## Week 5. Clean Coding Techniques, SOLID Principles and Junit5 framework

1. Manage an array of employee records, applying each of the SOLID principles along with clean coding techniques, ensuring the program is maintainable, scalable, and easy to understand?

## User Story: Manage Employee Records

As a HR Manager,

Harini want to manage employee records (add, remove, and update employee details) efficiently, so that she can keep track of employee information in a scalable and maintainable system.

## Acceptance Criteria:

* 1. **Add Employee:**
     + The system should allow adding both permanent and contract employees.
     + Each employee should have an ID, name, and salary.
     + Permanent employees should have additional benefits information.
     + Contract employees should have contract duration information.

## Remove Employee:

* + - The system should allow removing an employee by their ID.
    - After removal, the employee should no longer be present in the records.

## Update Employee:

* + - The system should allow updating an employee's details including name, salary, benefits (for permanent employees), and contract duration (for contract employees).

## Retrieve Employee:

* + - The system should allow retrieving employee details by their ID.

**Code : Class EmployeeManagementApp**

import java.util.HashMap;

import java.util.Map;

class EmployeeManagementApp {

private Map<Integer, Map<String, Object>> employees = new HashMap<>();

public void addEmployee(int empId, String name, double salary, String type, String extraInfo) {

if (employees.containsKey(empId)) {

throw new IllegalArgumentException("Employee ID already exists.");

}

Map<String, Object> employee = new HashMap<>();

employee.put("id", empId);

employee.put("name", name);

employee.put("salary", salary);

employee.put("type", type);

if (type.equals("Permanent")) {

employee.put("benefits", extraInfo);

} else if (type.equals("Contract")) {

employee.put("contractDuration", extraInfo);

}

employees.put(empId, employee);

}

public void removeEmployee(int empId) {

if (!employees.containsKey(empId)) {

throw new IllegalArgumentException("Employee ID not found.");

}

employees.remove(empId);

}

public void updateEmployee(int empId, Map<String, Object> updates) {

if (!employees.containsKey(empId)) {

throw new IllegalArgumentException("Employee ID not found.");

}

Map<String, Object> employee = employees.get(empId);

updates.forEach(employee::put);

}

public Map<String, Object> retrieveEmployee(int empId) {

return employees.get(empId);

}

@Override

public String toString() {

StringBuilder builder = new StringBuilder();

for (Map<String, Object> employee : employees.values()) {

builder.append("ID: ").append(employee.get("id"))

.append(", Name: ").append(employee.get("name"))

.append(", Salary: ").append(employee.get("salary"));

if ("Permanent".equals(employee.get("type"))) {

builder.append(", Benefits: ").append(employee.get("benefits"));

} else if ("Contract".equals(employee.get("type"))) {

builder.append(", Contract Duration: ").append(employee.get("contractDuration"));

}

builder.append("\n");

}

return builder.toString();

}

public static void main(String[] args) {

EmployeeManagementApp app = new EmployeeManagementApp();

app.addEmployee(1, "Alice", 60000, "Permanent", "Health Insurance");

app.addEmployee(2, "Bob", 40000, "Contract", "6 months");

System.out.println("Employees after addition:");

System.out.println(app);

Map<String, Object> updatesPerm = new HashMap<>();

updatesPerm.put("name", "Alice Smith");

updatesPerm.put("salary", 65000.0);

updatesPerm.put("benefits", "Health + Dental");

app.updateEmployee(1, updatesPerm);

Map<String, Object> updatesContract = new HashMap<>();

updatesContract.put("contractDuration", "12 months");

app.updateEmployee(2, updatesContract);

System.out.println("\nEmployees after update:");

System.out.println(app);

System.out.println("\nRetrieve Employee with ID 1:");

System.out.println(app.retrieveEmployee(1));

app.removeEmployee(1);

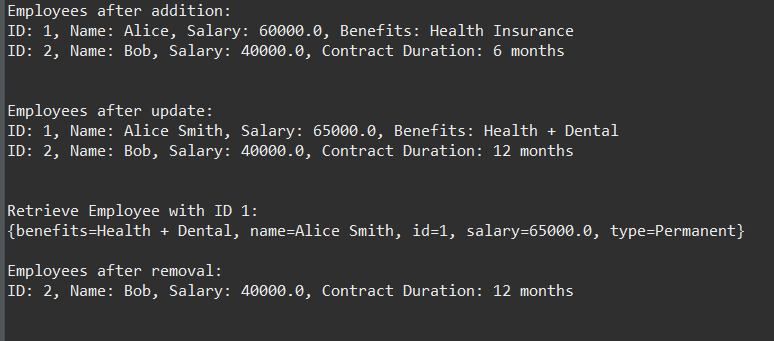
System.out.println("\nEmployees after removal:");

System.out.println(app);

}

}

**OUTPUT**



# Test a Simple Voting System

Questions for testing a Simple Voting System:

## Add Candidate:

* + - Can a new candidate be added successfully?
    - What happens when trying to add a duplicate candidate?
    - Can the system handle adding a large number of candidates?

