Ćwiczenie: Enkapsulacja „obcych" struktur i logiki w obiekcie (1. przypadek)

W klasie **tools.ShortageFinder** w metodzie statycznej **findShortages** wprowadź w kod adapter opakowujący operacje na zmiennej lokalnej:

Map<LocalDate, List<ProductionEntity>> outputs = **new** HashMap<>();

1. Utwórz nową klasę **ProductionOutputs** i zamknij w niej powyższą kolekcję jako prywatne pole klasy. Klasę **ProductionOutputs** umieść w nowym pakiecie **shortages.**
2. Zadbaj by część logiki ShortageFinder związana z **konstrukcją** i **odczytem** danych z tej struktury była enkapsułowana wewnątrz nowej klasy adaptera.
3. Zaproponuj takie API w klasie **ProductionOutputs** by jej użycie w ShortageFinder było możliwie wygodne i zwięzłe.

Wybrany kod ShortageFinder-a:

```
public static List<ShortageEntity> findShortages(LocalDate today, int daysAhead, CurrentStock stock,
                                                 List<ProductionEntity> productions, List<DemandEntity> demands) {

List<LocalDate> dates = Stream.iterate(today, date -> date.plusDays(1))
        .limit(daysAhead)
        .collect(toList());

                                                                                                    Konstrukcja
String productRefNo = null;
HashMap<LocalDate, List<ProductionEntity>> outputs = new HashMap<>();
for (ProductionEntity production : productions) {
    if (!outputs.containsKey(production.getStart().toLocalDate())) {
        outputs.put(production.getStart().toLocalDate(), new ArrayList<>());
    }
    outputs.get(production.getStart().toLocalDate()).add(production);
    productRefNo = production.getForm().getRefNo();
}
HashMap<LocalDate, DemandEntity> demandsPerDay = new HashMap<>();
for (DemandEntity demand1 : demands) {
    demandsPerDay.put(demand1.getDay(), demand1);
}

long level = stock.getLevel();

List<ShortageEntity> gap = new LinkedList<>();
for (LocalDate day : dates) {
    DemandEntity demand = demandsPerDay.get(day);
    if (demand == null) {
        for (ProductionEntity production : outputs.get(day)) {                                      Odczyt
            level += production.getOutput();
        }
        continue;
    }
    long produced = 0;                                                                              Odczyt
    for (ProductionEntity production : outputs.get(day)) {
        produced += production.getOutput();
    }

long levelOnDelivery;
if (Util.getDeliverySchema(demand) == DeliverySchema.atDayStart) {
    levelOnDelivery = level - Util.getLevel(demand);
} else if (Util.getDeliverySchema(demand) == DeliverySchema.tillEndOfDay) {
```

Ćwiczenie: Enkapsulacja „obcych" struktur i logiki w obiekcie (2. przypadek)

W klasie **tools.ShortageFinder** w metodzie statycznej **findShortages** wprowadź w kod adapter opakowujący operacje na zmiennej lokalnej:

HashMap<LocalDate, DemandEntity> demandsPerDay = **new** HashMap<>();

1. Utwórz nową klasę **Demands** i zamknij w niej powyższą kolekcję jako prywatne pole klasy. Zaproponuj takie API w klasie **Demands** by jej użycie w ShortageFinder było możliwie wygodne i zwięzłe.
2. Zadbaj by część logiki ShortageFinder związana z **konstrukcją** i **odczytem** danych z tej struktury była enkapsułowana wewnątrz nowej klasy adaptera.
3. Ponieważ tym razem wyczytujemy kilka różnych pól dla wskazanego dnia rozważ dwa warianty implementacji:
   - Wiele metod getXYZ(LocalDate) zwracając proste wartości jak long czy enum
   - Jedna metoda get(LocalDate) zwracającą instancję nowej klasy DailyDemand reprezentującą wszystkie wartości dla jednego dnia, enkapsułującą DemandEntity
     wybież z nich ten, który Twoim zdaniem jest lepszy.

