**Project Management Pet Care Management**

**Project Name:** Pet Care Management  
**Group:** Paws & Co  
**Sprint:** 4

**Introduction**

The Pet Care Management System is a comprehensive application developed in C# to support the operations of veterinary clinics. The system efficiently manages client (owner), pet, and appointment data by combining a custom hash table for in‑memory storage, EF Core for robust database interactions, and Spectre.Console to deliver a clear text‑based user interface (CLI). This report outlines the design rationale, detailed algorithm analysis with pseudocode, testing methodologies, critical reflection on encountered challenges, and strategies for future improvement.

**Design and Implementation**

**a. System Architecture**

* **User Interface (CLI):**  
  Leveraging Spectre.Console, the CLI supports an interactive menu system offering options to display, add, update, and delete records, search data, and save synchronised data to a database or load from a CSV file. This design ensures clarity and ease-of-use for non-technical end-users, such as clinic receptionists and veterinarians.
* **Data Structures:**  
  A custom hash table is deployed to store Owner, Pet, and Appointment objects in memory. Chosen for its average-case constant time complexity (O(1)) for key operations (insert, search, delete), the structure also incorporates linear probing for collision resolution. This guarantees rapid data access, which is critical in high-paced clinical environments.
* **Data Management:**  
  EF Core manages database operations, ensuring that the in-memory data remains consistent with the SQL Server backend. A dedicated CSV parser also allows for importing external data, addressing situations where legacy data must be integrated with minimal manual intervention.

**b. Algorithm Analysis and Pseudocode**

**i. Hash Table Insertion (Linear Probing)**

procedure Insert(key, value)

index ← Hash(key) mod Capacity

while Buckets[index] is occupied do

if Buckets[index].key = key then

Update Buckets[index] with value

return

index ← (index + 1) mod Capacity

Buckets[index] ← (key, value)

Size ← Size + 1

if Size ≥ LoadFactor × Capacity then

Resize HashTable

* **Analysis:**
  + **Best/Average Case:** O(1)
  + **Worst Case:** O(n) under heavy clustering  
    This algorithm provides direct access for data retrieval, critical in a system that must quickly update and retrieve patient records.

**ii. Unique ID Generation**

procedure GenerateUniqueId(HashTable, DbSet)

repeat

id ← RandomInt(0, 9999)

until (id not in HashTable) and (id not in DbSet)

return id

* **Analysis:**  
  This loop-based approach ensures uniqueness by checking both the in-memory and database records, achieving expected constant time performance for moderate data sizes.

**iii. Database Upsert Operation**

procedure UpsertRecord(record, DbSet)

existing ← DbSet.Find(record.Id)

if existing is null then

DbSet.Add(record)

else

Update existing with record’s data

Commit transaction

* **Analysis:**  
  By leveraging EF Core’s transactional capabilities, the upsert method maintains data integrity and efficiency through batched commits.

**Testing and Quality Assurance.**

**a. Unit Testing**

* **Hash Table Functions:**  
  Multiple unit tests confirmed the integrity of insertion, deletion (including handling of tombstones), and search operations. Special cases, such as collision resolution, were rigorously tested.
* **Unique ID Generation & Input Validation:**  
  Tests ensured that ID generation consistently produces unique values, while validators for names, emails, phone numbers, and addresses correctly distinguish between valid and invalid inputs.

**b. Integration Testing**

* **Database Synchronisation:**  
  By employing an in‑memory SQL Server (via EF Core’s in-memory provider), integration tests validated upsert, deletion, and cascading operations. These tests confirmed that changes in the in-memory hash tables are accurately reflected in the database.
* **CSV Parsing:**  
  End-to-end tests using sample CSV files verified that the parser correctly loaded records into the appropriate hash tables, handling errors in malformed rows gracefully.

**c. Manual Testing**

* **CLI and User Interaction:**  
  Comprehensive manual tests ensured that the menu navigation is intuitive and that error messages and prompts are clear. The testing also confirmed the system’s responsiveness across various user scenarios, including invalid data entries.

**d. Performance Testing**

* **Load Simulation:**  
  The system was subjected to simulated heavy data loads to ensure that the hash table maintains its efficiency and that the unique ID generation mechanism scales adequately.

**Limitations and Critical Reflection**

**a. Limitations Encountered**

* **VSCode Instability:**  
  The predominant issue was frequent crashes in VSCode during Git uploads. This instability disrupted workflow and led to delays, occasional data loss, and a greater need for manual change tracking.
* **Hardware Constraints:**  
  Limited hardware resources compounded the IDE issues, necessitating a less-than-ideal development environment.

**b. Impact on the Project**

* **Reduced Efficiency:**  
  The disruptions compelled the team to adopt a cautious approach with incremental commits and more frequent backups, which, while mitigating data loss, ultimately slowed progress.
* **Increased Manual Intervention:**  
  Manual monitoring became essential for ensuring repository integrity, detracting from development time that could have been spent enhancing features.

**Proposed Improvements for Future Projects**

* **Enhanced Development Environment:**  
  Optimise IDE configurations or consider an alternative development tool better suited to managing large, complex projects and extensive Git operations.
* **Refined Version Control Practices:**  
  Adopt incremental commits and robust branching strategies to avoid large-scale uploads prone to causing crashes. Implement automated backups and regular Git repository checks.
* **Automated Testing and Continuous Integration:**  
  Integrate continuous integration (CI) tools to automate unit and integration tests, thereby isolating environment-specific issues and ensuring consistent code quality.
* **Streamlined Workflow Documentation:**  
  Maintain up-to-date process documentation and a shared project management tool to track progress and promptly address issues, reducing the need for manual intervention.

**Conclusion**

The Pet Care Management System is a robust, feature-rich application that meets the needs of veterinary clinics by providing rapid data access, efficient record management, and a user-friendly interface. Despite setbacks due to IDE instability and limited hardware resources, the project achieved its objectives through rigorous testing, adaptive version control practices, and strategic problem-solving. The critical reflections and suggested future improvements outline a pathway to even more efficient development cycles, ensuring that similar challenges are mitigated in future projects.

**Project Monitoring & Evaluation**

**Gantt Chart (Project Timeline)**

To manage the Pet Care Management System development efficiently, the team followed a structured timeline over 4 weeks, with overlapping tasks for improved productivity.

**A graph with blue rectangles

AI-generated content may be incorrect.**

**Effort Distribution Pie Chart**

**A pie chart with different colored circles

AI-generated content may be incorrect.**