Telecom Smart Plan Recommender

Project Objective

Goal: Build an **OOP-based Telecom Smart Plan Recommender** that helps ConnectTel customers avoid bill shocks by recommending the best plan based on:

1. Customer 30-day usage (data, voice, SMS).
2. OTT app requirements.
3. Plan costs, including normalized pricing for different validities and overages.

Deliverables:

- Models for **plans, usage, OTT requirements**.
- Price calculation logic that **normalizes plans** to a 30-day window.
- **Recommendation engine** that filters plans by OTT and selects the cheapest viable plan.
- **Readable output** showing plan breakdown and final recommendation.

Domain Model (OOP)

We can structure this with **clean inheritance and encapsulation**.

1. Classes

1. **`Plan` (abstract/base)**
    a. Attributes:
        i. `name`
        ii. `price`
        iii. `validity_days`
        iv. `data_quota` (GB or MB/day or total)
        v. `voice_quota` (mins)
        vi. `sms_quota`
        vii. `overage_rates` (data/voice/sms)
        viii. `ott_bundle` (list of OTT enum)
    b. Methods:

        i. `calculate_quote(usage: Usage) -> PlanQuote` *(core pricing logic with normalization)*
        ii. `is_eligible(ott_req: OTTRequirement) -> bool`
2. **Concrete Plan Classes** (8 plans: `BasicLite`, `Saver30`, `UnlimitedTalk30`, etc.)
    a. Override any plan-specific pricing rules if needed.
3. **`Usage`**
    a. Attributes:
        i. `voice_minutes`
        ii. `sms_count`
        iii. `data_mb`
4. **`OTTRequirement`**
    a. Attributes (booleans): `netflix`, `prime`, `hotstar`, `spotify`
5. **`PlanQuote`**
    a. Attributes:
        i. `plan_name`
        ii. `total_cost`
        iii. `breakdown` (dictionary or formatted string: rental, data overage, voice extra, SMS extra)
6. **`PlanOptimizer`**
    a. Methods:
        i. `filter_by_ott(plans, requirement) -> List[Plan]`
        ii. `recommend(plans, usage, requirement) -> PlanQuote`
7. **Optional: OTT Enum**
    a. Values: `NETFLIX, PRIME, HOTSTAR, SPOTIFY`

Steps to Implement

Step 1: Model the Domain

- Create abstract `Plan` class.
- Define concrete plan subclasses with plan-specific data/voice/SMS/OTT quotas.
- Define `Usage` and `OTTRequirement`.

Step 2: Implement Pricing Logic

- Normalize plans to **30-day equivalent**:

    - `normalized_price = plan_price * (30 / validity_days)`
    - Adjust quotas accordingly:
        - Daily quota → 30-day total: `daily_quota * 30`
        - Total quota → prorated if validity ≠ 30
- Calculate overages for:
    - Data (`overage_rate per MB`)
    - Voice (`overage_rate per min`)
    - SMS (`overage_rate per SMS`)

Step 3: Check OTT Eligibility

- Compare plan OTT bundle with required OTTs.
- Only retain plans that **cover all requested OTTs**.

Step 4: Generate Quotes

- Each plan should produce a **`PlanQuote`** for given usage.
- Include:
    - Rental cost (normalized)
    - Data overage cost
    - Voice overage cost
    - SMS overage cost
    - Total

Step 5: Optimize & Recommend

- Filter plans by OTT eligibility.
- Pick plan with **lowest total cost**:
    - If tie → choose plan with more OTTs or simpler pricing (document choice).

