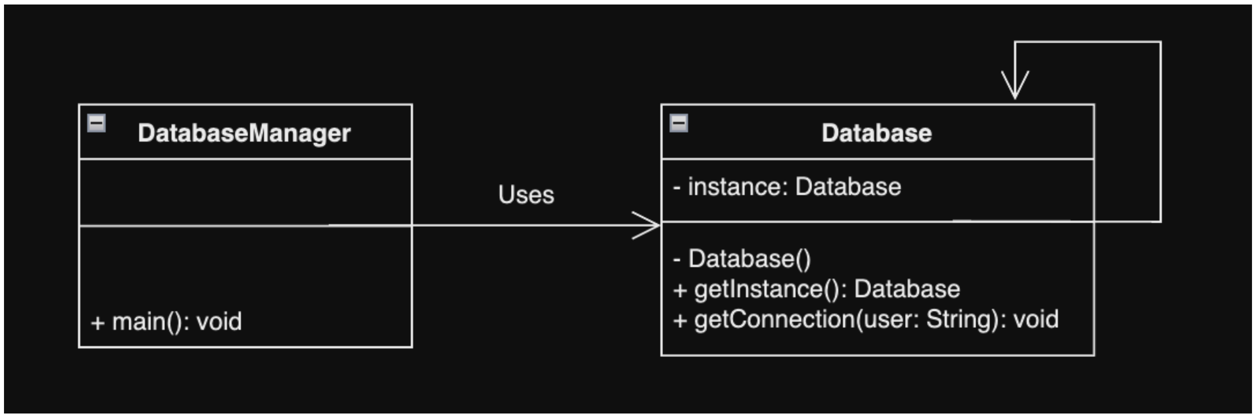
ANSWER 1.

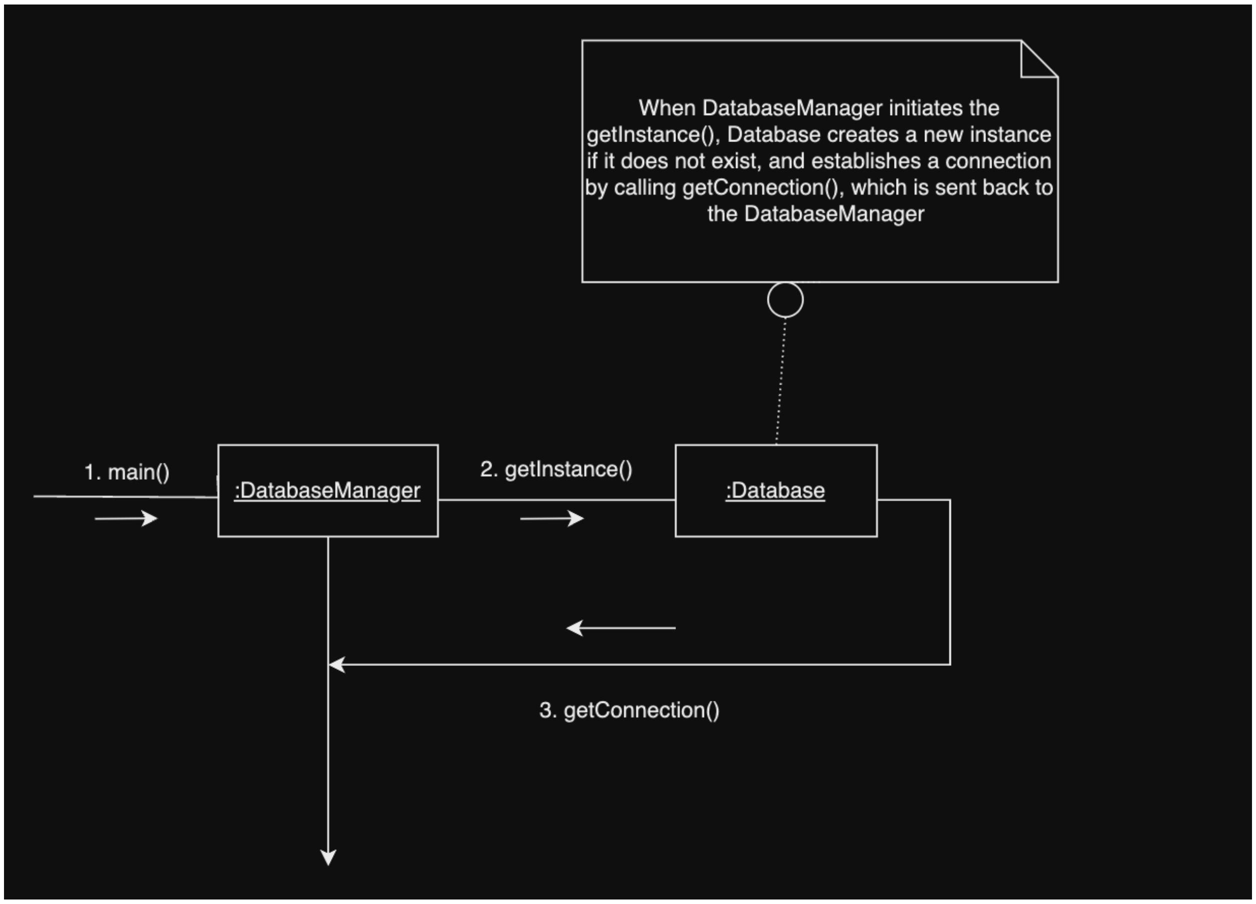
A compelling application for the Singleton pattern is to manage the database connection. The system must ensure that the application has only one instance of the database connection object throughout its lifetime. Applying this pattern ensures that all the parts of the application interact with the database using the same connection instance.

