Healthcare Data Warehouse Implementation Project

Course: Data Warehousing

Student Name: Syed Abdullah Ali

Student ID: 23SW082

Teacher: Dr. Naeem Ahmed

Department of Software Engineering, Mehran UET

Submission Date: April 15, 2025

Index

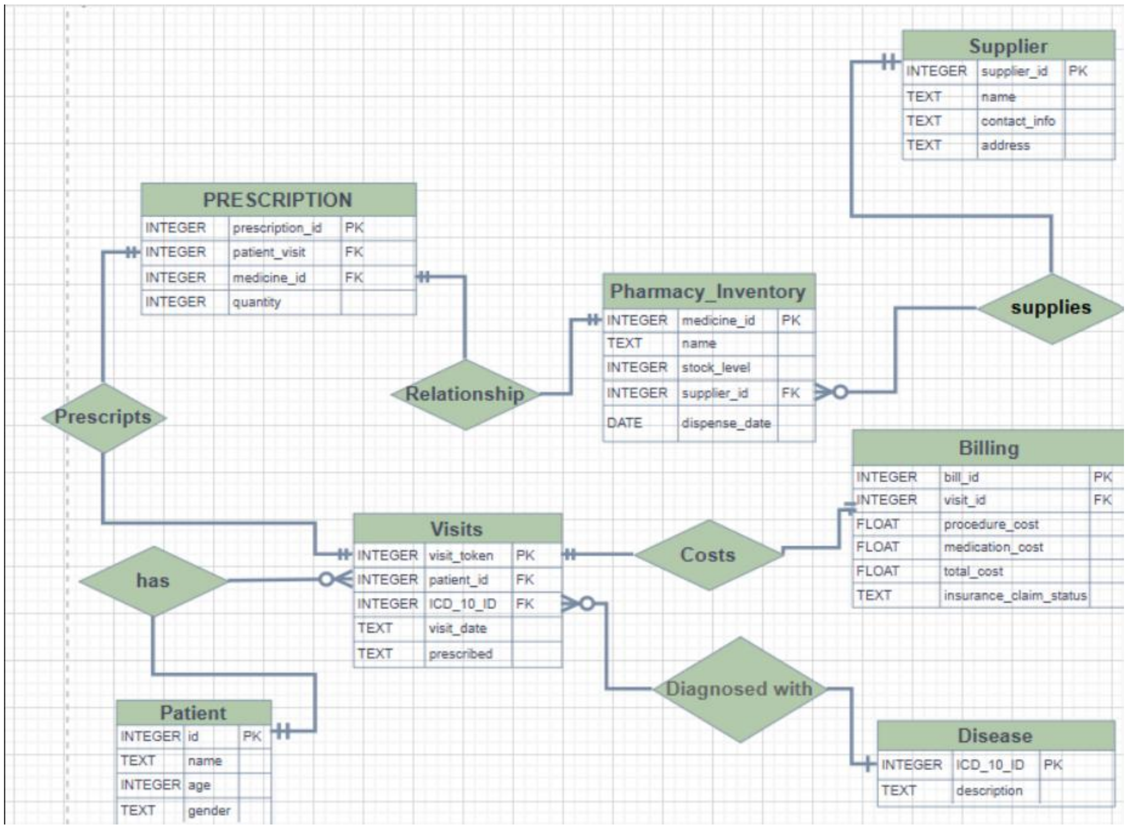
Introduction:

This project implements a healthcare data warehouse by integrating data from a hospital database and public health outbreak reports. It simulates real-world data using random, missing, and dirty data generation techniques. A complete ETL pipeline extracts, transforms, and loads the data into a star schema-based warehouse optimized for analysis. The system supports analytical queries on treatment costs, medication stock trends, and disease outbreak correlations. Optimization techniques enhance storage, query performance, and data quality throughout the process.

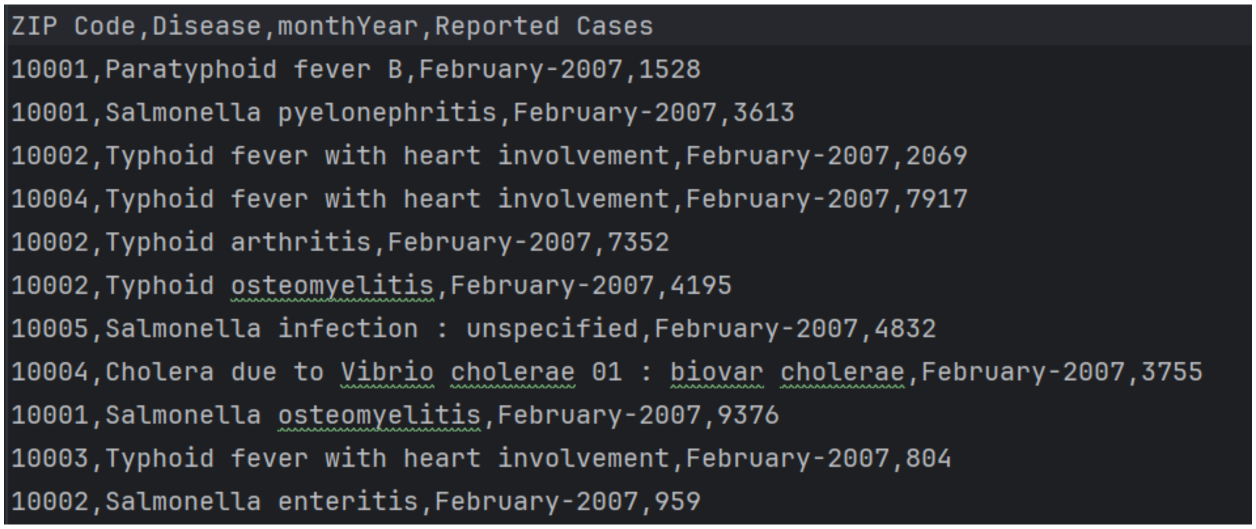
Data Sources:

Data sources in the project were created to store transactional data in a relational data base of hospital and a CSV file of public health (Disease Outbreak) which is generated very month it simulates reports published by health organization about outbreaks

Hospital DB:



Example csv:



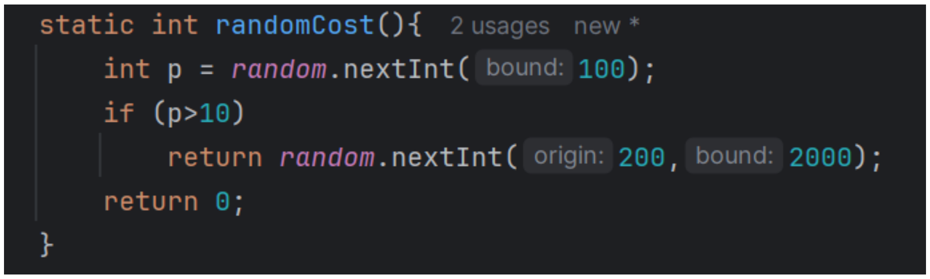
```
ZIP Code,Disease,monthYear,Reported Cases
10001,Paratyphoid fever B,February-2007,1528
10001,Salmonella pyelonephritis,February-2007,3613
10002,Typhoid fever with heart involvement,February-2007,2069
10004,Typhoid fever with heart involvement,February-2007,7917
10002,Typhoid arthritis,February-2007,7352
10002,Typhoid osteomyelitis,February-2007,4195
10005,Salmonella infection : unspecified,February-2007,4832
10004,Cholera due to Vibrio cholerae 01 : biovar cholerae,February-2007,3755
10001,Salmonella osteomyelitis,February-2007,9376
10003,Typhoid fever with heart involvement,February-2007,804
10002,Salmonella enteritis,February-2007,959
```

Random, Missing and Dirty Data Generation:

In class RandomData in package GenerateData method are built to generate random, missing and corrupted data.

