Ćwiczenie: Zaproponuj refactoring testu

Zaproponuj listę zmian, które chciałbyś wykonać w klasie **tools.ShortageFinderTest**.

Możesz „zaszaleć” z refactoringiem - zakładamy nieograniczony czas na realizację refactoringu.
„Napiszmy to od nowa” nie wchodzi w grę.

Ćwiczenie: Zaimplementuj scenariusze testowe

W osobnej klasie testowej zaimplementuj poniższe **proste scenariusze**:
1. Plan produkcyjny na jeden dzień, brak zapotrzebowań, brak produktu na stoku.
2. Produkcja nie jest planowana, jest zapotrzebowanie na jutro, produktu jest na stoku w ilości równej zapotrzebowaniu.
3. Produkcja nie jest planowana, jest zapotrzebowanie na jutro, brak produktu na stoku.

W osobnej klasie testowej zaimplementuj scenariusze, które weryfikują następujące **cechy algorytmu:**
1. Niedobory zapisywane są jako liczba dodatnia.
2. Niedobory nie akumulują się z dnia na dzień, to znaczy jeśli brakuje w kolejnych dniach:
   0 szt, 1000 szt, 1000 szt, 0 szt, 0 szt, …
   to zostaną zwrócone tylko 2 niedobory po 1000 szt
3. Output dla dnia jest sumą wszystkich produkcji danego produktu z danego dnia

Ćwiczenie: Wady testów

Oceń klasę testową **tools.ShortageFinderTest** oraz testy z poprzedniego ćwiczenia.

Zakładając, że w realnym systemie mieli byśmy ok 200 podobnej jakości testów wypisz jakie wady i problemy dostrzegasz. Jak tego typu testy będą się zachowywać podczas dalszego rozwoju oprogramowania czy refaktoryzacji.

Ćwiczenie: Poprawa designu testów przy pomocy Assert Object

Popraw design klasy testowej **tools.ShortageFinderTest**.

Zaimplementuj nową klasę **ShortagesAssert**, której rolą będzie wygodne weryfikowanie cech kolekcji: **List<ShortageEntity> shortages**.

Za jej pomocą zastąp poniższy ciąg asercji:

```
Assert.assertEquals(2, shortages.size());
Assert.assertEquals(date.plusDays(2), shortages.get(0).getAtDay());
Assert.assertEquals(3400, shortages.get(0).getMissing());
Assert.assertEquals(date.plusDays(3), shortages.get(1).getAtDay());
Assert.assertEquals(7800, shortages.get(1).getMissing());
```

Równoważne asercje mogą wyglądać następująco:

```
assertThat(foundShortages)
        .foundExactly(2)
        .missingPartsAt(date.plusDays(1), 3400)
        .missingPartsAt(date.plusDays(2), 4400);
```

Metoda **assertThat** to statyczna metoda klasy **ShortagesAssert** zwracająca instancję tej klasy. Metody **missingPartsAt**, **noOtherShortages** to metody instancyjne tej samej klasy.
Wewnątrz klasy **ShortagesAssert** użyj biblioteki AssertJ (jest w classpath).

Ćwiczenie: Poprawa designu testów przy pomocy Fabryki (Object Mother)

Popraw design klasy testowej **tools.ShortageFinderTest**.

Zaimplementuj nową klasę **ExampleDemands**, której rolą będzie wygodne fabrykowanie kolekcji: **List<DemandEntity> demands** o określonych cechach.
Umieść tą klasę w źródłach testowych, nie jest potrzebna w kodzie produkcyjnym.

Przenieś do nowej klasy wszystkie metody prywatne z **tools.ShortageFinderTest** związane z List<DemandEntity> demands.

Zaproponuj metodę, która wygodnie tworzy sekwencję zapotrzebować od wskazanego dnia dla podanej serii liczb:

ExampleDemands.*demandSequence*(**date**, 17000, 17000);

Za jej pomocą zastąp poniższą konstrukcję:

demands(demand(2, 17000), demand(3, 17000))

Ćwiczenie: Poprawa designu testów przy pomocy Wzorca Builder

W klasie **ExampleDemands**, dodaj nową metodę:

public static List<DemandEntity> demandSequence(LocalDate startDate, DemandBuilder... demands)

która pozwala zwięźle i precyzyjne określić jakie zapotrzebowania są wymagane w kolejnych dniach, np:

```
ExampleDemands.demandSequence(date,
        demand(17000).tillEndOfDay(), demand(17000).adjustedTo(20000)
);
```

Zaimplementuj klasę DemandBuilder oraz statyczne metody demandSequence i demand.

Ćwiczenie: Poprawa designu testów przy pomocy metod given i when

Na poniższym przykładzie **ShortageFinderTest** podkreślono na czerwono „szum" związany z imperatywnym stylem programowania:
- deklaracje zmiennych lokalnych
- zapisy i odczyty zmiennych lokalnych.

Oraz uwypuklone jest wywołanie metody pod testami ShortageFinder.findShortages z pełną (długą) listą argumentów.

```
@Test
public void findShortages() {
   CurrentStock stock = warehouseStock(1000);
   List<ProductionEntity> productions = productPlan(forProductionLine(0)
                .plannedOutputs(date, 7, 6300, 6300, 6300, 6300, 6300, 6300, 6300)
                .plannedOutputs(date, 14, 6300, 6300, 6300, 6300, 6300, 6300, 6300)
         );
   List<DemandEntity> demands = demandSequence(date.plusDays(1), 17000, 17000);

   List<ShortageEntity> foundShortages = ShortageFinder.findShortages(
         date.plusDays(1), 7,
         stock,
         productions,
         demands
   );

   assertThat(foundShortages)
         .foundExactly(2)
         .missingPartsAt(date.plusDays(1), 3400)
         .missingPartsAt(date.plusDays(2), 4400);
}
```

Zamień zaznaczone zmienne lokalne jako pola klasy testowej oraz wprowadź metody prywatne given(…), whenShortagesArePredicted(), thenPredicted() by osiągnąć poniższy efekt:

```
@Test
public void findShortages() {
   given(
         warehouseStock(1000),
         productPlan(forProductionLine(0)
                .plannedOutputs(date, 7, 6300, 6300, 6300, 6300, 6300, 6300, 6300)
                .plannedOutputs(date, 14, 6300, 6300, 6300, 6300, 6300, 6300, 6300)
         ),
         customerDemands(date.plusDays(1), 17000, 17000)
   );

   whenShortagesArePredicted();

   thenPredicted()
         .foundExactly(2)
         .missingPartsAt(date.plusDays(1), 3400)
         .missingPartsAt(date.plusDays(2), 4400);
}
```