Code:

```python
from typing import List
from models import Usage, OTTRequirement
from plans import (
    Plan, BasicLite, Saver30, UnlimitedTalk30, DataMax20,
    StudentStream56, FamilyShare30, DataMaxPlus30, PremiumUltra30
)
from optimizer import PlanOptimizer

def get_user_usage() -> Usage:
    print("\n--- Please enter your estimated 30-day usage ---")
    while True:
        try:
            voice_mins = int(input("Enter total voice minutes: "))
            sms_count = int(input("Enter total SMS count: "))
            data_gb = float(input("Enter total data usage in GB: "))
            # Convert GB to MB for calculations
            data_mb = int(data_gb * 1024)
            return Usage(voice_mins=voice_mins, sms_count=sms_count, data_mb=data_mb)
        except ValueError:
            print("Invalid input. Please enter numbers only. Let's try again.")

def get_ott_requirements() -> OTTRequirement:
    print("\n--- Do you need the following OTT services? (yes/no) ---")

    def ask_yes_no(prompt: str) -> bool:
        while True:
            answer = input(f"{prompt}: ").lower().strip()
            if answer in ["yes", "y"]:
                return True
            if answer in ["no", "n"]:
                return False
            print("Invalid input. Please enter 'yes' or 'no'.")

    needs_hotstar = ask_yes_no("Hotstar?")
    needs_spotify = ask_yes_no("Spotify?")
    needs_amazon_prime = ask_yes_no("Amazon Prime?")
```

```python
def main():
    available_plans: List[Plan] = [
        BasicLite(),
        Saver30(),
        UnlimitedTalk30(),
        DataMax20(),
        StudentStream56(),
        FamilyShare30(),
        DataMaxPlus30(),
        PremiumUltra30(),
    ]

    optimizer = PlanOptimizer(available_plans)

    print("================================================")
    print("Welcome to the Telecom Smart Plan Recommender")
    print("================================================")

    customer_usage = get_user_usage()
    customer_ott = get_ott_requirements()

    recommendation = optimizer.recommend(customer_usage, customer_ott)

    if recommendation:
        print("\n--- Recommendation ---")
        print(f"The best plan for you is: **{recommendation.plan_name}**")
        print("Reason: It's the cheapest plan that meets your OTT requirements.")
        print(recommendation.breakdown())
    else:
        print("\nSorry, no suitable plan was found based on your requirements.")


if __name__ == "__main__":
    main()
```

```python
from dataclasses import dataclass
from typing import Set


@dataclass(frozen=True)
class Usage:
    voice_mins: int
    sms_count: int
    data_mb: int

@dataclass(frozen=True)
class OTTRequirement:
    needs_hotstar: bool = False
    needs_spotify: bool = False
    needs_amazon_prime: bool = False
    needs_netflix: bool = False

    def get_required_services(self) -> Set[str]:
        required = set()
        if self.needs_hotstar:
            required.add("Hotstar")
        if self.needs_spotify:
            required.add("Spotify")
        if self.needs_amazon_prime:
            required.add("Amazon Prime")
        if self.needs_netflix:
            required.add("Netflix")
        return required

@dataclass
class PlanQuote:
    plan_name: str
    total_cost: float
    rental_30d: float
    data_overage_cost: float = 0.0
    voice_overage_cost: float = 0.0
```

```python
from typing import List, Optional
from models import Usage, OTTRequirement, PlanQuote
from plans import Plan

class PlanOptimizer:
    def __init__(self, all_plans: List[Plan]):
        self.all_plans = all_plans

    def recommend(self, usage: Usage, ott_reqs: OTTRequirement) -> Optional[PlanQuote]:
        print("--- Calculating Costs for All Plans ---")
        all_quotes = [plan.calculate_quote(usage) for plan in self.all_plans]
        for quote in all_quotes:
            print(quote.breakdown())

        print("\n--- Filtering by OTT Requirements ---")
        eligible_plans = [
            plan for plan in self.all_plans if plan.meets_ott_requirements(ott_reqs)
        ]

        if not eligible_plans:
            print("No plans meet your OTT requirements.")
            return None

        print(f"Found {len(eligible_plans)} eligible plan(s): {[p.name for p in eligible_plans]}")

        eligible_quotes = [plan.calculate_quote(usage) for plan in eligible_plans]

        best_quote = min(
            eligible_quotes,
            key=lambda q: (q.total_cost, -len(self._get_plan_by_name(q.plan_name).ott_bundle))
        )

        return best_quote

    def _get_plan_by_name(self, name: str) -> Plan:
        for plan in self.all_plans:
```

Input Image:

```
==============================================
Welcome to the Telecom Smart Plan Recommender
==============================================

--- Please enter your estimated 30-day usage ---
Enter total voice minutes: 500
Enter total SMS count: 100
Enter total data usage in GB: 8

--- Do you need the following OTT services? (yes/no) ---
Hotstar?: yes
Spotify?: no
Amazon Prime?: yes
Netflix?: yes
```

Output Image:

```
--- Filtering by OTT Requirements ---
Found 1 eligible plan(s): ['Premium Ultra 30']

--- Recommendation ---
The best plan for you is: **Premium Ultra 30**
Reason: It's the cheapest plan that meets your OTT requirements.
-> Premium Ultra 30:
   - 30-Day Rental: ₹2999.00
   - Data Overage:  ₹0.00
   - Voice Overage: ₹0.00
   - SMS Overage:   ₹0.00
   --------------------------------
   => Total 30-Day Cost: ₹2999.00
```

CityLink Farebox

Project Objective

Goal: Build an **OOP-based Fare Computation Engine** for CityLink metro taps that:

1. Reproduces historical fares exactly by inferring rules from tap logs.
2. Provides a flexible system where fare rules can be toggled on/off for testing.
3. Applies these rules to **new tap data** to compute the correct charges automatically.