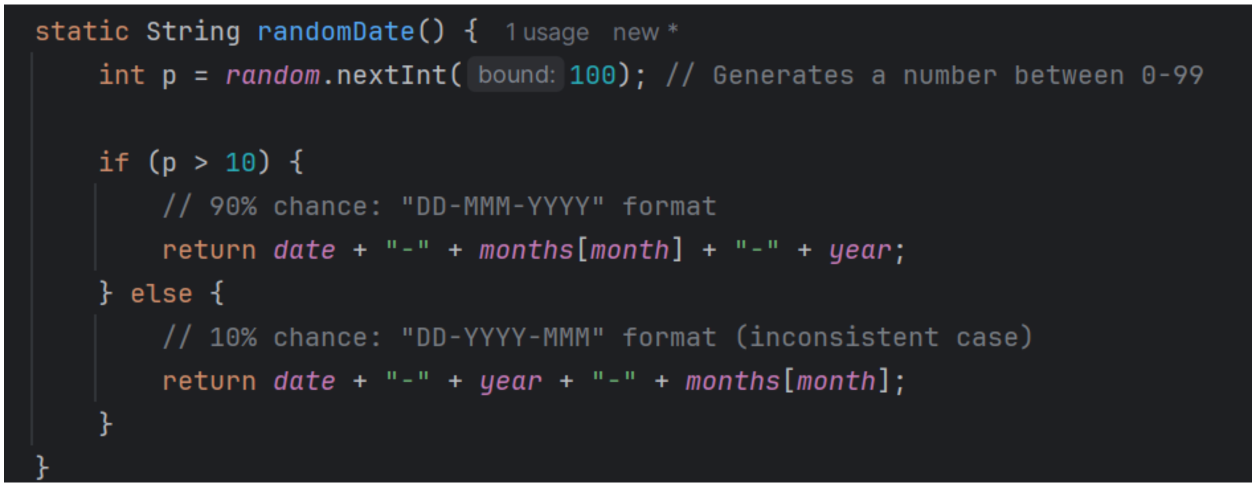
Examples:

Random cost method is an example of corrupted data being generated which 90% of time generates clean random data but 10% of time throws 0 as corrupted value.



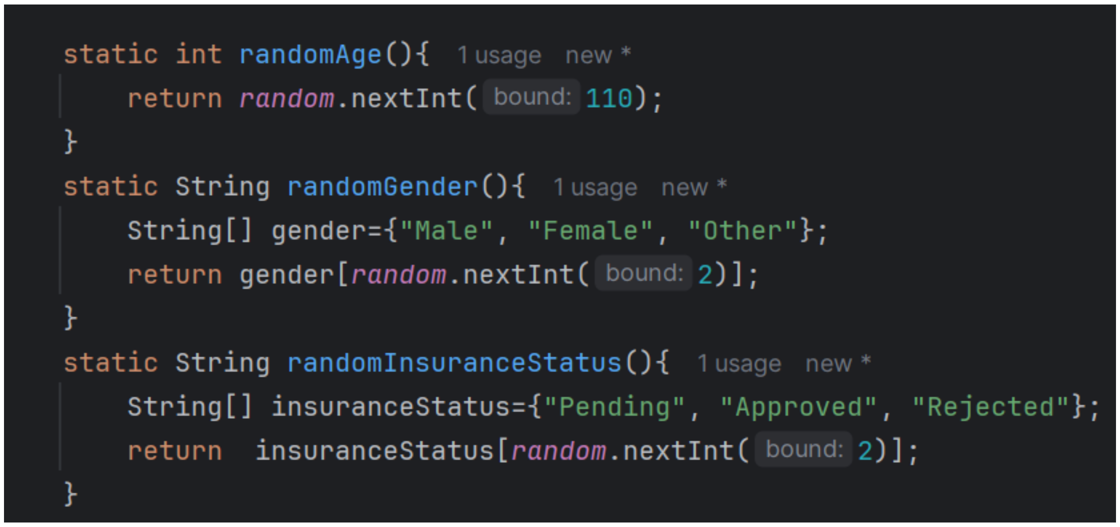
```
static int randomCost(){  2 usages  new *
    int p = random.nextInt( bound: 100);
    if (p>10)
        return random.nextInt( origin: 200, bound: 2000);
    return 0;
}
```

In method randomDate, date is in 10% of time is created in wrong formate.

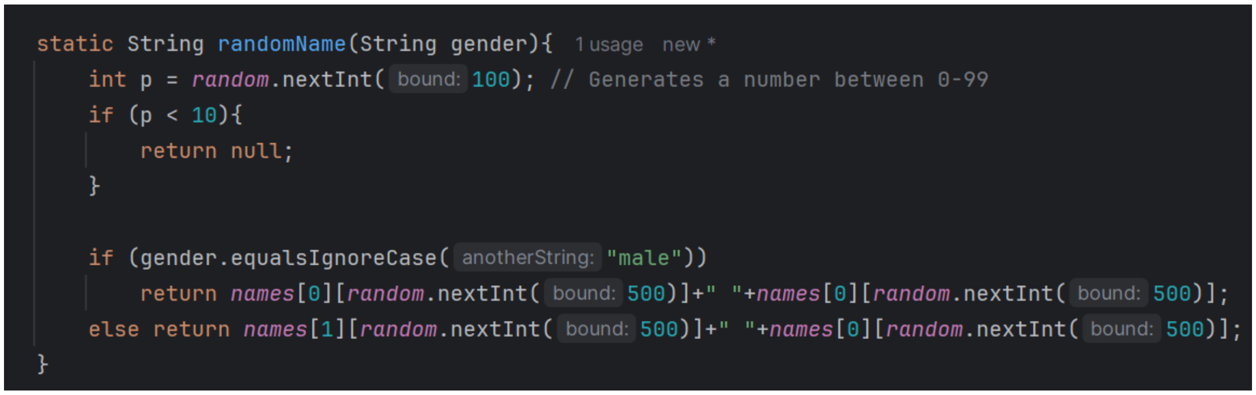

```
static String randomDate() {  1 usage  new *
    int p = random.nextInt( bound: 100); // Generates a number between 0-99

    if (p > 10) {
        // 90% chance: "DD-MMM-YYYY" format
        return date + "-" + months[month] + "-" + year;
    } else {
        // 10% chance: "DD-YYYY-MMM" format (inconsistent case)
        return date + "-" + year + "-" + months[month];
    }
}
```

Methods like random age, random gender and random insurance status generates clean data 100% of times.


```
static int randomAge(){  1 usage  new *
    return random.nextInt( bound: 110);
}
static String randomGender(){  1 usage  new *
    String[] gender={"Male", "Female", "Other"};
    return gender[random.nextInt( bound: 2)];
}
static String randomInsuranceStatus(){  1 usage  new *
    String[] insuranceStatus={"Pending", "Approved", "Rejected"};
    return  insuranceStatus[random.nextInt( bound: 2)];
}
```

In method randomName missing values are generated 10% of time

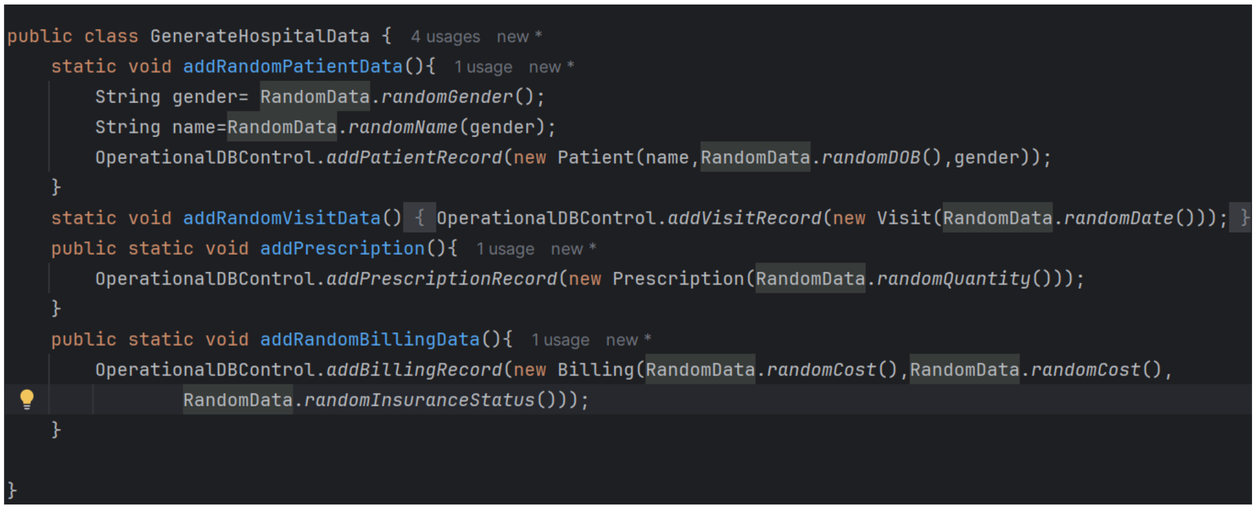

```
static String randomName(String gender){  1 usage  new *
    int p = random.nextInt( bound: 100); // Generates a number between 0-99
    if (p < 10){
        return null;
    }

    if (gender.equalsIgnoreCase( anotherString: "male"))
        return names[0][random.nextInt( bound: 500)]+" "+names[0][random.nextInt( bound: 500)];
    else return names[1][random.nextInt( bound: 500)]+" "+names[0][random.nextInt( bound: 500)];
}
```

