lazy Evaluation:** Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

   Disadvantages of Functional Programming:

   (a) Limited Mutable State

   (b) Less Tooling and Libraries

2. **Event-Driven Programming:**

   Features of Event-Driven Programming:

- Event Driven Programming offers a wide array of features that enhance the performance, maintainability, and adaptability of software applications. These features are essential for handling events efficiently and ensuring seamless interaction between different components of a system.

(a) **Modularity and Reusability:**

  - Modularity and Reusability are two key features of Event Driven Programming that make it simple to construct, maintain, and enhance applications. In this paradigm, event handlers and other components are separated, allowing developers to compose their applications using well-defined modules. This separation between responsibility and implementation greatly reduces code duplication, as each module is only responsible for a specific task.
  - Developing modular code can result in numerous benefits, such as:
    - i. Improved code readability and maintainability.
    - ii. Minimised integration efforts
    - iii. Enhanced testing and debugging potential
    - iv. Reduced development time
    - v. Ability to reuse components across multiple projects
  - Reusability is another significant advantage of modular code in Event Driven Programming. Event handlers, utility functions, and other application components can be easily repurposed for various projects, simplifying the development process and resulting in a more consistent codebase. Reusability also boosts the efficiency of collaboration amongst developers, as a shared codebase allows them to quickly understand and analyze the source code.
  - Example: Imagine a software application with a menu system containing multiple buttons. Each button has a distinct action associated with it. With modularity and reusability, you can create multiple modules that handle the actions corresponding to the specific buttons instead of writing repetitive code in one large file.

(b) **Flexibility and Scalability:**

  - Flexibility and Scalability are two indispensable features of Event Driven Programming that allow applications to meet changing demands and grow over time. Flexibility implies that an application can modify its behavior or extend its functionality without requiring substantial code alterations. Scalability refers to an application's ability to maintain optimal performance as the workload increases or resources are added to the system.
  - Event-Driven Programming fosters flexibility by employing loosely coupled components and promoting modularity. This architecture ensures that modifying, extending, or replacing a single event handler does not impact other parts of the application. Thus, developers can readily adapt their software to new requirements, emerging technologies, or changing business environments.
  - Some of the advantages of flexibility in Event Driven Programming include:
    - i. Efficient adaptation to changing requirements or users' needs
    - ii. Swift incorporation of new functionality or third-party services

iii. Opportunities for continuous software improvement
iv. Decreased risk of system obsolescence

- Scalability is crucial for applications that must accommodate growing workloads or operate with variable resources. Event Driven Programming, particularly through its support of asynchronous and concurrent execution, enables software to efficiently utilise available resources, thereby maintaining consistent performance as system demands evolve.
- Some of the advantages of scalability in Event Driven Programming include:
  i. Efficient resource utilisation
  ii. Performance consistency under heavy workloads
  iii. Effective horizontal and vertical scaling
  iv. Ability to adapt to diverse deployment environments

(c) **Asynchronous and Non-Blocking:**

- In Event Driven Programming, asynchronous and non-blocking are crucial features that ensure high-performance and responsive applications. Asynchronous execution refers to the processing of tasks independently from the main control flow, enabling other operations to continue while a task is being executed. Non-blocking refers to the absence of blocking calls, which ensures that an application is not halted until a particular operation is completed.
- Asynchronous operations contribute to enhanced performance, particularly in resource-intensive or time-consuming tasks, as they permit other components of an application to progress concurrently. The non-blocking feature further ensures that the event loop remains uninterrupted, avoiding any bottleneck or potential freezing of the system.
- Example: Consider a web application that retrieves data from a remote server. If the operation blocks the main thread, the entire application becomes unresponsive until the data retrieval process is completed. With asynchronous and non-blocking execution, however, the application remains active, allowing users to interact with it while the data is being fetched.
- Implementing asynchronous and non-blocking operations within Event Driven Programming can offer numerous advantages, such as:
  i. Improved application responsiveness
  ii. Maximised system resource usage
  iii. Effective handling of concurrent tasks
  iv. Faster processing of background operations
  v. Real-time processing and data streaming
- In conclusion, Event-Driven Programming is characterized by its modularity and reusability, flexibility and scalability, as well as its capacity for asynchronous and non-blocking operations. These powerful features enable developers to build highly adaptable, maintainable, and responsive applications capable of addressing a wide range of use cases and requirements.