The reason for applying the Singleton pattern to the database connection is because:
1. ***Resource Efficiency:*** Unnecessary resources, such as CPU and memory cycles are consumed by creating and managing multiple database connections. It can be ensured that only one database connection instance is created and reused throughout the application's lifecycle, optimizing resource usage by using the Singleton pattern.
2. ***Consistency:*** A Singleton database connection guarantees that all parts of the application interact with the database utilizing the same connection instance. This helps maintain data consistency and avoids potential issues that may arise from concurrent access to multiple database connections.
3. ***Centralized Management:*** Having a single database connection instance allows for centralized management of database-related operations such as transaction management and error handling. This simplifies the maintenance and troubleshooting of database-related functionality in the application.

CLASS DIAGRAM :



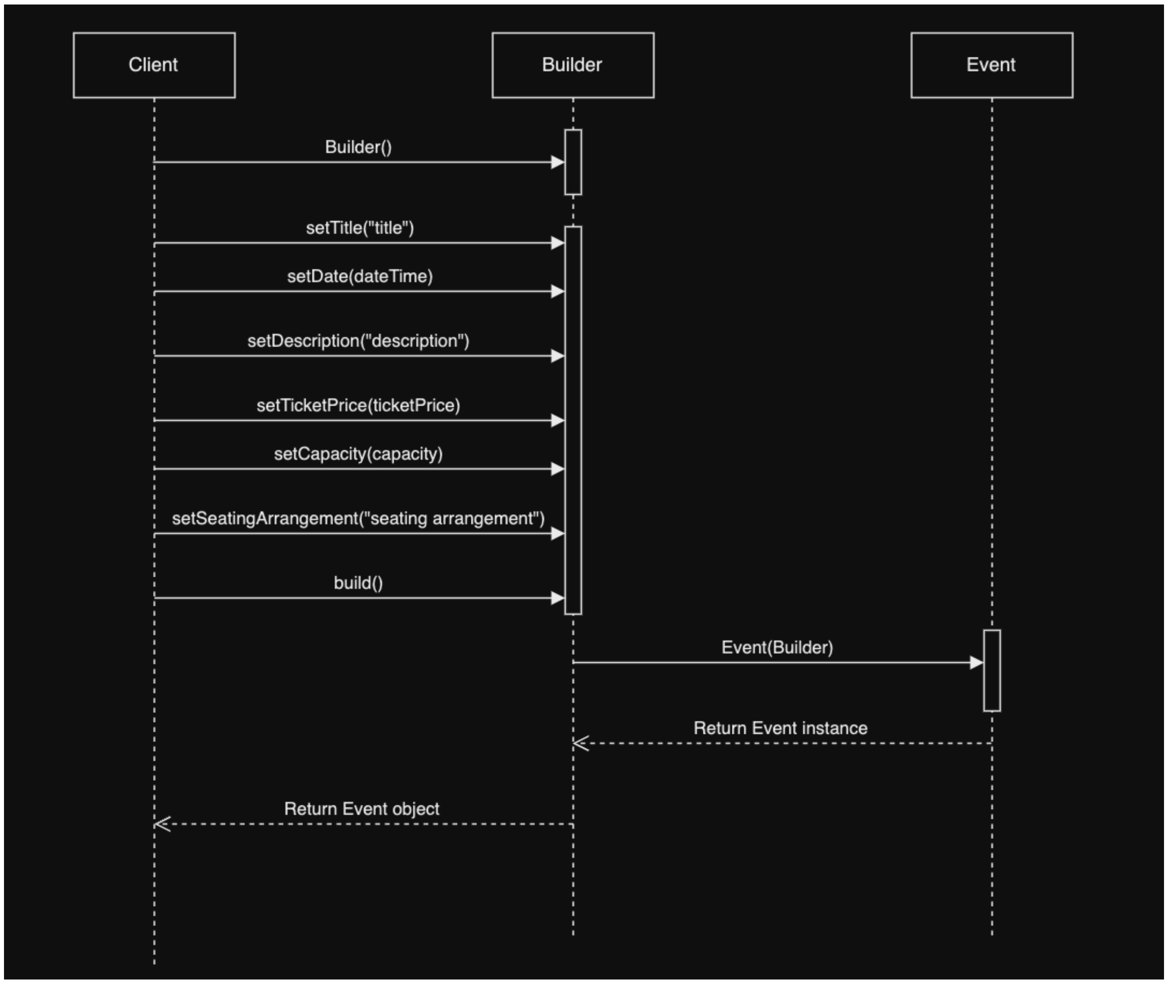
COLLABORATION DIAGRAM:


When DatabaseManager initiates the getInstance(), Database creates a new instance if it does not exist, and establishes a connection by calling getConnection(), which is sent back to the DatabaseManager
1. main()
:DatabaseManager
2. getInstance()
:Database
3. getConnection()

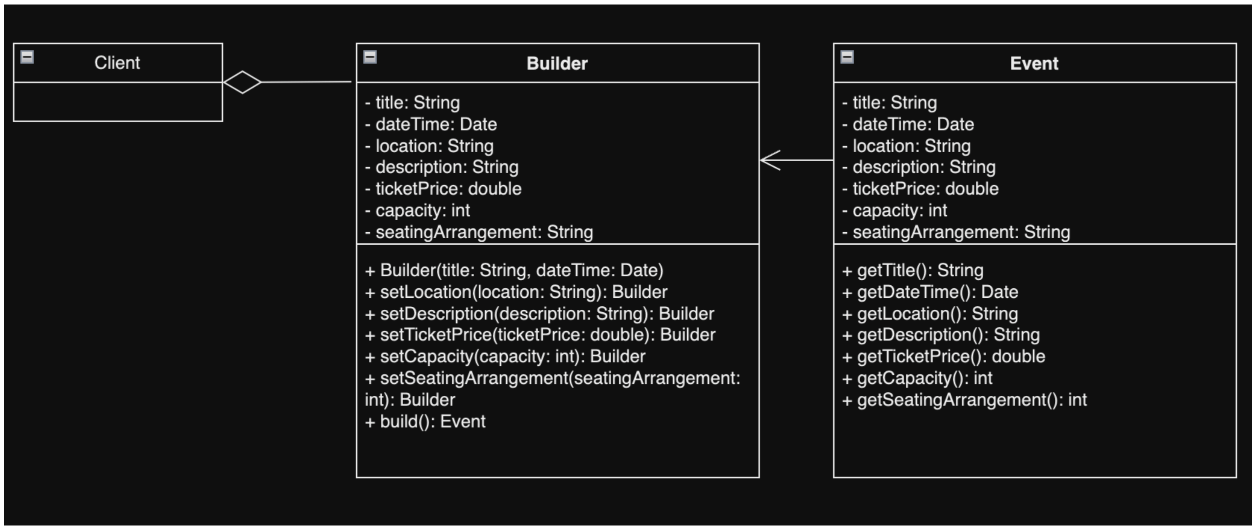
ANSWER 2:

A compelling scenario where one can apply Joshua Bloch's Builder design pattern to the XYZ system is for creating complex event objects with diverse optional parameters. In the XYZ system, events may have a variety of attributes such as title, date, time, location, description, ticket price, capacity, category, and more. Utilizing the Builder pattern, the system can provide a flexible and fluent interface for constructing event objects with distinct combinations of these attributes.
In the XYZ system, clients may want to create events with different attributes. Some events may include only basic information, while others may include additional details such as ticket price, capacity, or category. With the help of the Builder pattern, the system can provide a flexible way to construct event objects with different sets of attributes.

SEQUENCE DIAGRAM:



CLASS DIAGRAM:


Client
Builder
- title: String
- dateTime: Date
- location: String
- description: String
- ticketPrice: double
- capacity: int
- seatingArrangement: String
+ Builder(title: String, dateTime: Date)
+ setLocation(location: String): Builder
+ setDescription(description: String): Builder
+ setTicketPrice(ticketPrice: double): Builder
+ setCapacity(capacity: int): Builder
+ setSeatingArrangement(seatingArrangement: int): Builder
+ build(): Event
Event
- title: String
- dateTime: Date
- location: String
- description: String
- ticketPrice: double
- capacity: int
- seatingArrangement: String
+ getTitle(): String
+ getDateTime(): Date
+ getLocation(): String
+ getDescription(): String
+ getTicketPrice(): double
+ getCapacity(): int
+ getSeatingArrangement(): int