Modern Intelligence Take Home

Part 1: Design

Merging Two Tracks

Name	Track #1	Track #2	Merged
track_id	8655725a	99c1bcc7	8655725a (Track #1)
callsign	"VESSEL1"	"TARGET1"	"VESSEL1" (Track #1)
start_time	1	2	1 (Track #1)
observation_time	9	10	10 (Track #2)
latitude	74.1200002	74.1200003	74.1200003 (Track #2)
longitude	33.4500007	33.4500006	33.4500006 (Track #2)

Merging Design

1. **Track ID Preservation**: The merged track inherits the track_id of the initially created track. This ensures consistency; once a track is created, it either merges into an existing track or serves as a base for future merges.
2. **Callsign Stability**: The callsign of the merged track is retained from the initially created track. This approach prevents frequent changes in callsigns due to merging and ensures that any user-defined callsigns remain unchanged, enhancing stability and persistence.
3. **Start Time Integrity**: The start time of the merged track reflects the earliest start time among all the tracks involved in the merge. This policy maintains the historical integrity of the track by preserving its oldest known start time.
4. **Observation Time Updating**: The observation time for the merged track is sourced from the most recent track. This ensures that the merged track's observation time is always current, reflecting the latest data.
5. **Location Accuracy**: Latitude and longitude for the merged track are taken from the most recent track. In the absence of explicit accuracy indicators, this strategy assumes that the latest location data is the most accurate, optimizing the merged track's positional accuracy.

Distance Calculation

- **Euclidean Distance**: Euclidean distance is a measure used in geometry to determine the shortest straight-line distance between two points in a flat, two-dimensional plane. When applied to geographical locations, it simplistically treats the Earth as flat, calculating the distance based on the Pythagorean theorem. The formula sqrt((lat1 - lat2)^2 + (lon1 - lon2)^2) computes this distance using the differences between two points. While this method is computationally straightforward and useful for short distances or theoretical discussions, its accuracy diminishes over long distances due to Earth's curvature.
- **Great Circle Distance**: Great circle distance is a measure that calculates the shortest path between two points on the surface of a sphere. It assumes the Earth is a perfect sphere, which allows it to provide more accurate distances than the Euclidean method for locations that are far apart. This calculation is based on spherical trigonometry, where the great circle

that passes through both points on the globe is considered, and the arc of that circle connecting the points represents the shortest distance. This method is widely used in aviation and maritime navigation due to its relative accuracy and simplicity, despite the Earth's slight equatorial bulge. I've added this method because it serves a great advantage in mid-range calculations where Geodetic calculations would be to cumbersome.

- **Geodetic Distance**: Geodetic distance accounts for the Earth's true shape more accurately by considering it as an oblate spheroid—a sphere flattened along the axis from pole to pole. This method involves complex calculations that consider the ellipsoidal shape of the Earth, allowing for precise distance measurements across its surface. The geodetic distance is calculated using the WGS-84 (World Geodetic System 1984) model or similar ellipsoidal models of Earth, making it the most accurate for real-world applications, especially for long distances and precision-required tasks in geodesy, surveying, and navigation. This approach reflects the Earth's irregular shape, providing superior accuracy over the great circle method for professional and scientific purposes.

	Euclidean	**Great Circle**	**Geodesic**
Divergence > 1 m	@9.168 m	@4890.76 m	-------------------------
Time to Execute 100,000 Calculations	0.0182 sec	0.5616 sec	4.9187 sec

For distances under 9 meters, where precision isn't a critical factor, the Euclidean distance calculation is adequately sufficient. This method provides a straightforward approach for short distances, making it a practical choice for applications where exact accuracy is not essential. For intermediate distances, ranging from 9 to 4,890 meters, employing the Great Circle method yields the most effective results. This method offers a balance between computational simplicity and improved accuracy over the Euclidean approach, making it ideal for most practical navigation and distance measurement tasks over these scales. Beyond 4,890 meters, the Geodetic distance calculation emerges as the superior option.