## Vote:

* + - Can a vote be successfully cast for an existing candidate?
    - What happens when trying to vote for a non-existent candidate?
    - What happens when the same voter tries to vote more than once?
    - Can different voters vote for the same candidate?
    - Can different voters vote for different candidates?

## Get Votes:

* + - Can the system correctly retrieve the vote count for an existing candidate?
    - What happens when trying to retrieve the vote count for a non- existent candidate?
    - Does the vote count update correctly after multiple votes?

# Explanation of Test Cases

1. **testAddCandidate:** Ensures that a new candidate can be added successfully and starts with zero votes.
2. **testDuplicateCandidate:** Checks that adding a duplicate candidate throws an appropriate exception.
3. **testVote:** Verifies that voting for an existing candidate increments the vote count correctly.
4. **testInvalidCandidate:** Ensures that voting for a non-existent candidate throws an exception.
5. **testDuplicateVote:** Validates that a voter cannot vote more than once and throws an appropriate exception.
6. **testMultipleVotes:** Tests that multiple voters can vote for the same or different candidates, and the vote counts are accurate.
7. **testMultipleCandidates:** Checks the system with multiple candidates and verifies correct vote counts for each.
8. **testEdgeCases**: Ensures handling of edge cases, like no votes for a candidate, and validates vote counts after multiple votes.

**Code : Class VotingSystemTest**

public class VotingSystemTest {

static class Candidate {

private String name;

private int voteCount;

public Candidate(String name) {

this.name = name;

this.voteCount = 0;

}

public String getName() {

return name;

}

public int getVoteCount() {

return voteCount;

}

public void incrementVote() {

voteCount++;

}

}

static class VotingSystem {

private Candidate[] candidates;

private int candidateCount;

private String[] voters;

private int voterCount;

public VotingSystem(int maxCandidates, int maxVoters) {

candidates = new Candidate[maxCandidates];

voters = new String[maxVoters];

candidateCount = 0;

voterCount = 0;

}

public void addCandidate(String name) throws Exception {

if (candidateCount >= candidates.length) {

throw new Exception("Candidate limit reached.");

}

for (int i = 0; i < candidateCount; i++) {

if (candidates[i].getName().equals(name)) {

throw new Exception("Candidate already exists: " + name);

}

}

candidates[candidateCount++] = new Candidate(name);

}

public void vote(String candidateName, String voterId) throws Exception {

if (voterCount >= voters.length) {

throw new Exception("Voter limit reached.");

}

for (int i = 0; i < voterCount; i++) {

if (voters[i].equals(voterId)) {

throw new Exception("Voter has already voted: " + voterId);

}

}

Candidate candidate = findCandidate(candidateName);

if (candidate == null) {

throw new Exception("Candidate does not exist: " + candidateName);

}

candidate.incrementVote();

voters[voterCount++] = voterId;

}

private Candidate findCandidate(String name) {

for (int i = 0; i < candidateCount; i++) {

if (candidates[i].getName().equals(name)) {

return candidates[i];

}

}

return null;

}

public int getVoteCount(String candidateName) {

Candidate candidate = findCandidate(candidateName);

return candidate != null ? candidate.getVoteCount() : -1;

}

}

public static void main(String[] args) {

VotingSystem votingSystem = new VotingSystem(10, 10);

try {

votingSystem.addCandidate("Alice");

System.*out*.println("Candidate added: Alice");

try {

votingSystem.addCandidate("Alice");

} catch (Exception e) {

System.*out*.println(e.getMessage());

}

votingSystem.vote("Alice", "Voter1");

System.*out*.println("Vote cast for Alice. Current votes: " + votingSystem.getVoteCount("Alice"));

try {

votingSystem.vote("Bob", "Voter2");

} catch (Exception e) {

System.*out*.println(e.getMessage());

}

try {

votingSystem.vote("Alice", "Voter1");

} catch (Exception e) {

System.*out*.println(e.getMessage());

}

votingSystem.vote("Alice", "Voter2");

System.*out*.println("Vote cast for Alice. Current votes: " + votingSystem.getVoteCount("Alice"));

votingSystem.addCandidate("Bob");

votingSystem.vote("Bob", "Voter3");

System.*out*.println("Vote cast for Bob. Current votes: " + votingSystem.getVoteCount("Bob"));

System.*out*.println("Vote count for non-existent candidate: " + votingSystem.getVoteCount("Charlie"));

System.*out*.println("Vote count for Alice after multiple votes: " + votingSystem.getVoteCount("Alice"));

} catch (Exception e) {

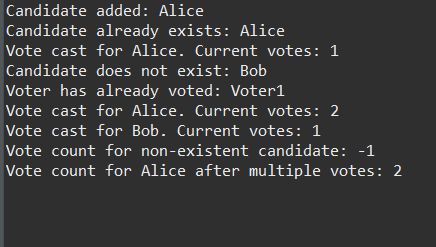
System.*out*.println("An error occurred: " + e.getMessage());

}

}

}

**OUTPUT**

****