Inserting in source data:

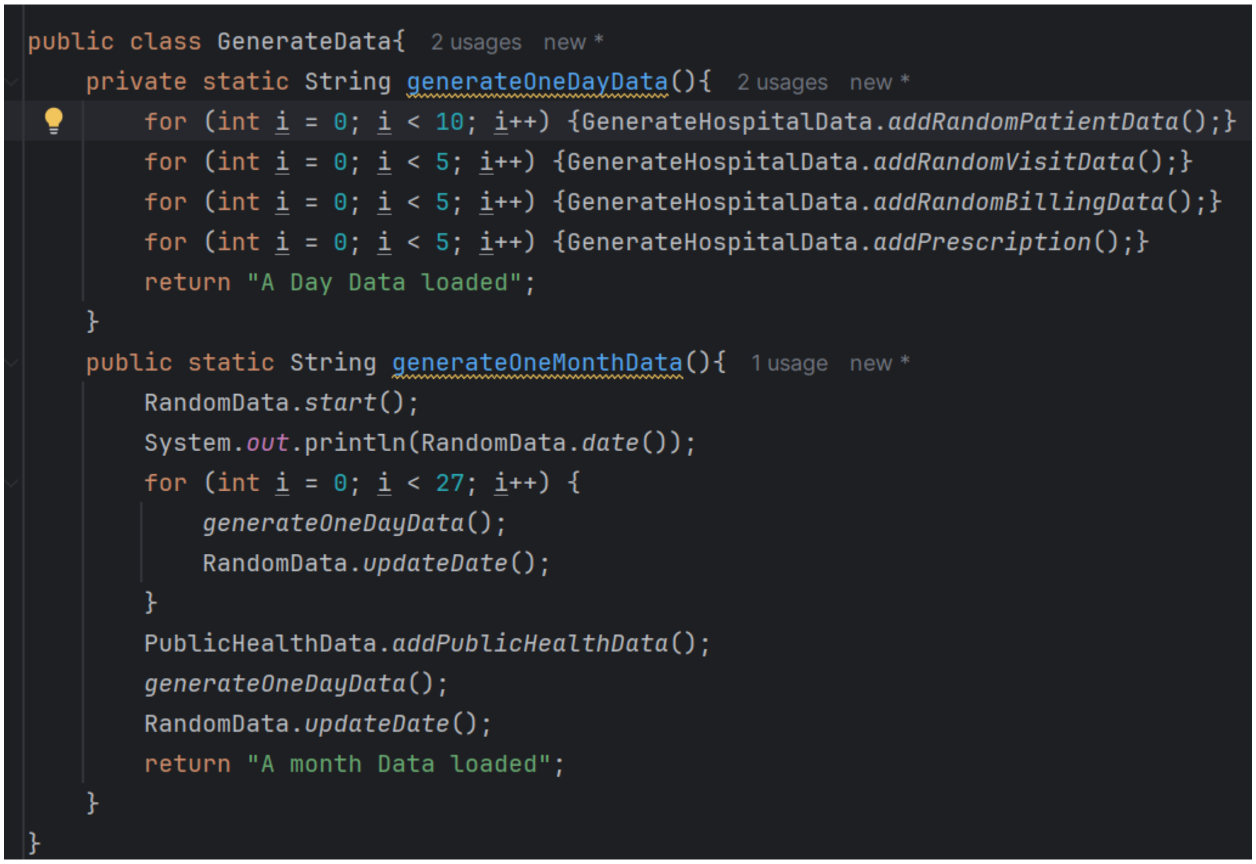
Data is inserted using operational database control class which contains methods for inserting data to operational data and public health CSV file is generated by extractPublic_health_data class both classes are in data base controls package.

Data is inserted in hospital database through generate hospital data class in data generation package


```
public class GenerateHospitalData {  4 usages  new *
    static void addRandomPatientData(){  1 usage  new *
        String gender= RandomData.randomGender();
        String name=RandomData.randomName(gender);
        OperationalDBControl.addPatientRecord(new Patient(name,RandomData.randomDOB(),gender));
    }
    static void addRandomVisitData() { OperationalDBControl.addVisitRecord(new Visit(RandomData.randomDate())); }
    public static void addPrescription(){  1 usage  new *
        OperationalDBControl.addPrescriptionRecord(new Prescription(RandomData.randomQuantity()));
    }
    public static void addRandomBillingData(){  1 usage  new *
        OperationalDBControl.addBillingRecord(new Billing(RandomData.randomCost(),RandomData.randomCost(),
                RandomData.randomInsuranceStatus()));
    }

}
```

Generate data class in data generators package helps to control the amount of data being generated in the hospital data and create new public health data csv after a month data entry.



```java
public class GenerateData{  2 usages   new *
    private static String generateOneDayData(){  2 usages   new *
        for (int i = 0; i < 10; i++) {GenerateHospitalData.addRandomPatientData();}
        for (int i = 0; i < 5; i++) {GenerateHospitalData.addRandomVisitData();}
        for (int i = 0; i < 5; i++) {GenerateHospitalData.addRandomBillingData();}
        for (int i = 0; i < 5; i++) {GenerateHospitalData.addPrescription();}
        return "A Day Data loaded";
    }
    public static String generateOneMonthData(){  1 usage   new *
        RandomData.start();
        System.out.println(RandomData.date());
        for (int i = 0; i < 27; i++) {
            generateOneDayData();
            RandomData.updateDate();
        }
        PublicHealthData.addPublicHealthData();
        generateOneDayData();
        RandomData.updateDate();
        return "A month Data loaded";
    }
}
```

ETL Process Implementation Details

The ETL (Extract, Transform, Load) process is the core of this data warehouse implementation, handling data from multiple healthcare systems and preparing it for analytical queries. The implementation follows a structured approach with dedicated classes for each phase.

