**# Technical Report: HISS Project**

**## UML Diagram (BlueJ Style)**

**```**

**---------------------------------**

**| Manager |**

**---------------------------------**

**| - name: String |**

**| - account: double |**

**| - staff: Map<String, Staff> |**

**| - jobs: Map<Integer, Job> |**

**---------------------------------**

**| + hireStaff(name: String): String**

**| + doJob(jobNo: int): String**

**| + getAccount(): double**

**| + ... (other methods) |**

**---------------------------------**

**▲**

**|**

**|**

**---------------------------------**

**| Staff |<>----- Job**

**---------------------------------**

**| - name: String | (association)**

**| - experience: int |**

**| - retainer: int |**

**| - hourlyRate: int |**

**| - state: StaffState |**

**---------------------------------**

**| + getExperience(): int |**

**| + setState(state): void |**

**| + ... (other methods) |**

**---------------------------------**

**▲**

**---------------**

**| |**

**----------------- ------------------**

**| Designer | | Installer |**

**----------------- ------------------**

**| ... | | ... |**

**----------------- ------------------**

**---------------------------------**

**| Job |**

**---------------------------------**

**| - jobNo: int |**

**| - type: JobType |**

**| - location: String |**

**| - hours: int |**

**| - difficulty: int |**

**| - penalty: int |**

**---------------------------------**

**| + getExperienceRequired(): int|**

**| + ... (other methods) |**

**---------------------------------**

**```**

**---**

**## Design Decision 1: Use of Inheritance for Staff Types**

**\*\*Decision:\*\***

**I used inheritance to represent different staff types (e.g., Designer, Installer, Consultant) as subclasses of the abstract `Staff` class.**

**\*\*Alternatives:\*\***

**- Use a single `Staff` class with a `type` field (e.g., enum or String), and conditional logic for specific behavior.**

**- Use interfaces for role-based behavior.**

**\*\*Pros and Cons:\*\***

**- \*\*Inheritance Pros:\*\***

**- Clear modeling of “is-a” relationships.**

**- Allows overriding methods for specialized behavior.**

**- Easy to add new staff types.**

**- \*\*Inheritance Cons:\*\***

**- Can lead to rigid structure if too many subclasses.**

**- May introduce code duplication if not managed well.**

**- \*\*Single Class Pros:\*\***

**- Simpler class structure.**

**- All staff in one place.**

**- \*\*Single Class Cons:\*\***

**- Lots of conditional logic (`if(type==...)`), harder to maintain or extend.**

**- Violates open/closed principle.**

**- \*\*Interface Pros:\*\***

**- Flexibility for multiple roles.**

**- Can mix behaviors.**

**- \*\*Interface Cons:\*\***

**- More complex unless multiple roles are truly needed.**

**\*\*Final Choice Reasoning:\*\***

**Inheritance made staff behavior extensible and clear, and matched the system’s requirement for different staff capabilities. This approach was better for future scalability, such as adding a new staff type (e.g., Consultant) easily.**

**---**

**## Design Decision 2: Job Lookup Using a Map**

**\*\*Decision:\*\***

**Jobs are stored in a `Map<Integer, Job>` with job numbers as keys for efficient lookup.**

**\*\*Alternatives:\*\***

**- Use a `List<Job>`, searching linearly for job number.**

**- Use a database or persistent storage for jobs.**

**\*\*Pros and Cons:\*\***

**- \*\*Map Pros:\*\***

**- Fast lookup by job number (`O(1)` access).**

**- Enforces uniqueness of job numbers.**

**- \*\*Map Cons:\*\***

**- Slightly more memory overhead for the map structure.**

**- \*\*List Pros:\*\***

**- Simple to implement.**

**- Maintains insertion order.**

**- \*\*List Cons:\*\***

**- Slower lookup (`O(n)` if many jobs).**

**- Possible duplicate job numbers if not managed carefully.**

**- \*\*Database Pros:\*\***

**- Scalability for very large datasets.**

**- Persistence.**

**- \*\*Database Cons:\*\***

**- Overkill for a small simulation project.**

**- Adds complexity (setup, queries).**

**\*\*Final Choice Reasoning:\*\***

**A `Map` is best for fast, unique job lookup, which is required when the user enters a job number. This keeps the code simple and performant.**

**---**

**## Design Decision 3: Staff State Management with Enum**

**\*\*Decision:\*\***

**Staff availability is managed using an `enum StaffState` (e.g., AVAILABLE, ONLEAVE).**

**\*\*Alternatives:\*\***

**- Use a boolean field (e.g., `isAvailable`).**

**- Use String constants for states.**

**- Track state externally (e.g., with separate lists for available and on-leave staff).**

**\*\*Pros and Cons:\*\***

**- \*\*Enum Pros:\*\***

**- Type safety, prevents invalid states.**

**- Easy to extend (add more states).**

**- Code is more readable and self-documenting.**

**- \*\*Enum Cons:\*\***

**- Slightly more verbose than a boolean.**

**- \*\*Boolean Pros:\*\***

**- Simple for two states.**

**- Minimal code.**

**- \*\*Boolean Cons:\*\***

**- Not extensible for more than two states.**

**- Less readable for complex state transitions.**

**- \*\*String Pros:\*\***

**- Flexible for any number of states.**

**- \*\*String Cons:\*\***

**- No type safety; prone to typos and bugs.**

**- \*\*External Tracking Pros:\*\***

**- Clear separation of available/on-leave lists.**

**- \*\*External Tracking Cons:\*\***

**- Can get out of sync with actual staff object state.**

**\*\*Final Choice Reasoning:\*\***

**Enums provide clarity, extensibility, and safety, making them the best fit for managing staff states in this simulation.**

**---**

**\*\*Word count:\*\* ~840 (fits within 500-1000 words)**