Part 2: Code Design

In the process of designing this software package, I meticulously focused on two critical dimensions: the functional integrity of the code and its adaptability to future changes. Recognizing the elements within the code that may need to evolve allowed the codebase to grow and adapt organically, ensuring it remains manageable for developers over time. To facilitate this adaptability, I strategically implemented abstraction, dependency injection, and polymorphism. Here's how these principles were applied to address specific architectural challenges:

1. **UUID and Time Generation**: With the expansion of datasets and their distribution across systems, it's vital to have a flexible mechanism for assigning unique identifiers and timestamps. While simple UUIDs and local timestamps suffice for small-scale, monolithic systems, distributed environments require a more sophisticated approach to avoid conflicts and synchronize time across nodes. To address this, I employed abstract classes for identifier and timestamp generation, coupled with a builder pattern to seamlessly

support dependency injection, thereby accommodating changes in identifier and timestamp generation strategies without disrupting existing systems.

2. **Distance Calculation**: Calculating distances can range from straightforward point-to-point measurements to complex computations considering various error margins and data freshness. To accommodate this range of complexity, I used an abstract class for distance calculation, allowing for the dynamic injection of different calculation strategies via a builder class. This design ensures our system can adapt to varying precision requirements and data characteristics.
3. **Interface Stability**: When developing a library or package intended for widespread use across an architecture, maintaining stable and reliable interfaces is paramount. This stability ensures that dependent modules can reliably interact with the library without concern for breaking changes. Through careful abstraction and inheritance, I ensured that these interfaces remain consistent and robust against changes in the underlying implementation.
4. **Data Object Stability**: As the system evolves, so will the structure and nature of the data it processes. Anticipating and planning for this evolution is crucial. By leveraging generics and abstraction, I designed our data handling mechanisms to be inherently flexible, enabling the integration of new data types and structures with minimal effort and no need for overhaul.
5. **Fusion**: The system's requirements and inputs will inevitably change, necessitating a flexible approach to how data is merged or "fused." To this end, I decoupled the fusion logic from the core tracking functionality, allowing for the dynamic injection of different ‘FusibleCollections’. This separation ensures that the tracking component remains agnostic to the specifics of the fusion process, allowing for easy adaptation to new fusion methods as requirements evolve.

In summary, my architectural decisions were guided by a forward-looking approach, balancing the immediate needs of the software with the inevitability of change. By embracing principles like abstraction, dependency injection, and polymorphism, I crafted a solution that not only meets current technical and business requirements but is also poised to adapt to future challenges.

Geo Hashing

The time complexity of my 'Nearest Neighbor' fusion method led me to explore more efficient alternatives. Hash maps can typically reduce lookup time complexity to O(1) and, at worst, O(n). Interested in merging latitude and longitude into a single hashable key for faster searches, I discovered the pygeohash package. I felt this was an interesting endeavor and would properly illustrate the flexibility of the tracker package.

Geo Hashing: A spatial indexing strategy that converts geographical coordinates (latitude and longitude) into a compact string using base32 encoding. This method facilitates efficient storage and retrieval of location data. The essence of Geo Hashing lies in its precision-based approach to spatial representation. Each Geo Hash encodes a rectangular area on the Earth's surface, known as a bounding box. The length of the Geo Hash determines its precision; longer hashes represent smaller areas. This granularity allows users to control the balance between precision and spatial coverage, tailoring it to specific requirements.

Findings:

- Pros:
    1. Nearly 5000 times faster than the Nearest Neighbor implementation on a 10,000-element dataset.
    2. Simplification of search algorithm.
    3. Flexibility in sizing of search area.
- Cons:
    1. Lack of resolution: a 9-character hash has a maximum error of 6.8 meters; 8 characters has a maximum error of 42 m.
    2. Potential to not fuse tracks on the edges of adjacent bounding box hashes.

Code Improvements

1. **Refinement of the Ping Class**: Elevate the Ping class to serve as a Data Transfer Object (DTO). By structuring the Ping class as a DTO, we can streamline the transfer of data across different layers or services within the application. This approach facilitates efficient, transport-agnostic operations and enhances decoupling, making the system more modular and easier to maintain.

2. **Decomposition of FusibleCollections Responsibilities**: The current implementation of FusibleCollections encompasses multiple responsibilities and does not abide by Single Responsibility Principle (SRP). It's advisable to refactor FusibleCollections by breaking it down into smaller, more focused components. Each component should address a single aspect of functionality, improving code clarity, ease of testing, and adherence to SRP.

3. **Expansion of Test Coverage**: Develop a comprehensive suite of test cases and data sets that more accurately reflect real-world scenarios. This expansion is crucial for evaluating the system's efficiency and fault tolerance. Implementing a wider range of tests, including edge cases and stress tests, will help identify potential bottlenecks or vulnerabilities, ensuring the system is robust and reliable under various conditions.