Deliverables:

- Documented **fare rules hypothesis** derived from historical data.
- **OOP class design** encapsulating rules and fare calculation.
- A **TariffEngine** that applies rules in sequence and produces per-tap charges.
- Ability to **toggle rules** to test alternate hypotheses.

Domain Model (OOP)

We can organize this cleanly with **rule encapsulation and extensibility**.

1. Classes

1. **`Tap`**
    a. Attributes:
        i. `datetime` (timestamp)
        ii. `line` (G, R, Y, etc.)
        iii. `station_code`
        iv. `fare_charged` (from log, optional for testing)
2. **`FareRule` (interface or abstract)**
    a. Method:
        i. `apply(previous_taps: List[Tap], current_tap: Tap) -> double`
    b. Attribute:
        i. `enabled` (boolean)
    c. Purpose: encapsulate a single pricing logic. Can be toggled on/off.

3. **Concrete FareRule Classes**
    a. **`BaseFareRule` (R1)** → applies base fare.
    b. **`PeakPeriodRule` (R2)** → adjusts fare for peak hours.
    c. **`TransferWindowRule` (R3)** → makes taps free within 30 minutes of last paid tap.
    d. **`NightDiscountRule` (R4)** → applies 20% discount 10 pm–midnight.
    e. **`PostMidnightDiscountRule` (R5)** → applies 35% discount midnight–4 am.
4. **`TariffEngine`**
    a. Attributes:
        i. `rules: List[FareRule]`
    b. Method:
        i. `compute_fare(taps: List[Tap]) -> List[TapCharge]`
    c. Responsibility: chain rules in order, compute final fare per tap.

5. **`TapCharge`** (optional helper)
    a. Attributes:
        i. `tap` (Tap)
        ii. `computed_fare`
        iii. `breakdown` (which rules contributed to fare)

Steps to Implement

Step 1: Infer Fare Rules

- Analyze tap log:
    - Look for repeating fares (R1)
    - Identify time-based adjustments (R2, R4, R5)
    - Check for free transfers within 30 minutes (R3)
- Document **hypothesis** in a concise one-page brief.

Step 2: Model the Domain

- Implement `Tap` and optional `TapCharge` classes.
- Implement **abstract `FareRule`** interface.
- Implement **concrete rules** (R1–R5) individually.

Step 3: Build Tariff Engine

- `TariffEngine` holds **list of rules** in priority order.
- Iterates over tap log:
    - Passes each tap through **all enabled rules**.
    - Combines results to compute final fare.

Step 4: Enable/Disable Rules

- Each `FareRule` has a boolean `enabled`.
- TariffEngine applies only enabled rules.
- Allows **A/B testing of rule hypotheses**.

Step 5: Validate Against Tap Log

- Compute fares for historical taps.
- Compare with `fare_charged` in log.
- Refine rules if necessary.

Step 6: Apply to New Taps

- Use same engine to compute fares for new user tap sequences.