Advantages of Event-Driven Programming:

(a) **Responsive Applications:** Event Driven Programming allows applications to effectively respond to user input, resulting in a more dynamic and user-friendly experience. The event loop and event queue maintain the timely processing of events, ensuring that user interactions are effectively handled.

(b) **Concurrency:** Asynchronous event handling enables applications to execute multiple tasks concurrently. This capability can improve the overall performance and responsiveness of an application, particularly in situations where tasks are resource-intensive or time-consuming.

(c) **Modularity and Maintainability**: The separation of concerns in event-driven applications, through distinct event handlers and event management, promotes modularity and maintainability. Developers can focus on individual event handlers, making it simpler to comprehend, adjust, and augment the software.

(d) **Scalability:** Asynchronous event-driven architecture allows applications to efficiently utilize system resources, making it possible to scale both vertically and horizontally.

(e) **Real-Time Processing:** In the context of real-time applications, Event-Driven Programming enables the processing of events as they occur, ensuring the continuous distribution of up-to-date information and consistent system responsiveness.

(f) **Wide Range of Applications:** Event Driven Programming can be applied across diverse domains, including web applications, graphical user interfaces, server-side systems, and data-driven applications.

Disadvantages of Event-Driven Programming:

(a) **Complexity:** The asynchronous nature of event-driven applications can increase the software's complexity. Ensuring correct synchronization, managing race conditions, and addressing deadlock scenarios may necessitate extensive effort and precision in coding.

(b) **Debugging Difficulties:** Debugging event-driven applications can prove challenging, especially when dealing with concurrency, as the order of event execution is not predetermined and may vary during runtime. This unpredictability can complicate the process of identifying and resolving issues.

(c) **Event Handling Overhead:** The execution of event handlers and management of events demand additional system resources. Furthermore, the event loop and event queue require constant monitoring, potentially impacting performance.

(d) **Steep Learning Curve:** Developers who are unfamiliar with Event-Driven Programming may experience a steep learning curve, particularly when grappling with complex concurrency behaviors and synchronization.

(e) **Dependencies on External Libraries:** In certain programming languages and environments, Event-Driven Programming might rely on external libraries for managing events and handling asynchronous tasks. This dependence on external code may complicate deployment and maintenance.

1 Comparison

Similarities between Functional and Event-Driven Paradigms:

1. **Asynchronous Programming:** Both paradigms often involve asynchronous programming, allowing for the execution of tasks without waiting for the completion of prior operations. This promotes non-blocking behavior and responsiveness in applications.
2. **Declarative Style:** Both paradigms encourage a declarative style of programming. Developers focus on describing what should be done rather than specifying step-by-step procedures, leading to more readable and maintainable code.
3. **Statelessness:** Both paradigms emphasize statelessness to some extent. In functional programming, immutability ensures that functions do not modify external state, while in event-driven programming, components respond to events without relying on a shared state.
4. **Modularity and Reusability:** Both paradigms promote modularity and reusability. Functions in functional programming and event handlers in event-driven programming can be designed as independent, reusable components, enhancing code maintainability.
5. **Flexibility and Scalability:** Both paradigms offer flexibility and scalability. The ability to compose functions or handle events independently allows for the creation of scalable and modular applications that can be easily extended.