Nie analizuj implementacji metod Util.getDeliverySchema(demand), Util.*getLevel*(demand) posłuż się tymi samymi metodami wewnątrz nowych klas **Demands** i opt. **DailyDemand**.

Wybrany kod ShortageFinder-a:

Konstrukcja

```
HashMap<LocalDate, DemandEntity> demandsPerDay = new HashMap<>();
for (DemandEntity demand1 : demands) {
   demandsPerDay.put(demand1.getDay(), demand1);
}
```

```
long level = stock.getLevel();

List<ShortageEntity> gap = new LinkedList<>();
for (LocalDate day : dates) {
```

Odczyt

```
   DemandEntity demand = demandsPerDay.get(day);
   if (demand == null) {
      for (ProductionEntity production : outputs.get(day)) {
         level += production.getOutput();
      }
      continue;
   }
   long produced = 0;
   for (ProductionEntity production : outputs.get(day)) {
      produced += production.getOutput();
   }

   long levelOnDelivery;
```

Odczyty

```
   if (Util.getDeliverySchema(demand) == DeliverySchema.atDayStart) {
      levelOnDelivery = level - Util.getLevel(demand);
   } else if (Util.getDeliverySchema(demand) == DeliverySchema.tillEndOfDay) {
      levelOnDelivery = level - Util.getLevel(demand) + produced;
   } else if (Util.getDeliverySchema(demand) == DeliverySchema.every3hours) {
      // TODO WTF ?? we need to rewrite that app :/
      throw new UnsupportedOperationException();
   } else {
      // TODO implement other variants
```

Ćwiczenie: Enkapsulacja edytowanych struktur i logiki w obiekcie na przykładzie danych własnych modułu

W klasie **tools.ShortageFinder** w metodzie statycznej **findShortages** wprowadź w kod klasę budowniczego opakowującego operacje na dotychczasowej zmiennej lokalnej:

List<ShortageEntity> gap = **new** LinkedList<>();

Utwórz nową klasę **ShortageBuilder** i zamknij w niej powyższą kolekcję jako prywatne pole klasy. Zaproponuj takie API w klasie **ShortageBuilder** by jej użycie w ShortageFinder było możliwie wygodne i zwięzłe. Klasę **ShortageBuilder** umieść w pakiecie **shortages.** Zadbaj by część logiki ShortageFinder związana z **konstrukcją** i **kumulacją** danych w tej strukturze była enkapsułowana wewnątrz nowej klasy **buildera**.

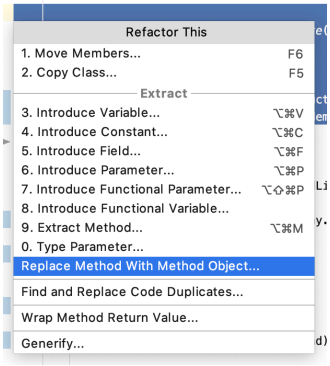
```
long level = stock.getLevel();
List<ShortageEntity> gap = new LinkedList<>();
for (LocalDate day : dates) {
    DailyDemand demand = demandsPerDay.get(day);
    if (demand == null) {
        level += outputs.get(day);
        continue;
    }
    long produced = outputs.get(day);

    long levelOnDelivery;
    if (demand.getDeliverySchema() == DeliverySchema.atDayStart) {
        levelOnDelivery = level - demand.getLevel();
    } else if (demand.getDeliverySchema() == DeliverySchema.tillEndOfDay) {
        levelOnDelivery = level - demand.getLevel() + produced;
    } else if (demand.getDeliverySchema() == DeliverySchema.every3hours) {
        // TODO WTF ?? we need to rewrite that app :/
        throw new UnsupportedOperationException();
    } else {
        // TODO implement other variants
        throw new UnsupportedOperationException();
    }

    if (levelOnDelivery < 0) {
        ShortageEntity entity = new ShortageEntity();
        entity.setRefNo(outputs.getProductRefNo());
        entity.setFound(LocalDate.now());
        entity.setAtDay(day);
        entity.setMissing(-levelOnDelivery);
        gap.add(entity);
    }
    long endOfDayLevel = level + produced - demand.getLevel();
    // TODO: ASK accumulated shortages or reset when under zero?
    level = endOfDayLevel >= 0 ? endOfDayLevel : 0;
}
```