RunETL Orchestration

The RunETL class serves as the orchestrator for the complete ETL workflow:

```java
public class RunETL {
    public static String runETL(){
        Load.loadPatientDim(Transform.transformPatientData());
        Load.loadVisitDim(Transform.transformVisitDim());
        Load.loadSupplier(Transform.transformSupplier());
        Load.loadMedicineDim();
        Load.loadFactMedication(Transform.transformFactMedicine());
        Load.loadNumberOfVisit(Transform.transformNumberOfVisit());
        Load.loadOutBreaks(ExtractPublic_Health_Data.ExtractOutbreak());
        return "ETL COMPLETED";
    }
}
```

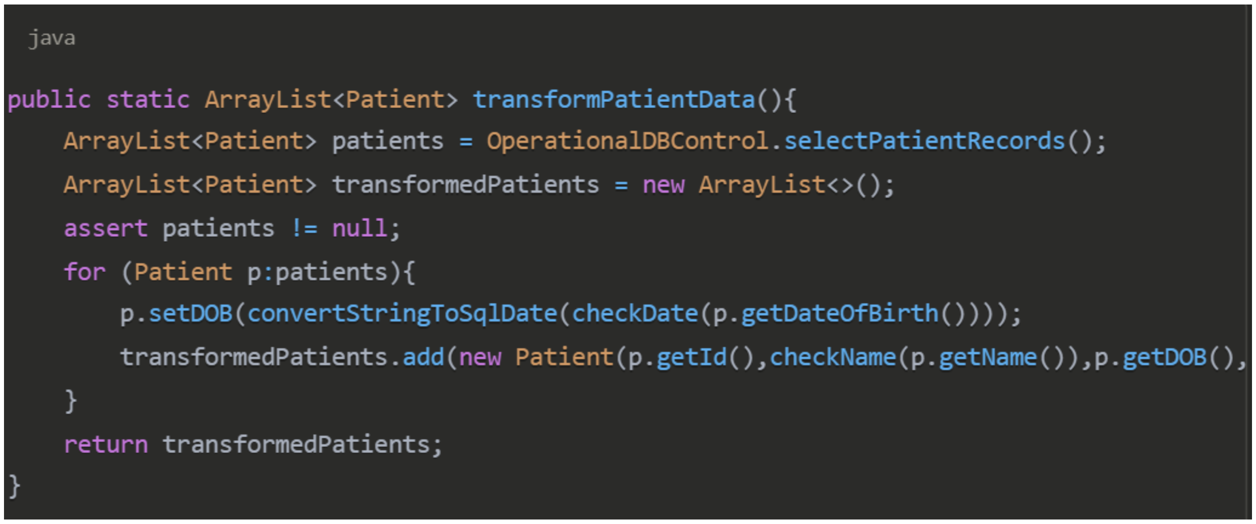
This orchestrator coordinates all ETL steps in a specific sequence, ensuring proper dependency management between dimension and fact tables.

Data Transformation

The Transform class implements data quality handling and business logic to prepare raw data for the warehouse:

1. **Patient Data Transformation**:

   This method standardizes patient names and dates of birth, handling missing values and inconsistent formats.

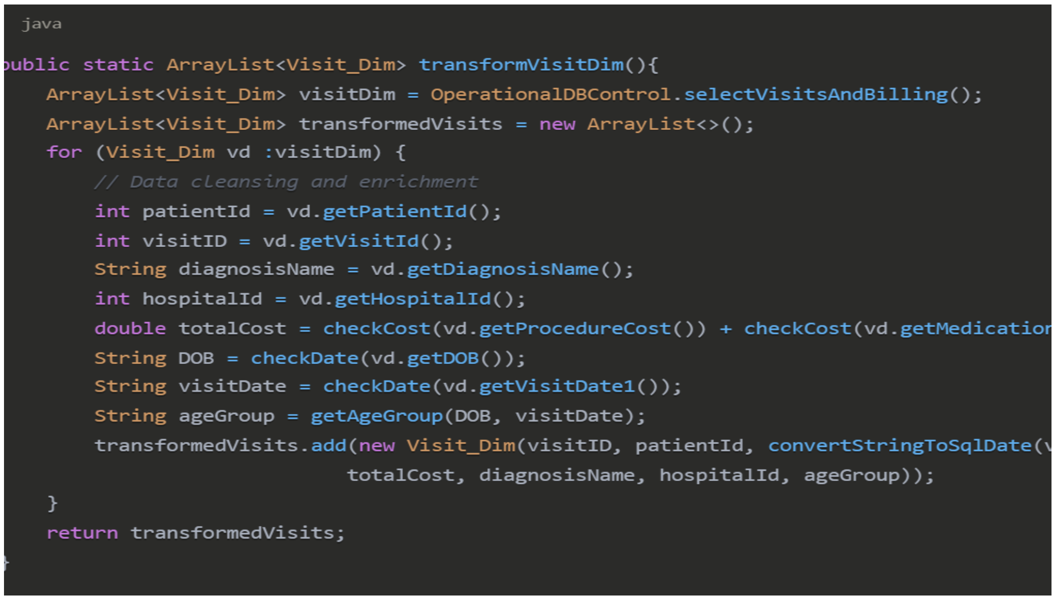


```java
public static ArrayList<Patient> transformPatientData(){
    ArrayList<Patient> patients = OperationalDBControl.selectPatientRecords();
    ArrayList<Patient> transformedPatients = new ArrayList<>();
    assert patients != null;
    for (Patient p:patients){
        p.setDOB(convertStringToSqlDate(checkDate(p.getDateOfBirth())));
        transformedPatients.add(new Patient(p.getId(),checkName(p.getName()),p.getDOB(),
    }
    return transformedPatients;
}
```

2. Visit Dimension Transformation:

This method:

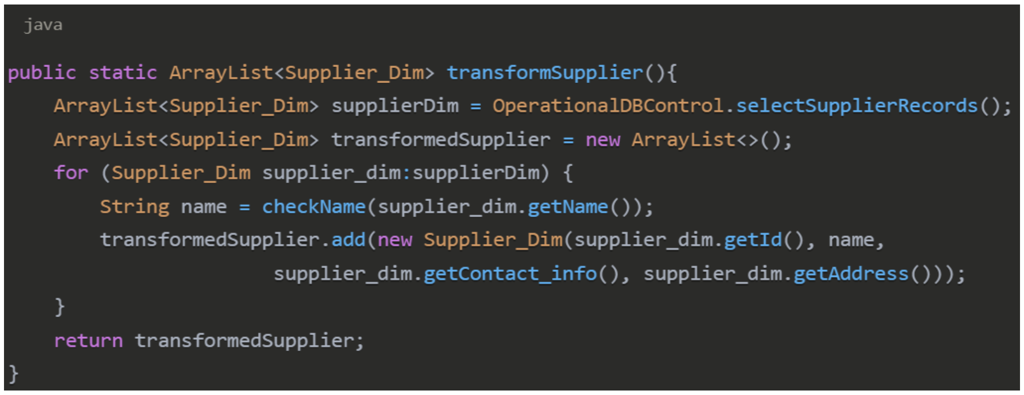
- Standardizes dates
- Calculates total costs
- Derives age groups based on patient birth date and visit date
- Creates a consistent dimensional model for visit analysis



```java
public static ArrayList<Visit_Dim> transformVisitDim(){
    ArrayList<Visit_Dim> visitDim = OperationalDBControl.selectVisitsAndBilling();
    ArrayList<Visit_Dim> transformedVisits = new ArrayList<>();
    for (Visit_Dim vd :visitDim) {
        // Data cleansing and enrichment
        int patientId = vd.getPatientId();
        int visitID = vd.getVisitId();
        String diagnosisName = vd.getDiagnosisName();
        int hospitalId = vd.getHospitalId();
        double totalCost = checkCost(vd.getProcedureCost()) + checkCost(vd.getMedicatio
        String DOB = checkDate(vd.getDOB());
        String visitDate = checkDate(vd.getVisitDate1());
        String ageGroup = getAgeGroup(DOB, visitDate);
        transformedVisits.add(new Visit_Dim(visitID, patientId, convertStringToSqlDate(
                             totalCost, diagnosisName, hospitalId, ageGroup));
    }
    return transformedVisits;
}
```

3. **Supplier Transformation**:

Handles potential null supplier names.

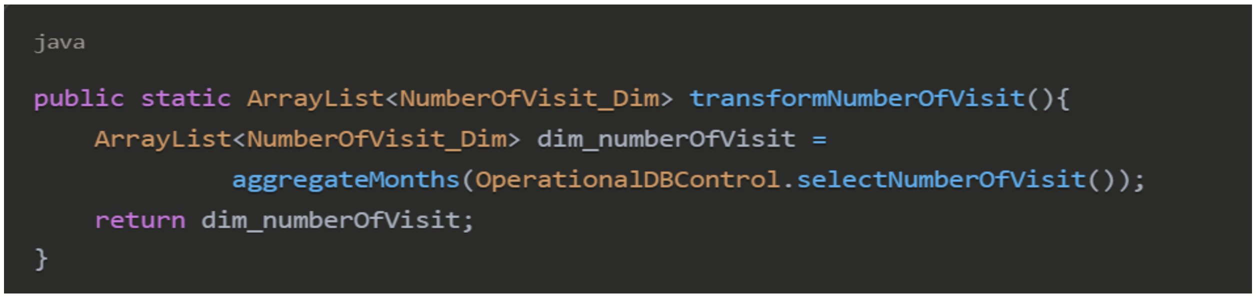


```
java

public static ArrayList<Supplier_Dim> transformSupplier(){
    ArrayList<Supplier_Dim> supplierDim = OperationalDBControl.selectSupplierRecords();
    ArrayList<Supplier_Dim> transformedSupplier = new ArrayList<>();
    for (Supplier_Dim supplier_dim:supplierDim) {
        String name = checkName(supplier_dim.getName());
        transformedSupplier.add(new Supplier_Dim(supplier_dim.getId(), name,
                       supplier_dim.getContact_info(), supplier_dim.getAddress()));
    }
    return transformedSupplier;
}
```

4. **Visit Aggregation Transformation**:

Aggregates visit data by month to support trend analysis.

