PROJECT REPORT

Object Oriented Programming

Submitted to
Ma'am Sajida Kalsoom

(SP24-BCS-082) NISHA SHAKOOR
(SP24-BCS-092) SYED DANIYAL HAIDER NAQWI

Smart City Resource Management System - Technical Documentation

Table of Contents

System Overview

The Smart City Resource Management System is a comprehensive Java Swing application that manages urban infrastructure including transportation, power generation, and emergency services. The system implements advanced Object-Oriented Programming principles while addressing real-world urban challenges through an integrated management platform.

Key Features

- Real-time resource monitoring and management
- Role-based access control (Admin vs User modes)
- Dynamic resource scheduling and optimization
- Emergency alert system with dependency management
- Interactive map visualization
- Comprehensive reporting system

Smart Grid Algorithms

1. Energy Distribution Algorithm

The system implements a smart grid management approach through the `PowerStation` class with the following algorithms:

Energy Output Calculation

```
// Base maintenance cost calculation for power stations
public double calculateMaintenanceCost() {
    return operationalCost * 0.15;
}
```

Algorithm Logic:

- Maintenance cost is calculated as 15% of operational cost
- This represents the industry standard for power station maintenance

- Lower operational costs result in proportionally lower maintenance needs

Energy Consumption Tracking

```
private static double totalEnergyConsumed = 0;
```

Smart Grid Features:

- **Global Energy Monitoring**: Static variable tracks city-wide energy consumption
- **Real-time Status Updates**: Power stations update their operational status dynamically
- **Efficiency Optimization**: System monitors energy output vs. operational costs

2. Load Balancing Algorithm

Power Station Status Management

```
public void setOperational(boolean operational) {
    this.isOperational = operational;
    this.status = operational ? "ACTIVE" : "DOWN";
}
```

Algorithm Components:

- **Automatic Failover**: When a power station goes down, the system automatically updates status
- **Cascade Alert System**: Power outages trigger emergency alerts to dependent services
- **Resource Optimization**: System can identify underutilized vs. overloaded stations

3. Dependency Management Algorithm

Inter-Resource Communication

```
// Resource Dependency: If status set to DOWN, alert all EmergencyServices
in same location
if (!newStatus.equalsIgnoreCase("ACTIVE")) {
    for (EmergencyService es : emergencyRepo.getAllResources())
        if (es.getLocation().equalsIgnoreCase(p.getLocation()))
            es.sendEmergencyAlert("Power outage at " + p.getLocation() +
"!");
}
```

Smart Grid Intelligence:

- **Location-Based Dependencies**: Services in the same location are automatically notified of power issues
- **Cascading Alert System**: Power failures trigger emergency service alerts
- **Predictive Maintenance**: System can identify at-risk infrastructure

Thread Safety Measures

1. GUI Thread Safety

Event Dispatch Thread (EDT) Compliance

```
SwingUtilities.invokeLater(() -> {
    // All GUI operations performed on EDT
    SmartCityGUI gui = new SmartCityGUI(isAdmin);
    gui.setVisible(true);
});
```

Thread Safety Implementation:

- **EDT Synchronization**: All GUI updates occur on the Event Dispatch Thread
- **Non-blocking Operations**: Long-running tasks don't freeze the GUI
- **Safe Component Updates**: Table refreshes and map redraws are EDT-compliant

2. Background Thread Management

Transport Simulation Thread

```
private void startTransportSimulation() {
    new Thread(() -> {
        Random rand = new Random();
        String[] routes = {"A-B", "B-C", "C-D", "D-E"};
        while (true) {
            for (TransportUnit t : transportRepo.getAllResources()) {
                t.route = routes[rand.nextInt(routes.length)];
                t.currentPassengers = rand.nextInt(t.capacity + 1);
            }
            refreshTable(); // Thread-safe table update
            try { Thread.sleep(5000); } catch (InterruptedException e) {}
        }
    }).start();
}
```

Thread Safety Features:

- **Separate Worker Thread**: Simulation runs on background thread to prevent GUI blocking
- **Safe Data Updates**: Resource modifications are atomic operations
- **Exception Handling**: InterruptedException properly handled for clean shutdown
- **Periodic Updates**: 5-second intervals prevent excessive resource consumption

3. Timer-Based Thread Safety

Emergency Response Timer

```
emergencyTimer = new javax.swing.Timer(dispatched.getResponseTime() *
60_000, e -> {
    dispatched.setAvailable(true);
    refreshTable();
    JOptionPane.showMessageDialog(this, "Emergency service " +
dispatched.getResourceID() + " is now available.");
});
```

Thread Safety Measures:

- **Swing Timer Usage**: Uses Swing Timer instead of regular Timer for EDT compliance
- **Atomic State Changes**: Service availability updates are atomic
- **GUI Synchronization**: Table refreshes occur safely on EDT
- **Resource Cleanup**: Timer properly disposes after single execution

4. Data Consistency Measures

Repository Thread Safety

```
public class CityRepository<T extends CityResource> {
    private List<T> resources = new ArrayList<>();

    public void addResource(T resource) {
        resources.add(resource);
    }

    public void removeResource(T resource) {
        resources.remove(resource);
    }
}
```

Thread Safety Considerations:

- **ArrayList Usage**: While not inherently thread-safe, GUI operations ensure single-threaded access
- **Resource Counting**: Static totalResources counter uses Math.max for safe decrements
- **State Consistency**: Resource state changes are atomic within GUI context

5. Concurrent Access Protection

Static Variable Thread Safety

```
protected static int totalResources = 0;

public void onDelete() {
    totalResources = Math.max(0, totalResources - 1);
}
```

Protection Mechanisms:

- **Math.max Protection**: Prevents negative resource counts in concurrent scenarios
- **Atomic Operations**: Single variable updates are atomic in Java
- **Fail-Safe Design**: System gracefully handles edge cases

OOP Implementation

Inheritance Hierarchy

- **Abstract Base Class**: `CityResource` provides common functionality
- **Concrete Implementations**: `TransportUnit`, `PowerStation`, `EmergencyService`
- **Polymorphic Behavior**: Each class implements `calculateMaintenanceCost()` differently

Interface Implementation

- **Alertable Interface**: Emergency notification capability
- **Reportable Interface**: Usage reporting functionality
- **Multiple Inheritance**: Classes can implement multiple interfaces

Composition and Aggregation

- **Repository Pattern**: `CityRepository<T>` manages collections of resources
- **GUI Composition**: `SmartCityGUI` contains multiple specialized panels
- **Loose Coupling**: Interfaces enable flexible system design

Security Features

Password Protection

```
String adminHash =
"240be518fabd2724ddb6f04eeb1da5967448d7e831c08c8fa822809f74c720a9";
if (hash.equals(adminHash)) {
    isAdmin = true;
}
```

Security Implementation:

- **SHA-256 Hashing**: Passwords are hashed using cryptographic algorithms
- **No Plain Text Storage**: Original passwords are never stored
- **Role-Based Access**: Admin privileges are properly protected

GUI Architecture

Model-View Architecture

- **Data Models**: Repository classes manage data
- **View Components**: Swing components display information
- **Controller Logic**: Event handlers manage user interactions

Real-Time Updates

- **Dynamic Table Updates**: Resource changes immediately reflect in GUI

- **Map Visualization**: Color-coded resource status display
- **Responsive Design**: GUI adapts to different resource counts

Error Handling

Exception Management

```
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {
    // Proper exception handling for thread interruption
}
```

Input Validation

- **Capacity Limits**: Passenger updates are bounded by transport capacity
- **Null Checks**: Dialog inputs are validated before processing
- **Range Validation**: Numeric inputs are constrained to valid ranges

Graceful Degradation

- **Missing Resources**: System handles empty repositories gracefully
- **Invalid Operations**: User errors result in helpful error messages
- **Resource Cleanup**: Timers and threads are properly managed

Performance Optimizations

Efficient Data Structures

- **ArrayList Usage**: Optimal for sequential access patterns
- **HashMap Potential**: Could be implemented for O(1) resource lookups
- **Memory Management**: Objects are properly dereferenced

GUI Optimization

- **Selective Repainting**: Only necessary components are updated
- **Batch Operations**: Multiple changes are batched for efficiency
- **Resource Pooling**: Swing components are reused where possible

Future Enhancements

Proposed Thread Safety Improvements

1. **Concurrent Collections**: Replace ArrayList with ConcurrentHashMap
2. **ReadWriteLocks**: Implement fine-grained locking for repository access
3. **Actor Model**: Consider actor-based concurrency for resource management

Smart Grid Algorithm Enhancements

1. **Machine Learning**: Predictive maintenance using historical data
2. **Optimization Algorithms**: Genetic algorithms for resource allocation
3. **Real-Time Analytics**: Stream processing for live data analysis

Conclusion

The Smart City Resource Management System demonstrates advanced software engineering principles while maintaining thread safety and implementing intelligent algorithms. The system's modular design allows for future enhancements while providing a solid foundation for urban infrastructure management.

The combination of OOP principles, thread-safe operations, and smart grid algorithms creates a robust platform capable of handling real-world urban management challenges. The documentation serves as a comprehensive guide for understanding, maintaining, and extending the system's capabilities.