Differences between Functional and Event-Driven Paradigms:

1. **Core Focus:** Functional programming primarily revolves around the computation of mathematical functions and emphasizes immutability, while event-driven programming focuses on responding to and handling events triggered by user actions, system events, or messages.
2. **Data Flow:** In functional programming, data flows through a series of pure functions with well-defined inputs and outputs. In event-driven programming, data flow is often event-triggered, and components respond to events by executing specific handlers.
3. **Handling of State:** Functional programming relies on immutability and avoids changing state. In contrast, event-driven programming involves managing state changes triggered by events, allowing components to respond dynamically to the system's or user's actions.
4. **Control Flow:** Functional programming often follows a deterministic control flow, where the order of execution is explicitly defined. Event-driven programming, on the other hand, relies on events to dictate the flow of control, making it event-triggered and sometimes asynchronous.
5. **Concurrency Model:** Functional programming, with its emphasis on immutability and pure functions, provides a strong foundation for concurrent programming. Event-driven programming, with its focus on responsiveness, is well-suited for managing multiple asynchronous events concurrently.
6. **Use Cases:** Functional programming is commonly applied in scenarios where mathematical precision and data transformations are crucial, such as data processing and algorithmic computations. Event-driven programming is often used in user interfaces, real-time systems, and scenarios where responsiveness to events is essential.

2 Challenges Faced

In this section, we explore the challenges encountered during this study, specifically focusing on the Event-Driven paradigm in TypeScript and the Functional paradigm in Clojure.

A significant challenge arose from the lack of clarity in available resources related to the Event-Driven paradigm in TypeScript. Many articles were excessively analytical and seemed to go beyond the scope of this study. Determining the relevance of these articles posed a challenge, prompting the need to filter out the most pertinent information. To overcome this, the study predominantly relied on information sourced from reputable platforms such as the TypeScript documentation and practical examples shared within TypeScript communities. Relying on reliable sources and practical use cases facilitated the identification of relevant content and ensured the accuracy of the study.

Similarly, challenges were encountered in establishing meaningful connections between the Functional paradigm in Clojure and related resources. While numerous materials discussed the differences between functional programming and other paradigms, finding content that specifically highlighted the similarities between Clojure's Functional paradigm and other programming approaches proved to be challenging. This obstacle was effectively addressed by leveraging ChatGPT, which provided valuable insights and assistance in bridging gaps within the existing literature.

Additionally, obtaining comprehensive information on the Functional paradigm in Clojure faced challenges due to the extensive focus on TypeScript and the Event-Driven paradigm in this study. The wealth of information to be conveyed about TypeScript and Event-Driven programming led to a relatively less comprehensive report on the Functional paradigm in Clojure. Nevertheless, dedicated efforts were made to explore articles, content, and examples, ensuring a comprehensive understanding of the Functional paradigm and its implementation in Clojure.

In both instances, ChatGPT played a crucial role in supplementing the study by providing additional information, clarifications, and insights, particularly where existing resources were insufficient or unclear.

Conclusion

A comprehensive examination of the two paradigms and the features inherent to their respective languages was conducted. The Event-Driven paradigm was explored, highlighting its distinctive features along with the advantages and disadvantages associated with each. The report delved into the implementation of these features in TypeScript, providing insights into how they are manifested and discussing the pros and cons of their utilization. Real-time use cases of TypeScript in the context of the Event-Driven paradigm were briefly examined.

Similarly, the Functional paradigm was investigated, shedding light on its core features and illustrating them through examples. The report outlined the advantages and disadvantages of the Functional paradigm, offering insights into the workings of the Clojure programming language within this paradigm. Real-time applications of Clojure were explored and discussed briefly.

A comparative analysis between the two paradigms and their respective languages was presented, elucidating their differences and similarities, as well as the advantages, disadvantages, and real-time use cases.

3 References

Include any references or sources you consulted for your assignment.

1. https://www.geeksforgeeks.org/functional-programming-paradigm/
2. https://en.wikipedia.org/wiki/Functional_programming
3. https://www.geeksforgeeks.org/introduction-of-programming-paradigms/
4. https://link.springer.com/chapter/10.1007/978-3-031-34144-1_11
5. https://homes.cs.aau.dk/ normark/prog3-03/html/notes/paradigms-book.html
6. https://chat.openai.com/chat
7. https://www.studysmarter.co.uk/explanations/computer-science/computer-programming/event-driven-programming/