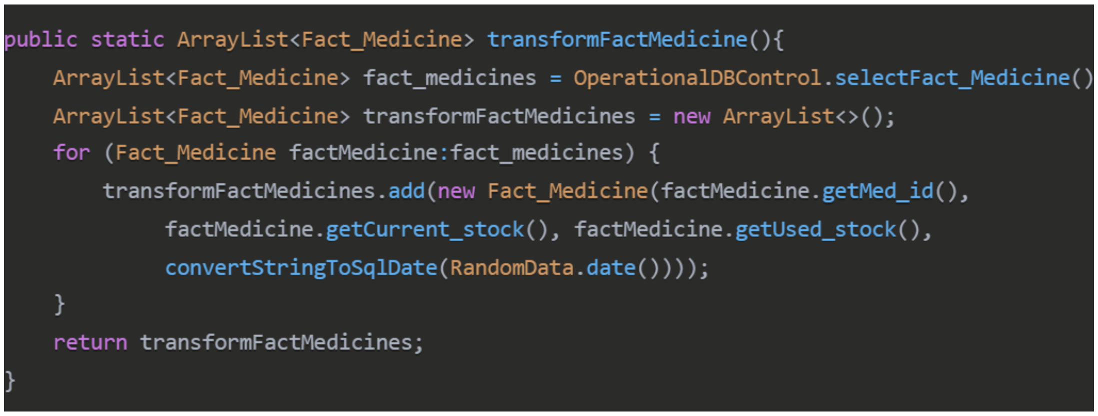


```
java

public static ArrayList<NumberOfVisit_Dim> transformNumberOfVisit(){
    ArrayList<NumberOfVisit_Dim> dim_numberOfVisit =
             aggregateMonths(OperationalDBControl.selectNumberOfVisit());
    return dim_numberOfVisit;
}
```

5. **Medication Fact Transformation:**

Standardizes medication inventory data.



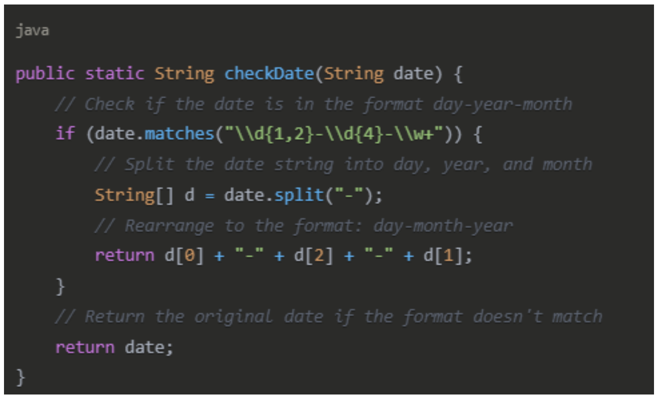
```
public static ArrayList<Fact_Medicine> transformFactMedicine(){
    ArrayList<Fact_Medicine> fact_medicines = OperationalDBControl.selectFact_Medicine()
    ArrayList<Fact_Medicine> transformFactMedicines = new ArrayList<>();
    for (Fact_Medicine factMedicine:fact_medicines) {
        transformFactMedicines.add(new Fact_Medicine(factMedicine.getMed_id(),
            factMedicine.getCurrent_stock(), factMedicine.getUsed_stock(),
            convertStringToSqlDate(RandomData.date())));
    }
    return transformFactMedicines;
}
```

Data Quality Handlers

The FiltersHelperMethod class implements core data quality functions:

1. Date Standardization:

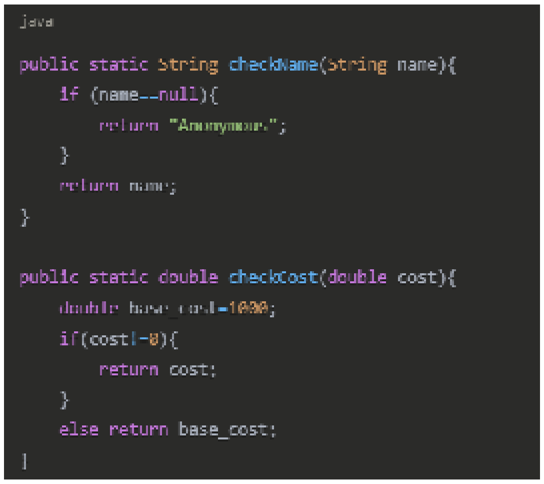
This method detects and standardizes inconsistent date formats, addressing the issue identified in the Hospital Billing System.



```java
public static String checkDate(String date) {
    // Check if the date is in the format day-year-month
    if (date.matches("\\d{1,2}-\\d{4}-\\w+")) {
        // Split the date string into day, year, and month
        String[] d = date.split("-");
        // Rearrange to the format: day-month-year
        return d[0] + "-" + d[2] + "-" + d[1];
    }
    // Return the original date if the format doesn't match
    return date;
}
```

2. Missing Value Handling

These methods handle missing names and costs, providing default values based on business rules.

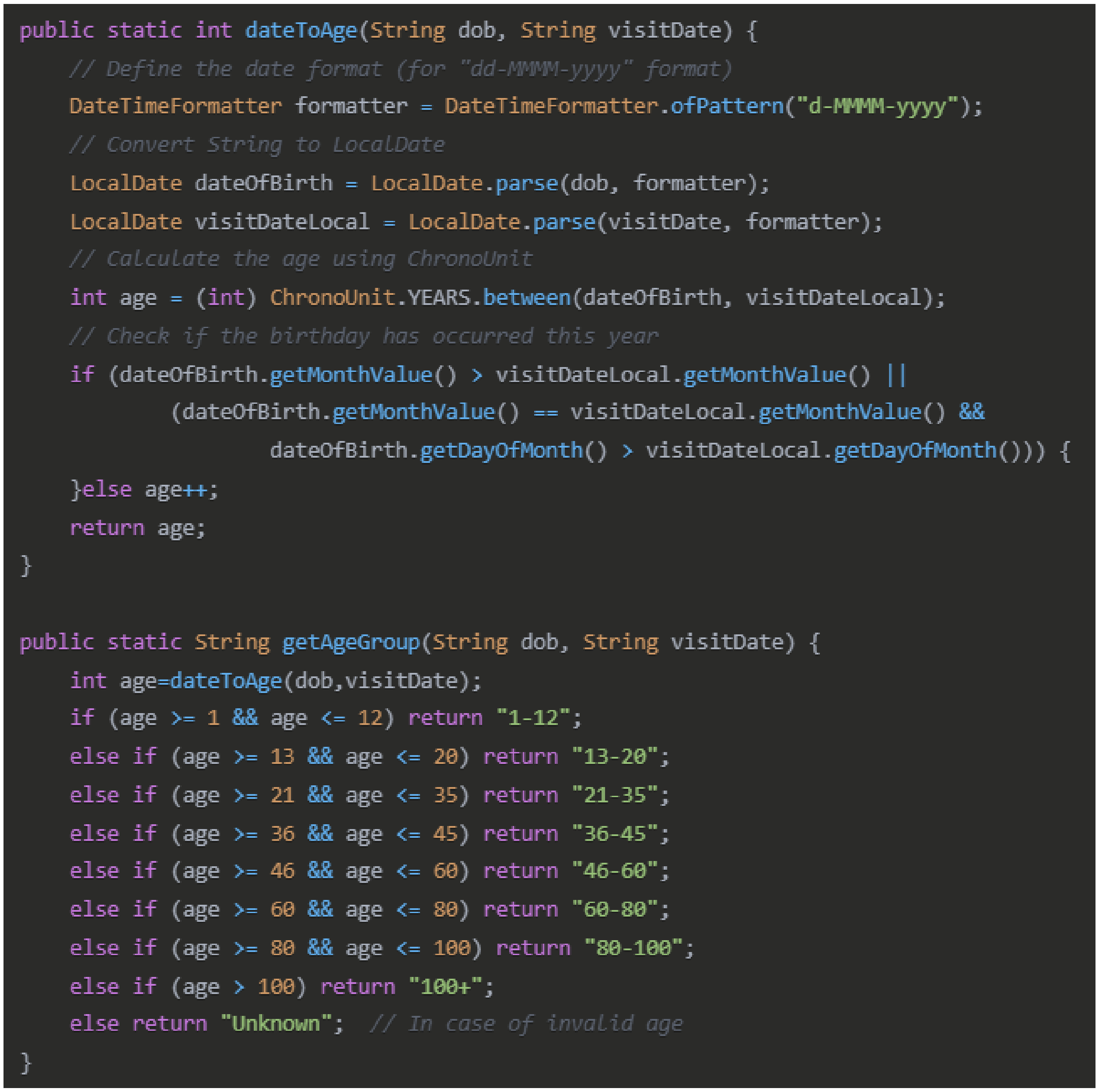


```java
public static String checkName(String name){
    if (name==null){
        return "Anonymous";
    }
    return name;
}

public static double checkCost(double cost){
    double base_cost=1000;
    if(cost!=0){
        return cost;
    }
    else return base_cost;
}
```