Code:

```python
from tap import Tap
from tarrif_engine import TarrifEngine

config={
    "R1":True,
    "R2":True,
    "R3":True,
    "R4":True,
    "R5":True,
}

engine=TarrifEngine(config)
tap_data=[
    ("07-01 07:20", "G","BD"),
    ("07-01 08:01", "G","NC"),
    ("07-01 08:30", "R","YH"),
    ("07-01 08:32", "Y","YH"),
    ("07-01 10:01", "R","KL"),
    ("07-01 10:28", "Y","NC"),
    ("07-01 10:32", "Y","JT"),
    ("07-01 14:36", "G","NC"),
    ("07-01 22:15", "Y","BD"),
    ("07-01 23:58", "G","NC"),
    ("07-02 00:45", "X","NC"),
    ("07-02 01:10", "G","BD"),
    ("07-02 04:01", "G","BD"),
    ("07-02 13:05", "Y","JT"),
    ("07-02 13:15", "G","KL"),
    ("07-02 13:36", "G","JT"),
    ("07-02 18:02", "Y","BD"),
    ("07-02 18:18", "Y","NC"),
    ("07-02 20:01", "G","KL"),
    ("07-02 20:15", "R","YT"),
    ("07-02 22:02", "Y","KL"),
    ("07-02 23:15", "G","BD"),
    ("07-03 00:20", "R", "NC")
```

```python
from base_fare_rule import BaseFareRule
from peak_period_rule import PeakPeriodRule
from transfer_window_rule import TransferWindowRule
from night_discount_rule import NightDiscountRule
from post_midnight_discount_rule import PostMidNightDiscountRule

class TarrifEngine:
    def __init__(self, config):
        self.config=config
        self.rules=[]

        if config.get("R1"):self.rules.append(BaseFareRule())
        if config.get("R2"):self.rules.append(PeakPeriodRule())
        if config.get("R3"):self.rules.append(TransferWindowRule())
        if config.get("R4"):self.rules.append(NightDiscountRule())
        if config.get("R5"):self.rules.append(PostMidNightDiscountRule())

    def compute_fare(self, tap, history):
        fare=0.0
        for rule in self.rules:
            fare=rule.apply(tap,history,fare)
        return fare
```

Input Image:

```
tap_data=[
    ("07-01 07:20", "G","BD"),
    ("07-01 08:01", "G","NC"),
    ("07-01 08:30", "R","YH"),
    ("07-01 08:32", "Y","YH"),
    ("07-01 10:01", "R","KL"),
    ("07-01 10:28", "Y","NC"),
    ("07-01 10:32", "Y","JT"),
    ("07-01 14:36", "G","NC"),
    ("07-01 22:15", "Y","BD"),
    ("07-01 23:58", "G","NC"),
    ("07-02 00:45", "X","NC"),
    ("07-02 01:10", "G","BD"),
    ("07-02 04:01", "G","BD"),
    ("07-02 13:05", "Y","JT"),
    ("07-02 13:15", "G","KL"),
    ("07-02 13:36", "G","JT"),
    ("07-02 18:02", "Y","BD"),
    ("07-02 18:18", "Y","NC"),
    ("07-02 20:01", "G","KL"),
    ("07-02 20:15", "R","YT"),
    ("07-02 22:02", "Y","KL"),
    ("07-02 23:15", "G","BD"),
    ("07-03 00:20", "R", "NC")
]
```

Output Image:

```
C:\Users\akashraj.r\Desktop\Project-2>python main-2.py
07-01 07:20|Line:G|Station:BD|Fare:25
07-01 08:01|Line:G|Station:NC|Fare:37.5
07-01 08:30|Line:R|Station:YH|Fare:0.0
07-01 08:32|Line:Y|Station:YH|Fare:0.0
07-01 10:01|Line:R|Station:KL|Fare:25
07-01 10:28|Line:Y|Station:NC|Fare:0.0
07-01 10:32|Line:Y|Station:JT|Fare:0.0
07-01 14:36|Line:G|Station:NC|Fare:25
07-01 22:15|Line:Y|Station:BD|Fare:20.0
07-01 23:58|Line:G|Station:NC|Fare:20.0
07-02 00:45|Line:X|Station:NC|Fare:16.25
07-02 01:10|Line:G|Station:BD|Fare:0.0
07-01 08:32|Line:Y|Station:YH|Fare:0.0
07-01 10:01|Line:R|Station:KL|Fare:25
07-01 10:28|Line:Y|Station:NC|Fare:0.0
07-01 10:32|Line:Y|Station:JT|Fare:0.0
07-01 14:36|Line:G|Station:NC|Fare:25
07-01 22:15|Line:Y|Station:BD|Fare:20.0
07-01 23:58|Line:G|Station:NC|Fare:20.0
07-02 00:45|Line:X|Station:NC|Fare:16.25
07-02 01:10|Line:G|Station:BD|Fare:0.0
07-01 14:36|Line:G|Station:NC|Fare:25
07-01 22:15|Line:Y|Station:BD|Fare:20.0
07-01 23:58|Line:G|Station:NC|Fare:20.0
07-02 00:45|Line:X|Station:NC|Fare:16.25
07-02 01:10|Line:G|Station:BD|Fare:0.0
07-02 00:45|Line:X|Station:NC|Fare:16.25
```