**Technical Report – HISS REFDEF CWK – Jordan Anyanwu**

**UML DIAGRAM:** A screenshot of a computer

AI-generated content may be incorrect.

**This technical report outlines three critical design decisions made in the development of the HISS (Home Installation Scheduling System) Coursework. For each decision, I describe my chosen approach, consider possible alternatives, evaluate both my design decisions and alternatives trade-offs, and justify my final selection.**

**1. Staff Type Hierarchy: Abstract Class with Subclasses**

**Design Decision**  
**I implemented an abstract Staff class, with specific staff roles (e.g., Designer, Installer, Consultant) as subclasses, that would inherit methods from the Parent class to be implemented on their own. Some other alternative approaches that could have been considered;**

1. **Use a single Staff class with a role field or enum.**
2. **Use interfaces to define role-based behavior.**

**Evaluation:**

* **Abstract Class with Subclasses - Chosen Approach**
  + **Pros:**
    - **Enables a “is-a” relationship between Staff and its subtypes.**
    - **allows for different staff role-specific behaviour and attributes.**
    - **Allows for easy addition of future roles, if necessary.**
  + **Cons:**
    - **There is an Issue of a more complex class hierarchy.**
    - **There is a small amount of run-time overhead, as the code must do additional lookups.**
* **Single Class with Role Field**
  + **Pros:**
    - **Simple structure; all staff represented in one class.**
    - **Easy to add roles via an Enum or string.**
  + **Cons:**
    - **Requires conditional logic throughout the codebase.**
    - **Less maintainable as role-based complexity increases.**
* **Interfaces for Role Behavior**
  + **Pros:**
    - **Flexible design supporting multiple behaviors per class.**
    - **Decouples functionality from the base class.**
  + **Cons:**
    - **Potentially excessive abstraction for current needs.**
    - **Adds overhead for simple role modelling.**

**Justification**  
We used an abstract class with subclasses to follow object-oriented design. This keeps role-specific behaviour organized, avoids messy role checks, and makes it easy to add new roles later.

**2. Job Collection: HashMap Indexed by Job Number**

**Design Decision**

**I decided to have all Job objects stored in a HashMap<Integer, Job>, using each job’s unique job number, e.g. #1001, as the key.**

**Alternative Approaches:**

1. **Use an ArrayList<Job> and perform linear search.**
2. **Use a TreeMap<Integer, Job> to store jobs in sorted order.**

**Evaluation:**

* **HashMap - Chosen Approach**
  + **Pros:**
    - **Hashmap can be considered time effective due to its O(1) time complexity for retrieval, insertion, and deletion.**
    - **HashMap enforces job number uniqueness.**
    - **HashMap memory is Ideal for frequent, direct job lookups.**
  + **Cons:**
    - **HashMap requires more main memory**
    - **HashMap does not maintain any order.**
* **ArrayList**
  + **Pros:**
    - **Ease of use to add, remove and manipulate elements.**
    - **Maintains insertion order.**
  + **Cons:**
    - **O(n) lookup time, leading to inefficient insertion and deletion operations.**
    - **When the list grows, it may allocate more memory than needed**
* **TreeMap**
  + **Pros:**
    - **Keeps jobs sorted by job number.**
    - **TreeMap time complexity of O(log n) allows for high performance for operations like add, remove and containskey().**
  + **Cons:**
    - **Treemap way of Sorting would not be necessary for the current HISS use cases.**
    - **Treemap can be slower than HashMap, as HashMap time complexity is 0(1).**

**Justification**  
**The HashMap is optimal for HISS primary use case: fast job lookup by unique job number. The added benefits of sorted order (TreeMap) or simplicity (ArrayList) do not outweigh the performance advantages of a HashMap.**

**3. Immutable Job Attributes through constructor**

**Design Decision**

**Job attributes are set once through the constructor and accessed via getters only—no setters, making them immutable.**

**Alternatives approaches:**

* **Allow setters to modify attributes**
* **Use a mutable structure**

**Evaluation**

* **Immutability – Chosen approach** 
  + **Pros:**
    - **Ensure data consistency**
    - **Easy for debugging**
  + **Cons:**
    - **Attributes cannot be changed after creation**
    - **Less flexible if job updates are necessary**
* **Mutability** 
  + **Pros:**
    - **Allows for updating job data**
    - **More flexible for future changes**
  + **Cons** 
    - **Difficult to trace bugs from changes in data**
    - **Chances of inconsistent state.**

**Justification**

**In this context Jobs are static task in this context, immutability keeps the logic simple and reliable, and design can evolve later when and if needed.**

**Conclusion**  
Each design choice was made to keep the HISS system codebase easy to understand, grow, and manage over time. By using good programming practices and picking the right ways to store and organize data, I have built a solid base that works in line with the specification provided, while matching the case study of the HISS system and can easily adapt in the future.