Comparison of Methods Used

Author: Sigal Lev
Team: ASDs

Problem A. Key Word in Context (KWIC)

Method 4. Implicit Invocation (Event-Driven)

- **Implementation Changes:** Event-driven design is highly flexible; each listener or observer handles events independently. For example, changing the sorting logic affects only the listener responsible for sorting, without impacting other components.
- **Data Representation Changes:** This approach provides moderate flexibility in changing data representation, as each observer interacts with data via events, not directly. However, adjustments to event propagation may be needed.
- **Adding Functions:** Adding new functions is straightforward; you can add new observers to listen for specific events. For example, adding an "export index to file" event requires only a new observer for that event.
- **Performance:** Event-driven structures may be less performant in single-threaded environments due to event overhead, but they perform well in multi-threaded environments with asynchronous processing for large datasets.
- **Reusability:** This solution is highly reusable, as the observer pattern allows each component to be used in other contexts. This approach is ideal for systems that require flexibility and expandability.

Method 1. Abstract Data Types (ADT)

- **Implementation Changes:** Since data and functions are encapsulated, implementation changes are isolated in each class. For instance, changing how KWIC lines are formatted or stored is straightforward, as each class can be modified independently.
- **Data Representation Changes:** ADT design makes it easy to alter data representations, like switching the structure for storing titles, without affecting other components.

- **Adding Functions:** Adding functions, such as filtering keywords or exporting indexes, is simple. These can be added as new methods to classes without affecting existing ones.
- **Performance:** Performance is generally good as ADTs avoid shared state and encapsulate data with its processing functions. However, as datasets grow, performance might be affected by increased object management.
- **Reusability:** This approach is modular and reusable, particularly suited for text processing tasks, as it enforces clear responsibilities across objects.

Method 3. Pipes and Filters

- **Implementation Changes:** Each module operates independently, relying solely on inputs/outputs, which simplifies changing algorithms in any module without affecting others.
- **Data Representation Changes:** Each module processes only its input and generates standardized output, making it easy to change internal data representations while maintaining consistent input/output formats.
- **Adding Functions:** New modules can be added to introduce additional functions without disrupting existing ones.
- **Performance:** Data transfer between modules can add overhead, especially in asynchronous setups. Performance varies based on inter-process communication methods.
- **Reusability:** Each module is self-contained, making it easy to reuse independently or with minimal adaptation, which is advantageous in modular and parallel processing systems.

Comparison of Methods for Problem A

Criteria	Method 4. Implicit Invocation (Event-Driven)	Method 1. Abstract Data Types (ADT)	Method 3. Pipes and Filters
Implementation Changes	High flexibility due to independent listeners	Easy to change in isolated modules	Very easy; independent modules
Data Representation Changes	Moderate, with minor event adjustments	Moderate, due to encapsulation	Easy; independent data in each module

Adding Functions	Very easy; add new listeners	Straightforward, add as methods	Very easy; add as new modules
Performance	Moderate, best in multi-threaded environments	Good	Moderate, data transfer overhead
Reusability	Very high	High	Very high

Conclusion:

- **Most Flexible:** Event-Driven approach, especially useful for future system expansions or modifications.
- **Ideal for Small-Scale Tasks:** ADT, suitable for well-defined problems with minimal expansion needs.
- **Best for Modular Systems:** Pipes and Filters, particularly beneficial if the system requires processing large amounts of data in parallel or independently.

Problem B. Eight Queens (8Q)

Method 1. Abstract Data Types (ADT)

- **Implementation Changes:** The ADT structure encapsulates algorithms within methods, allowing changes to be made by modifying only the relevant methods. However, high dependencies between classes can increase complexity.
- **Data Representation Changes:** Since data is encapsulated within classes, internal representations (e.g., representing the board as a 2D array or a list of positions) can be changed without affecting other system parts.
- **Adding Functions:** New functions, such as visualization or analysis tools, can be added as methods to existing classes, keeping functionality centralized and manageable.
- **Performance:** Performance is generally acceptable, though emphasis on modularity and encapsulation might add minor overhead, especially with complex object interactions.
- **Reusability:** The modular structure makes it easy to reuse the Queen and Chessboard classes in similar problems or in variants like N-Queens for different board sizes.

Method 2. Main/Subroutine with Stepwise Refinement (also Shared Data)

- **Implementation Changes:** Each subroutine can be independently refined or replaced since this method relies on procedural decomposition. Changing one part of the algorithm (e.g., the IsSafe check) usually doesn't impact others.
- **Data Representation Changes:** If the data is shared across subroutines, changing the data structure (e.g., from an array to a list of coordinates) might require updates across multiple subroutines.
- **Adding Functions:** Additional functions can be added as new subroutines, though shared data might complicate the interaction between them.
- **Performance:** This approach tends to be efficient with minimal encapsulation overhead and direct function calls.
- **Reusability:** Subroutines are reusable, but they may depend on specific data structures or calling conventions, making it harder to adapt for different problems.

Method 4. Implicit Invocation (Event-Driven)

- **Implementation Changes:** High flexibility, as each event is processed independently by listeners.
- **Data Representation Changes:** Moderately flexible for data structure changes, as components interact via events rather than direct data access.
- **Adding Functions:** Adding new functions is easy; simply introduce new events or listeners.
- **Performance:** Performance is moderate, with improvements in multi-threaded environments due to asynchronous event processing.
- **Reusability:** Highly reusable due to the observer pattern, allowing components to be adapted to other contexts.

Comparison of Methods for Problem B

Criteria	Method 1. Abstract Data Types (ADT)	Method 2. Main/Subroutine with Stepwise Refinement	Method 4. Implicit Invocation (Event-Driven)
Implementation Changes	Easy; isolated modules	Moderate; shared data makes changes harder	Flexible due to independent events

Data Representation Changes	Moderate; encapsulated classes	Difficult; requires global changes	Moderate; some event adjustments needed
Adding Functions	Straightforward; add methods	Moderate; new routines may require data access	Very easy; add new events
Performance	Good; slight encapsulation overhead	High; efficient with direct calls	Moderate; best in multi-threaded environments
Reusability	High; modular classes	Moderate; routines depend on shared data	High

Conclusion:

- **Easiest to Change Algorithm:** Event-Driven, as each component is independent and can be modified without affecting others.
- **Easiest to Change Data Representation:** ADT, as encapsulation localizes changes.
- **Most Performant Solution:** Main/Subroutine, due to minimal abstraction and direct function calls.
- **Preferred Solution for Reusability:** ADT, as it balances modularity with performance and is suitable for complex interactions like managing a chessboard and queens.