The result of comparing the methods used

Author: Derezovskiy Ilya
Team: ASDs

Problem A. Key Word in Context (KWIC)

1. Abstract Data Types (ADT)

- **Implementation Changes**: ADTs allow isolated changes in each class since data and functions are encapsulated. If we need to change how titles are stored or KWIC lines are formatted, we can modify each class independently without affecting others.
- **Data Representation Changes**: ADT design makes it relatively easy to alter data representations, like changing `Title`'s data structure from a list to a dictionary, without affecting other parts.
- **Adding Functions**: Adding functions like filtering specific keywords or exporting indexes is straightforward, as these can be added as new methods to the classes without affecting existing ones.
- **Performance**: Performance is generally good, as ADTs avoid shared states and encapsulate data closely with its processing functions. Performance might decrease with a large dataset, as we still need to manage each instance.
- **Reusability**: This approach is modular and reusable, particularly suited for similar text processing or indexing tasks, since it enforces clear responsibilities across objects.

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

- **Implementation Changes**: Changes to the implementation (e.g., how sorting is done) require modifications across multiple subroutines since each operates on the shared data, making this approach less flexible.
- **Data Representation Changes**: Changing the data representation is more challenging, as the shared data structure is accessed by many routines. Every routine would need to adapt to the new data format.
- **Adding Functions**: Adding new functions is straightforward in a stepwise refinement structure. We could introduce new subroutines without altering the main structure, as long as they work with the shared data.
- **Performance**: Shared data structures reduce memory overhead since data is accessible by all subroutines. However, this can also lead to bottlenecks and synchronization issues if we add concurrency for large data processing.
- **Reusability**: This structure is less reusable since routines are tightly coupled to the shared data. Repurposing or extending for other contexts would require significant changes to both the data structure and the flow.

4. Implicit Invocation (Event-Driven)

- **Implementation Changes**: Event-driven design is flexible for implementing changes, as each listener or observer responds independently to events. For instance, changing sorting logic only affects the listener that handles the sorting event, so it can be changed without impacting other components.
- **Data Representation Changes**: This approach is moderately flexible for changing data representation, as each observer interacts with the data through events, rather than directly. The main challenge is that event propagation may need adjustments.

- **Adding Functions**: It's easy to add functions in an event-driven setup by adding new observers or listeners for new events. For example, adding an event for "exporting index to file" would only need a new observer to listen to the relevant event.
- **Performance**: Event-driven structures may be less performant in single-threaded environments due to event overhead. However, they can perform well in multi-threaded setups, allowing asynchronous processing for potentially large datasets.
- **Reusability**: This solution is highly reusable, as the observer pattern allows each component to be used in other contexts. This is ideal for systems requiring flexibility and expandability.

Problem A. Key Word in Context (KWIC)			
	Method 1. Abstract Data Types	**Method 2.** Main/Subroutine with stepwise refinement	**Method 4.** Implicit invocation (event-driven)
a) Implementation Changes	Easy to change in isolated modules	Harder, as modules share data	Flexible due to independent listeners
b) Data Representation Changes	Moderate, due to encapsulation	Difficult, as it requires global changes	Moderate, with minor event adjustments
c) Adding Functions	Straightforward, add as methods	Easy, but may require shared data access	Very easy, add as new listeners
d) Performance	Good	Good, but may have sync issues	Moderate, potentially high in async
e) Reusability	High	Low	Very High

Conclusion

- **Most Flexible**: **Event-Driven** design is the most flexible, especially if the system may need future expansion or modifications.
- **Best for Small-Scale Tasks**: **ADT** is ideal for well-defined problems with minimal expansion needs, offering encapsulation and modularity.
- **Best for Simple, Stepwise Workflows**: **Main/Subroutine** is more suited for simple applications without high modification or reuse requirements, as it involves more coupling between data and routines.
- If tasked with implementing a similar program, I would choose the **Event-Driven (Implicit Invocation)** approach. It balances flexibility and modularity, allowing easy modifications, additions, and integration with other components, which is especially useful for a KWIC system that may require adding new functionalities (e.g., advanced search features, analytics) in the future.

