Our strategy is built around the following key concepts:

* **Defensive Programming:**  
  All input is treated as untrusted. The system validates input immediately upon receipt to prevent invalid data from propagating into the core logic.
* **Early Validation:**  
  Input is checked at the earliest point possible (in the *InputValidator* class). This includes verifying the format, ensuring the correct number of tokens, and validating that the date values are within the allowed range.
* **Exception Handling:**  
  Custom exceptions (specifically, an *InvalidInputException*) are used to signal validation errors. Localized error handling in individual routines ensures that specific issues are caught and reported with meaningful messages.
* **Graceful Degradation:**  
  In the event of an error, the system provides clear error messages and terminates gracefully rather than crashing unexpectedly.

**Local Error Handling**

* **Scope:**  
  Local error handling is performed within the modules that directly process input and perform validation.
* **Implementation:**
  + The InputValidator class contains methods (e.g., readNumberOfEvents and parseEvent) that immediately validate input.
  + When an error is detected (e.g., non-numeric input, invalid date ranges, or insufficient tokens), these methods throw an InvalidInputException with a descriptive error message.
* **Benefits:**  
  This approach confines error detection to the point where it occurs, making debugging easier and ensuring that only valid, sanitized data is passed to the core logic.

**Global Error Handling**

* **Scope:**  
  Global error handling is implemented in the main method of the application.
* **Implementation:**
  + The main method wraps the input and processing logic in a try-catch block to capture exceptions that bubble up from the lower layers.
  + This ensures that any unhandled or unexpected errors are caught at the top level, allowing the system to terminate gracefully with a clear error message.
* **Benefits:**  
  Global error handling acts as a safety net for any issues that were not handled locally, ensuring overall system stability.

The design incorporates a barricade layer to separate untrusted input from internal logic:

**Untrusted Input Layer**

* **Description:**  
  This layer represents the raw text input provided by the user (e.g., through the console or a file).
* **Risk:**  
  Since the input is untrusted, it may contain typos, formatting errors, or invalid data.

**Validation (Barricade) Layer**

* **Component:**  
  The InputValidator class acts as the barricade. It sanitizes and validates all incoming data before it is passed on.
* **Function:**
  + It ensures that the first line contains a valid integer for the number of events.
  + It validates each event line to ensure that it contains exactly two integers and that these integers are within the valid range (1–366) with the start date not exceeding the end date.
* **Outcome:**  
  Only well-formed and verified data (trusted data) is passed on to the internal processing components.

**Internal (Trusted) Layer**

* **Description:**  
  This layer consists of the core logic of the system, such as the ReservationSystem class, the Event class, and the overlap checking function.
* **Assumption:**  
  Because all input has been validated by the barricade, this layer can operate under the assumption that the data is correct, thus simplifying its logic and reducing complexity.

In addition to the above architecture, the following factors from the defensive programming checklist have been incorporated:

* **Low Routine Complexity:**  
  Every routine is designed to have a complexity of no greater than 4, keeping functions short and focused on a single task.
* **Avoiding Code Duplication:**  
  Common validation routines are centralized in the InputValidator class to avoid redundancy.
* **Clear Naming Conventions:**  
  All classes and methods are named descriptively (e.g., ReservationSystem, InputValidator, InvalidInputException) to enhance readability and maintainability.
* **Robust Testing:**  
  A comprehensive JUnit test suite covers all possible input scenarios, ensuring complete code and branch coverage.