3. Age Calculation and Grouping:

These methods accurately calculate age from dates and categorize into standardized age groups for demographic analysis.

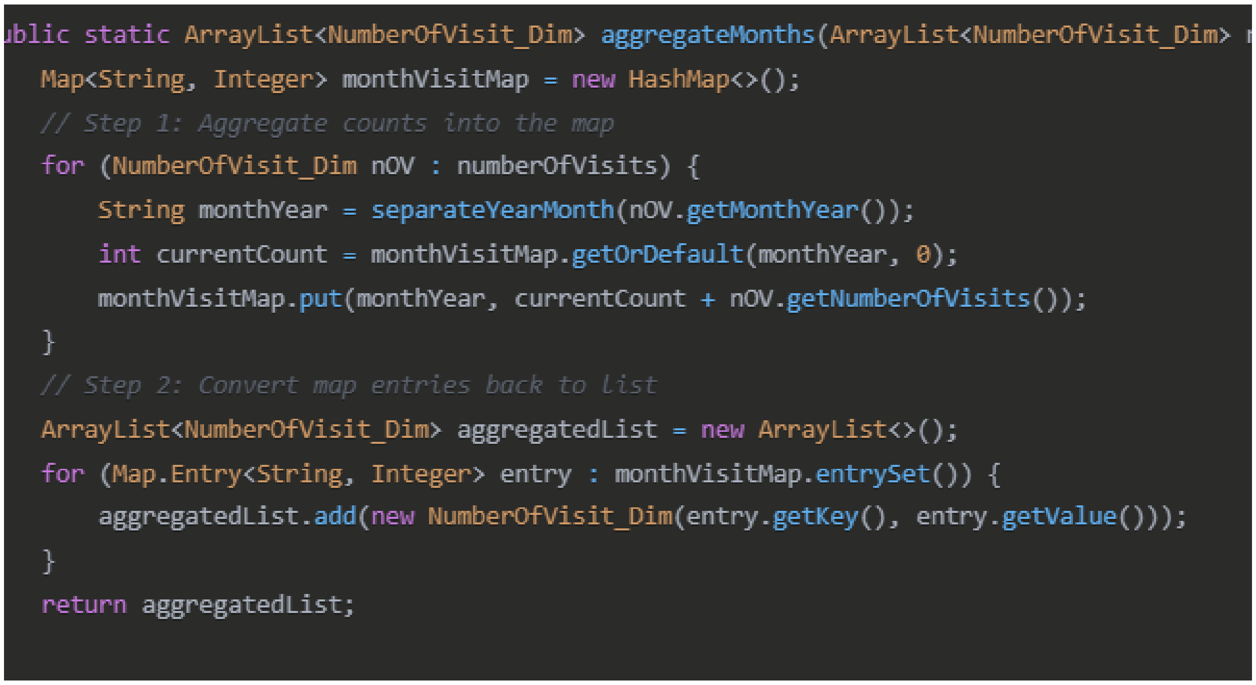


```java
public static int dateToAge(String dob, String visitDate) {
    // Define the date format (for "dd-MMMM-yyyy" format)
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d-MMMM-yyyy");
    // Convert String to LocalDate
    LocalDate dateOfBirth = LocalDate.parse(dob, formatter);
    LocalDate visitDateLocal = LocalDate.parse(visitDate, formatter);
    // Calculate the age using ChronoUnit
    int age = (int) ChronoUnit.YEARS.between(dateOfBirth, visitDateLocal);
    // Check if the birthday has occurred this year
    if (dateOfBirth.getMonthValue() > visitDateLocal.getMonthValue() ||
            (dateOfBirth.getMonthValue() == visitDateLocal.getMonthValue() &&
                    dateOfBirth.getDayOfMonth() > visitDateLocal.getDayOfMonth())) {
    }else age++;
    return age;
}

public static String getAgeGroup(String dob, String visitDate) {
    int age=dateToAge(dob,visitDate);
    if (age >= 1 && age <= 12) return "1-12";
    else if (age >= 13 && age <= 20) return "13-20";
    else if (age >= 21 && age <= 35) return "21-35";
    else if (age >= 36 && age <= 45) return "36-45";
    else if (age >= 46 && age <= 60) return "46-60";
    else if (age >= 60 && age <= 80) return "60-80";
    else if (age >= 80 && age <= 100) return "80-100";
    else if (age > 100) return "100+";
    else return "Unknown";  // In case of invalid age
}
```

4. **Data Aggregation**:

This method aggregates visit counts by month-year for trend analysis.



```
ublic static ArrayList<NumberOfVisit_Dim> aggregateMonths(ArrayList<NumberOfVisit_Dim> r
  Map<String, Integer> monthVisitMap = new HashMap<>();
  // Step 1: Aggregate counts into the map
  for (NumberOfVisit_Dim nOV : numberOfVisits) {
      String monthYear = separateYearMonth(nOV.getMonthYear());
      int currentCount = monthVisitMap.getOrDefault(monthYear, 0);
      monthVisitMap.put(monthYear, currentCount + nOV.getNumberOfVisits());
  }
  // Step 2: Convert map entries back to list
  ArrayList<NumberOfVisit_Dim> aggregatedList = new ArrayList<>();
  for (Map.Entry<String, Integer> entry : monthVisitMap.entrySet()) {
      aggregatedList.add(new NumberOfVisit_Dim(entry.getKey(), entry.getValue()));
  }
  return aggregatedList;
```

Data Loading

The Load class handles the systematic loading of transformed data into the warehouse:

The loading process:

