Ćwiczenie: Wady oprogramowania

Wskaż wady kodu, architektury, modularyzacji, technologii swojego obecnego projektu.

Ustaw je w kolejności od najbardziej poważnych, czyli mających najistotniejsze konsekwencje na dalszy rozwój produktu czy jego utrzymania.

Które z nich można naprawić za pomocą refactoringu?
W jaki sposób należało by naprawić najistotniejsze wady?

Ćwiczenie: Zaproponuj refactoring metody

Zaproponuj listę zmian, które chciałbyś wykonać w klasie **tools.ShortageFinder** w metodzie statycznej **findShortages**.

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

Ćwiczenie: Refactoring przy pomocy ekstrakcji metod prywatnych

W klasie **tools.ShortageFinder** w metodzie statycznej **findShortages** wprowadź w kod metody prywatne wykonujące operacje na dotychczasowej zmiennej lokalnej:

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

Zaproponuj takie sygnatury nowych metod by ich użycie w ShortageFinder było możliwie wygodne i zwięzłe.

Zadbaj by część logiki ShortageFinder związana z **konstrukcją** i **odczytem** danych z tej struktury była enkapsułowana wewnątrz nowych metod prywatnych.

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) {
```

UWAGA: to ćwiczenie NIE demonstruje dobrych praktyk refactoringowych.

Ćwiczenie: Refactoring przy pomocy Adaptera / Custom Collection

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

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

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

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());

String productRefNo = null;                                                                    Konstrukcja
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: Refactoring przy pomocy Adaptera / Custom Collection (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<>();

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.

Zadbaj by część logiki ShortageFinder związana z **konstrukcją** i **odczytem** danych z tej struktury była enkapsułowana wewnątrz nowej klasy adaptera.
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) {
    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;
    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
```

Odczyt

Odczyty

Debata Oksfordzka: Testy a refaktoryzkcja

Zadaniem debaty jest dyskusja nad tezą:
Testy automatyczne są niezbędne by móc refaktyryzować.

Debatują dwa zespoły obrońcy tezy (Propozycja) oraz jej przeciwnicy (Opozycja).
Wybór stanowisk czy jesteś Za (Propozycja) czy Przeciw (Opozycja) następuje losowo.
Musisz bronić stanowiska, które jest Ci przypisane niezależnie od Twoich poglądów.

Przebieg debaty:
- Omówienie zasad i tezy
- Losowanie stanowisk
- 10 min na przygotowanie argumentów w ramach zespołu
- Debatę rozpoczyna 1. mówca Propozycji.
- Następnie, 1. mówca Opozycji.
- Dalej naprzemiennie wypowiadają się mówcy poszczególnych stron.
- Debatę zawsze kończy ostatni mówca strony Opozycji.

Każdy z mówców ma 2 min na swoje wystąpienie. NIE przerywamy sobie wzajemnie. W wystąpieniu można i należy zadawać pytania stronie przeciwnej oraz argumentować w odpowiedzi na pytania przeciwnego zespołu.

Ćwiczenie: Refactoring przy pomocy Wzorca Budowniczy

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: Refactoring przy pomocy Wzorca Strategii (1/2)

W klasie **tools.ShortageFinder** w metodzie statycznej **findShortages** wprowadź w kod wzorzec strategii zastępujący ciąg if-else-if-else, zaznaczony poniżej czerwoną ramką.

By rozdzielić 3 typy logiki:

- logikę decyzji,
- implementację poszczególnych wariantów,
- efekty uboczne przetwarzania

refaktoryzuj według instrukcji:

1. Wyekstrahuj cały zaznaczony ciąg if-else-if-else, do **prywatnej metody**.
2. Stwórz **interfejs strategii** z jedną metodą o sygnaturze identycznej jak wyekstrahowana metoda.
3. Z nowo powstałej metody wyeliminuj efekty uboczne. W każdym z wariantów po prostu zwróć wymagane wartości lub rzuć wyjątek na wzór:
   - **return** level - demand.getLevel() + produced;
   - **throw new** UnsupportedOperationException();
4. Odseparuj implementację wariantów od wywołania (wyliczenia) wariantu:
   1. Dostosuj typ zwracany przez **metodę prywatną**, tak by zwracała typ **interfejsu strategii** oraz skoryguj miejsce wywołania **metody prywatnej**.
   2. Zamiast zwracać wynik wariantu wyliczenia (lub rzucać wyjątek) zwróć lambdę / instancję implementującą właściwy wariant jako **interfejs strategii**.
   3. Uprzątnij parametry metod, ewentualne błędy kompilacji oraz dostosuj nazwy metod.

```
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) {
```

Legenda podkreśleń:

logika decyzji

implementacja wariantu

efekt uboczny wariantu

Ćwiczenie: Refactoring przy pomocy Wzorca Strategii (2/2)

Po poprzednim ćwiczeniu w klasie **tools.ShortageFinder** w metodzie statycznej **findShortages** wybór wariantu kalkulacji oraz wywołanie odpowiedniej implementacji strategii wygląda następująco:

```
for (LocalDate day : dates) {
    DailyDemand demand = demandsPerDay.get(day);
    if (demand == null) {
        level += outputs.get(day);
        continue;
    }
    long produced = outputs.get(day);

    long levelOnDelivery = pick(demand).calculate(level, demand, produced);

    if (levelOnDelivery < 0) {
        ShortageEntity entity = new ShortageEntity();
        entity.setRefNo(outputs.getProductRefNo());
```

Legenda podkreśleń:

logika decyzji

implementacja wariantu

efekt uboczny wariantu

Wzorzec strategii pozwala na rozdzielenie **wyboru strategii** od **wywołania właściwego wariantu** i **efektów ubocznych**.

W powyższej implementacji nadal jednak wszystkie typy logiki wykonywane są w niemal tym samym momencie.

Zaproponuj gdzie wstrzyknąć instancję strategii oraz na którym etapie algorytmu dokonać wyboru implementacji.

Kieruj się ogólną zasadą programowania obiektowego:

Obiekt, który ma najwięcej informacji niezbędnych do wykonania operacji powinien być odpowiedzialny za implementację tej operacji.

Odpowiedz na dwa pytania:

Która klasa powinna mieć wstrzyknięta strategię?

Czy obiekt, który ma wstrzykniętą instancję strategii może enkapsułować ten fakt (nie udostępnia jej publicznie)?

Ćwiczenie: Refactoring 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:
Ctrl+Shift+A > „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: Refactoring przy pomocy Method Object (2. przypadek)

W klasie **tools.ShortageFinder** z metody statycznej **findShortages** wyekstrahuj wzorzec Method Object.
Zaznacz cały blok oznaczony czerwoną ramką oraz użyj funkcji IntelliJ-a:
Ctrl+Shift+A > „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.