Problem B. Eight Queens (8Q)

1. Abstract Data Types (ADT)

- **Implementation Changes**: The ADT structure can encapsulate algorithms within methods, so changing them requires modifying only the relevant methods. However, if dependencies between classes are high, this can increase complexity.
- **Data Representation Changes**: Since the data is encapsulated in classes, it's easier to modify internal representations (e.g., board as a 2D array or list of positions) without impacting other parts of the system.
- **Adding Functions**: New functions (like visualization or analysis tools) can be added as methods to existing classes, which keeps functionality centralized and manageable.
- **Performance**: Performance is generally acceptable, but since ADT emphasizes modularity and encapsulation, it might add slight overhead, especially with complex interactions between objects.
- **Reusability**: The modular structure makes it easy to reuse the Queen and Chessboard classes in similar problems, or in variations like N-Queens for different board sizes.

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

- **Implementation Changes**: Each subroutine can be independently refined or replaced, as the method relies on procedural decomposition. Changing one part of the algorithm (e.g., the IsSafe check) doesn't typically 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) could require adjustments across multiple subroutines.
- **Adding Functions**: Additional functions can be added as new subroutines. However, if these new functions require shared data, the interactions between subroutines can become complex.
- **Performance**: This approach tends to be efficient, as there's minimal encapsulation overhead, and function calls are often direct. It's straightforward and close to the system level.
- **Reusability**: Subroutines are reusable, but they may depend on specific data structures or calling conventions. Reusing this structure in similar problems might require more adaptation than ADTs.

3. Pipes-and-Filters

- **Implementation Changes**: Each module is isolated and only depends on data inputs/outputs, making it relatively easy to change the internal algorithm of any module without affecting others.
- **Data Representation Changes**: Since each module operates on its input and produces a standardized output, data representation changes are isolated within each module, as long as the input/output format remains consistent.
- **Adding Functions**: New modules can be added to the pipeline to introduce new functionality without disrupting existing modules. This extensibility is a key advantage of the Pipes-and-Filters model.
- **Performance**: The overhead of data transfer between modules (especially if done asynchronously) can reduce performance. Performance may also vary based on the inter-process communication methods.

- **Reusability**: Each module is self-contained, making it easy to reuse independently or with minor adaptations. This approach is particularly suitable for systems that require modular and parallel processing.

Problem B. Eight Queens (8Q)			
	Method 1. Abstract Data Types	**Method 2.** Main/Subroutine with stepwise refinement	**Method 3.** Pipes-and-Filters
a) Implementation Changes	Easy; isolated module updates	Moderate; shared data makes changes harder	Very easy; independent modules
b) Data Representation Changes	Moderate; encapsulated classes simplify	Difficult; global data changes required	Easy; independent data in each module
c) Adding Functions	Straightforward; add methods to classes	Moderate; new routines may need data access	Very easy; add as new modules
d) Performance	Good; slight encapsulation overhead	High; efficient with direct calls	Moderate; overhead from data transfer
e) Reusability	High; modular classes	Moderate; routines depend on shared data	Very high; reusable independent modules

Conclusion

a) **Easiest to change implementation algorithm**: Pipes-and-Filters, as each module is independent and changes do not impact others.

b) **Easiest to change data representation**: Pipes-and-Filters, since each module can adjust its data independently with minimal overall impact.

c) **Easiest to add additional functions**: Pipes-and-Filters, allowing new modules to be added seamlessly.

d) **Most performant solution**: Main/Subroutine, due to minimal abstraction and direct function calls.

e) **Preferred solution for reuse**: If I were to implement a similar program, I would likely choose **Abstract Data Types (ADT)** because:

- It balances modularity with performance, and its encapsulation of both data and methods suits complex data interactions like the chessboard and queens.
- ADTs provide a clean structure for representing objects in object-oriented languages, which can simplify the logic for a problem like Eight Queens.
- For systems requiring high modularity and flexibility in data flow, the Pipes-and-Filters approach is also appealing, particularly for scalable applications or environments where functions might need to run in parallel or as microservices. However, for a single-player, computationally-focused problem like Eight Queens, ADTs strike a good balance between design elegance and simplicity.