1. Loads dimension tables before fact tables to maintain referential integrity
2. Uses batch loading for performance optimization
3. Implements null checks to prevent load failures
4. Leverages direct connections to both source and target databases

```java
public class Load {
    public static void loadPatientDim(ArrayList<Patient> patients){
        HospitalDWHControl.insertDimPatientBatch(patients);
    }
    public static void loadVisitDim(ArrayList<Visit_Dim> visitDims){
        if (visitDims!=null)
            HospitalDWHControl.insertFactVisitBatch(visitDims);
    }
    public static void loadSupplier(ArrayList<Supplier_Dim> supplierDims){
        HospitalDWHControl.insertDimSupplierBatch(supplierDims);
    }
    public static void loadMedicineDim(){
        ArrayList<Med_dim> med_dims=OperationalDBControl.selectMedicineRecords();
        if (med_dims != null)
            HospitalDWHControl.insertDimMedicationBatch(med_dims);
    }
    public static void loadOutBreaks(ArrayList<Dim_Outbreak> dimOutbreaks){
        HospitalDWHControl.insertDimOutbreakBatch(dimOutbreaks);
    }
    public static void loadFactMedication(ArrayList<Fact_Medicine> factMedicines){
        HospitalDWHControl.insertFactMedicineBatch(factMedicines);
    }
    public static void loadNumberOfVisit(ArrayList<NumberOfVisit_Dim> factVisitPerMonths) {
        HospitalDWHControl.insertFactVisitPerMonthBatch(factVisitPerMonths);
    }
}
```

Optimization Techniques Used in the Healthcare Data Warehouse Design

Schema-Level Optimizations

1. **Star Schema Implementation** - Centralized fact tables with surrounding dimension tables optimize analytical query performance
2. **Pre-aggregated Fact Tables** - fact_visitpermonth stores pre-calculated monthly visit counts to avoid expensive runtime aggregations
3. **Denormalized Dimensions** - Flattened dimension tables reduce the need for multiple joins during analysis
4. **Integrated Billing Data** - Incorporating billing information directly into fact_visit eliminates additional joins
5. **Derived Attributes** - Pre-calculated fields like age_group and total_cost eliminate runtime calculations

Data Storage Optimizations

6. **Appropriate Data Types** - Using optimal data types (integer for IDs, date for temporal data) for storage efficiency
7. **Date Standardization** - Converting text dates to proper DATE types improves storage and query performance
8. **Missing Value Handling** - Default values for missing data eliminate null-handling overhead

Query Performance Optimizations

9. **Strategic Foreign Keys** - Carefully placed relationships enable efficient join paths
10. **Direct Support for Required Queries** - Schema specifically designed for the three key analytical requirements
11. **Temporal Analysis Support** - Date fields and time period attributes enable efficient time-based filtering

ETL Process Optimizations

12. **Batch Processing** - Loading data in batches reduces database round trips

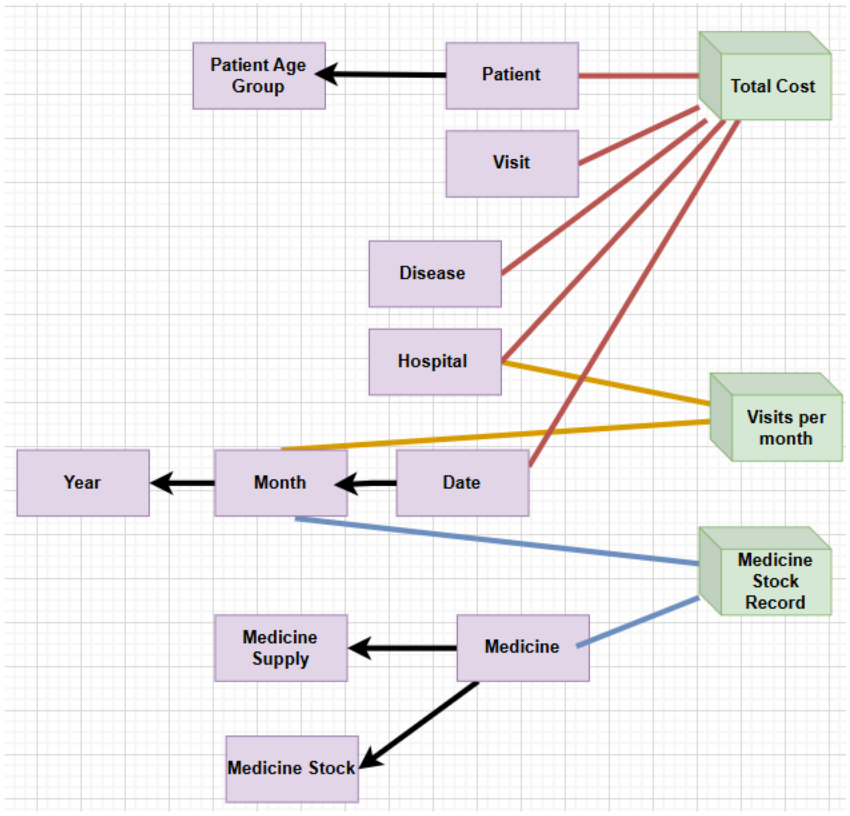
Operational Optimizations

13. **Elimination of Redundant Dimensions** - No separate billing dimension reduces maintenance overhead
14. **Historical Data Summarization** - Detailed recent data with summarized historical data balances storage needs and query performance
15. **Predefined Aggregation Levels** - Age groups and monthly periods provide standardized analysis dimensions

Data Ware-House design

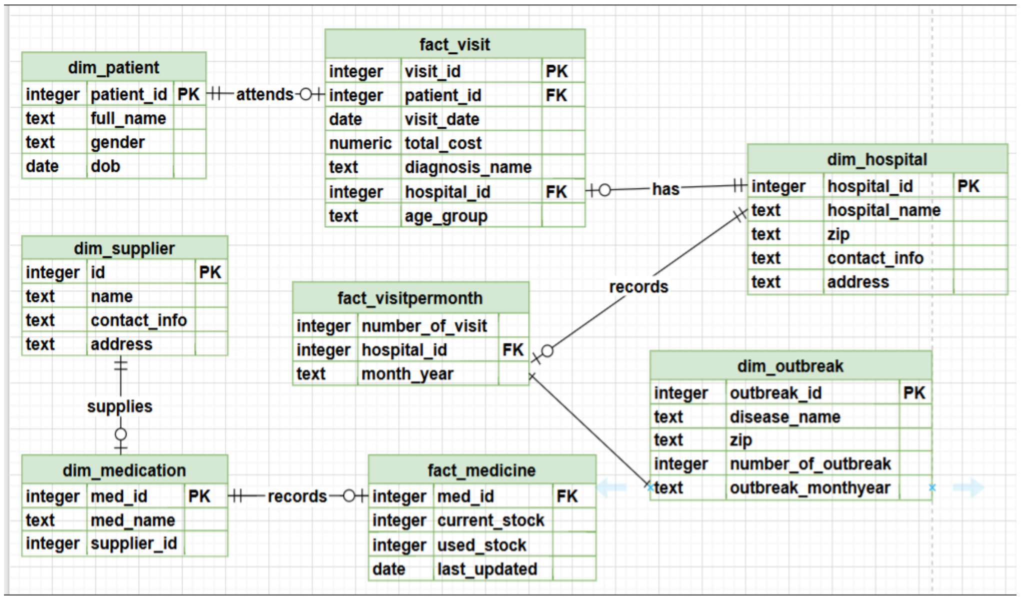
Conceptual design:

This conceptual design visually represents the data warehouse model for a healthcare system. It focuses on analysing total cost, monthly visits, and medicine stock by connecting key dimensions like patient, hospital, disease, and medication.



Logical design

This galaxy schema integrates multiple fact tables—such as visits, medicine stock, and visit counts—with shared dimension tables like patient, hospital, and date. It supports complex healthcare analytics by combining different perspectives such as patient visits, medical supplies, and disease outbreaks.



