Software Design Document Steps 1 and 2

1. Gather requirements and initial information: Gather all relevant information about the project, including project requirements and specifications, input from stakeholders about what they expect, technical constraints and limitations, and existing system documentation (if applicable). Starting with a solid foundation of information ensures that your design document accurately reflects the project's goals and constraints.

**Project Requirements and Specifications**

* Parse and store medical insurance records from a CSV dataset.
* Perform statistical analysis (count, mean, standard deviation, percentiles, etc.) on numerical attributes: age, BMI, children, charges.
* Generate horizontal and vertical text-based histograms using Java Collections only.
* Evaluate fairness and test hypotheses based on domain-specific logic.
* Implement simple linear regression and correlation analysis without external libraries.

**Stakeholder Input**

* Ensure accurate statistical calculations and readable console output.
* Avoid use of Python or third-party libraries—Java Collections only.
* Maintain modular, testable code with clear separation of concerns.
* Support hypothesis testing and regression analysis for exploratory insights.

**Technical Constraints**

* No use of numpy, pandas, matplotlib, or similar libraries.
* Must use Java and only built-in Collections (e.g., ArrayList, HashMap, TreeMap).
* Console-based output only—no GUI or plotting libraries.

**Existing Documentation**

* Kaggle dataset with labeled columns: age, sex, BMI, children, smoker, region, charges.
* Class diagram (Visio or UML) outlining core components: Records, Data Storage, Analysis, StatisticsEngine, Histogram, InsuranceApp.

2. Outline the system architecture: With the requirements identified, outline the overall system architecture. Identify major components and subsystems, define how these components relate to each other, and make high-level design decisions. Choose appropriate design patterns and architectural styles that will guide the rest of the development process.

**Identify Major components and Subsystems**

* Core classes
  + Records: encapsulates one row of data
  + Data Storage: stores and manages a list of Records objects.
  + StatisticsEngine: performs calculations like mean, standard deviation, percentiles.
  + Histogram: renders horizontal and vertical histograms.
  + Analysis: runs fairness checks, hypothesis tests, and regression.
  + InsuranceApp: main driver class that orchestrates the workflow.

**Define Relationships Between Components**

* InsuranceApp uses Data Storage to load records.
* InsuranceApp invokes Analysis and Histogram for insights.
* Analysis depends on Statistics for numerical computations.
* Histogram accesses data via Data Storage or receives filtered lists.

**Specify how classes interact with (e.g. InsuranceApp uses Analysis, which depends on Data Storage)**

* InsuranceApp acts as the main driver and entry point of the application. It:
  + Instantiates Data Storage to load and manage records.
  + Creates an Analysis object, passing in the Data Storage instance.
  + Calls methods from Analysis to perform statistical evaluations and hypothesis testing.
  + Invokes Histogram to display visual summaries of attributes like age, BMI, and smoker status.
* Analysis depends on Data Storage to access the full dataset. It:
  + Retrieves records from Data Storage to perform fairness checks, comparisons, and regression analysis.
  + Uses methods from Statistics to compute metrics like mean, standard deviation, and percentiles.
* Statistics provides static utility methods. It:
  + Accepts lists of numerical values (e.g., ages, charges) and returns computed results.
  + Is used by Analysis to support data-driven decisions and hypothesis testing.
* Histogram is used by InsuranceApp or Analysis to render text-based visualizations. It:
  + Accepts lists of integers or categories and prints horizontal or vertical histograms to the console.
* Data Storage manages a collection of Records objects. It:
  + Provides methods to add records, retrieve the full list, and query size.
  + Serves as the central data repository for all other classes.
* Records encapsulates a single row of data. It:
  + Stores attributes like age, BMI, children, charges, smoker status, and region.
  + Provides getter methods for each attribute, enabling access by Analysis, Statistics, and Histogram.

**Class Diagram of InsuranceApp System:** Use UML Class Diagrams to illustrate associations, dependencies, and multiplicities

A diagram of a computer

AI-generated content may be incorrect.

* The diagram above displays the relationship(s) between the classes created in the InsuranceApp
* Key Relationship: One-to-Many Between Data Storage and Records
  + To show that Data Storage holds many Records instances
  + Data Storage is Responsible for
    - Adding new records
    - Providing access to all records
    - Supporting queries and iteration for Analysis
* Analysis is the logic layer, performing evaluations and hypothesis testing. It depends on Data Storage to access the dataset.
* Data Storage is the data layer, managing a collection of Records objects. It’s the central repository that other classes query.
* Records is the data model, encapsulating individual entries with getters for each attribute.
* Statistics and Histogram are utility classes—they don’t store data but provide reusable methods for computation and visualization

Why Does This Matter?

* The relationships are loosely coupled: each class has a focused role and interacts through well-defined interfaces.
* The architecture supports modularity and testability: you can test Analysis independently by mocking Data Storage, or swap out Histogram without affecting core logic.
* The flow is top-down: InsuranceApp → Analysis → Data Storage → Records, with Statistics and Histogram assisting as needed.