Inicjalizacjalizaacja kolekcji

Kumulacja danych w kolekcji

Ćwiczenie: Wprowadzenie obiektu dziedzinowego przy pomocy Method Object

W klasie **tools.ShortageFinder** z metody statycznej **findShortages** wyekstrahuj wzorzec Method Object.
Zaznacz cały blok oznaczony czerwoną ramką oraz
Użyj funkcji IntelliJ-a:
„Refactor This" > „Replace Method with Method Object".



Nowej klasie nadaj nazwę „**ShortagePrediction**".

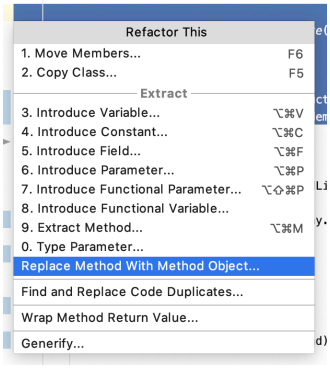
```
public static List<ShortageEntity> findShortages(LocalDate today, int daysAhead, CurrentStock stock,
                                                 List<ProductionEntity> productions, List<DemandEntity> demands) {

    List<LocalDate> dates = Stream.iterate(today, date -> date.plusDays(1))
            .limit(daysAhead)
            .collect(toList());

    ProductionOutputs outputs = new ProductionOutputs(productions);
    Demands demandsPerDay = new Demands(demands);

    long level = stock.getLevel();
    ShortageBuilder shortages = ShortageBuilder.builder(LocalDate.now(), outputs.getProductRefNo());
    for (LocalDate day : dates) {
        if (demands.anyForDate(day)) {
            Demands.Demand demand = demands.get(day);
            long produced = outputs.get(day);
            long levelOnDelivery = demand.calculate(level, produced);

            if (levelOnDelivery < 0) {
                shortages.add(day, levelOnDelivery);
            }
            long endOfDayLevel = level + produced - demand.getLevel();
            level = max(endOfDayLevel, 0);
        } else {
            level += outputs.get(day);
        }
    }
    return shortages.build();
}
```

Ten refactoring zawsze wymaga ręcznego uporządkowania wynikowego kodu:

- Zmień nazwę nowej metody invoke() na predict().
- Przenieś nową klasę do pakietu **shortages**.
- Rozdziel **new** ShortagePrediction() od wywołania metody predict() do osobnych statementów.

Ćwiczenie: Wprowadzenie repozytorium przy pomocy Method Object

W klasie **tools.ShortageFinder** z metody statycznej **findShortages** wyekstrahuj wzorzec Method Object.
Zaznacz cały blok oznaczony czerwoną ramką oraz
Użyj funkcji IntelliJ-a:
„Refactor This” > „Replace Method with Method Object”.



Nowej klasie nadaj nazwę **„ShortagePredictionFactory”**.

```
public static List<ShortageEntity> findShortages(LocalDate today, int daysAhead, CurrentStock stock,
                                                  List<ProductionEntity> productions, List<DemandEntity> demands) {
    List<LocalDate> dates = Stream.iterate(today, date -> date.plusDays(1))
            .limit(daysAhead)
            .collect(toList());

    ProductionOutputs outputs = new ProductionOutputs(productions);
    Demands demandsPerDay = new Demands(demands);

    ShortagePrediction shortages = new ShortagePrediction(stock, dates, outputs, demandsPerDay);

    return shortages.predict();
}
```

Tak jak w poprzednim zadaniu uporządkuj wynikowy kod.