Assumptions made during the design

Assumption in designing source database:

1. **Patient can only be prescribed by one medicine at a time.**

Assumption in designing Data WareHouse:

1. **Detail billing not needed for analysis.**
2. **Will be scalable as hospital dimension is managed when other hospital data is dumped in data warehouse.**

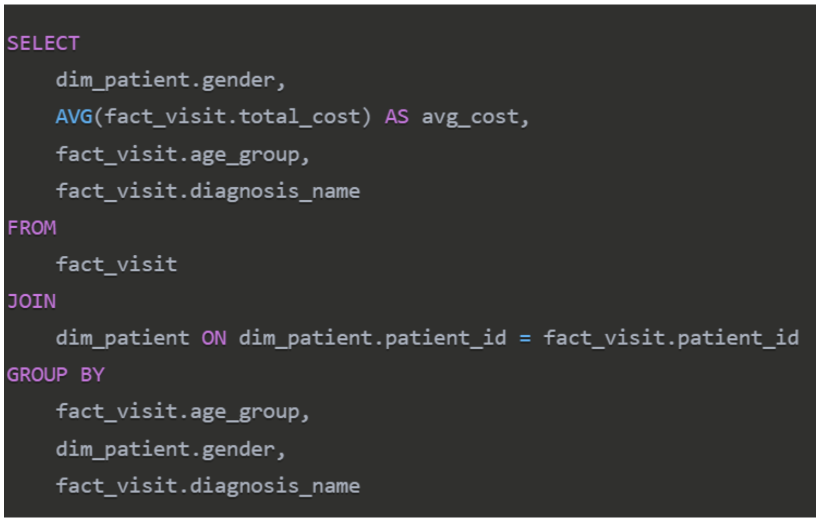
Assumption in designing Data WareHouse:

1. **Base procedure and medication cost will be 1000 if no cost is found.**
2. **Name set to Anonymous if name is not found.**

Queries And Result Analysis

The healthcare data warehouse supports three key analytical scenarios through carefully structured SQL queries:

Query 1: Average treatment cost per diagnosis by age group and gender.

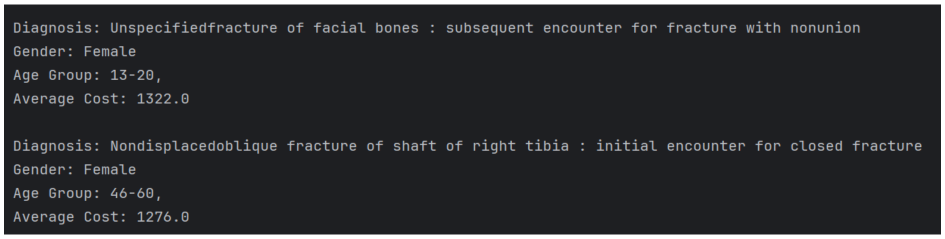


```
SELECT
    dim_patient.gender,
    AVG(fact_visit.total_cost) AS avg_cost,
    fact_visit.age_group,
    fact_visit.diagnosis_name
FROM
    fact_visit
JOIN
    dim_patient ON dim_patient.patient_id = fact_visit.patient_id
GROUP BY
    fact_visit.age_group,
    dim_patient.gender,
    fact_visit.diagnosis_name
```

Analysis: This query provides critical insights for healthcare resource allocation and cost management by:

- Breaking down average treatment costs by gender, age group, and diagnosis
- Enabling targeted cost control measures for specific demographic segments
- Identifying potentially concerning cost variations across patient groups
- Supporting evidence-based decision-making for resource allocation

Sample Results:

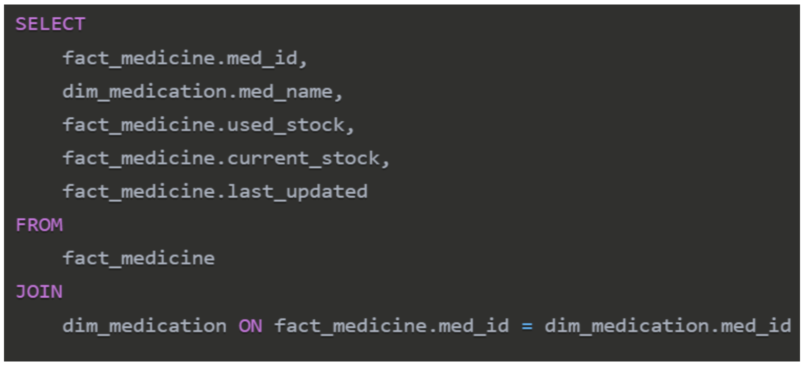


```
Diagnosis: Unspecifiedfracture of facial bones : subsequent encounter for fracture with nonunion
Gender: Female
Age Group: 13-20,
Average Cost: 1322.0

Diagnosis: Nondisplacedoblique fracture of shaft of right tibia : initial encounter for closed fracture
Gender: Female
Age Group: 46-60,
Average Cost: 1276.0
```

These results enable healthcare administrators to identify which demographic segments and diagnoses are driving costs, potentially revealing opportunities for targeted interventions and resource optimization.

Query 2: Medication stock levels and replenishment alerts based on patient demand trends.



```
SELECT
    fact_medicine.med_id,
    dim_medication.med_name,
    fact_medicine.used_stock,
    fact_medicine.current_stock,
    fact_medicine.last_updated
FROM
    fact_medicine
JOIN
    dim_medication ON fact_medicine.med_id = dim_medication.med_id
```

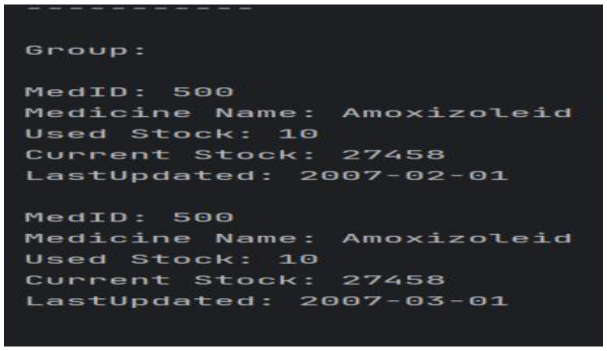
Analysis: This query tracks medication inventory changes over time through a sophisticated two-step process:

1. First, it retrieves the raw inventory data from the fact table
2. Then, the query2Data() method groups results by medication ID and analyzes stock changes across different time periods

This approach provides valuable insights into medication consumption patterns and inventory management by:

- Monitoring usage trends for specific medications
- Identifying potential supply chain issues or unexpected consumption patterns
- Supporting inventory optimization and cost reduction initiatives
- Enabling accurate forecasting of medication needs

Sample Results:



```
-----------

Group:

MedID: 500
Medicine Name: Amoxizoleid
Used Stock: 10
Current Stock: 27458
LastUpdated: 2007-02-01

MedID: 500
Medicine Name: Amoxizoleid
Used Stock: 10
Current Stock: 27458
LastUpdated: 2007-03-01
```

This tracking enables pharmacy managers to identify consumption patterns and ensure appropriate stock levels are maintained, potentially reducing waste and preventing shortages.

Query 3: Correlation between local disease outbreaks and hospital visit spikes over time

```sql
SELECT
    dim_outbreak.disease_name,
    dim_outbreak.outbreak_monthyear,
    dim_outbreak.zip AS outbreak_zip,
    fact_visitPerMonth.number_of_visit,
    dim_outbreak.number_of_outbreak
FROM
    dim_outbreak
JOIN
    fact_visitPerMonth ON dim_outbreak.outbreak_monthyear = fact_visitPerMonth.month_year
ORDER BY
    fact_visitPerMonth.number_of_visit DESC
```

Analysis: This query establishes critical relationships between disease outbreaks and hospital visit volumes by:

- Correlating outbreak data with visit patterns across time periods and geographic areas
- Sorting results by visit volume to highlight the most significant impact events
- Enabling analysis of the relationship between outbreak severity and healthcare utilization
- Supporting public health planning and hospital resource allocation

Sample Results:

```
Disease: Typhoid pneumonia
Outbreak Month: February-2007
Zip: 10003
Visits: 140
Outbreaks: 2407


Disease: Salmonella arthritis
Outbreak Month: February-2007
Zip: 10001
Visits: 140
Outbreaks: 6900
```

This correlation analysis helps healthcare administrators anticipate demand surges based on public health data, potentially improving resource allocation during outbreak situations.

Performance Considerations

The query implementation demonstrates several key performance optimization strategies:

1. Strategic Joining: All queries use direct joins between fact and dimension tables without complex multi-table joins
2. Effective Grouping: Query 1 uses multi-dimensional grouping to create meaningful analytical segments
3. Result Sorting: Query 3 uses ORDER BY to prioritize high-impact results for immediate attention
4. Post-Query Processing: Query 2 implements Java-based post-processing to perform more complex temporal analysis
5. Prepared Statements: All queries use PreparedStatement objects for optimal database interaction