Projekt „Moje ciuszki”

Kamil Mikołajczuk

Repozytorium z kodem: https://github.com/MikolajczukKamil/moj-sklep

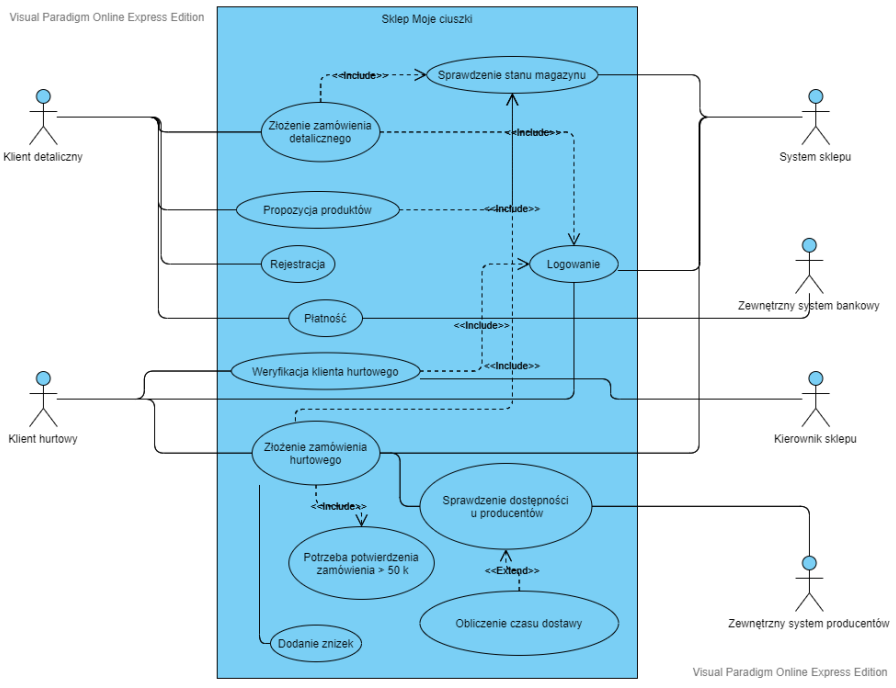
Wstęp

Mamy sklep / hurtownię o nazwie "Moje ciuszki" dalej po prostu sklep.
Sklep miał problem z tradycyjnym działaniem więc zatrudnił nas aby pomóc w informatyzacji i tym samy zwiększeniem skali działalności. Do tej pory informatyzacja sklepu polegała na używaniu Excela zamiast kartki i długopisu, teraz to nie wystarcza.

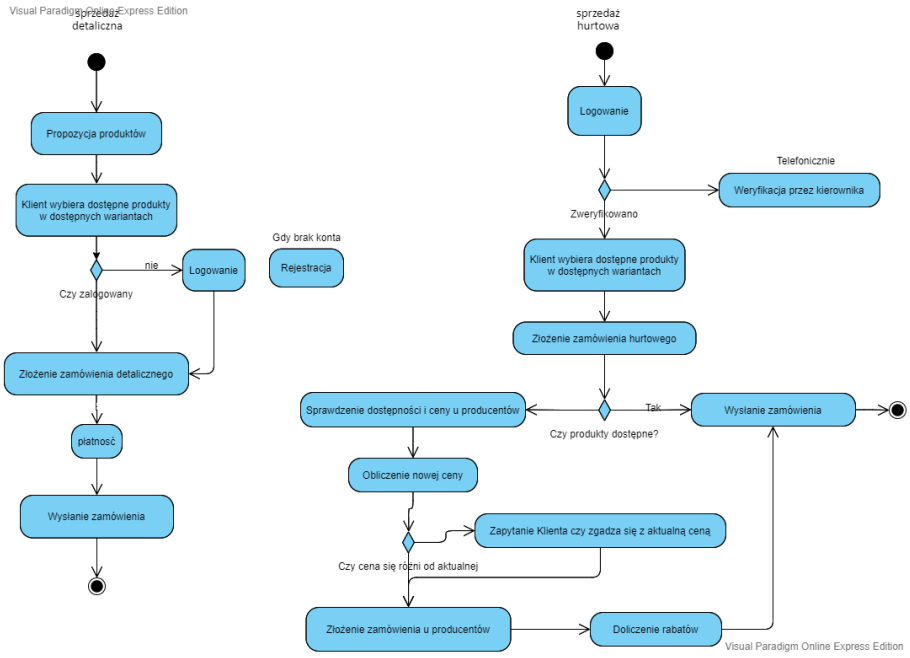
Informacje zdobyte na rozmowie z właścicielem i kierownikiem.

1. Sklep prowadzi sprzedaż detaliczną i hurtową artykułów odzieżowych.
2. Do przeglądania produktów logowanie nie jest wymagane, ale do dokonania zakupu już jest.
3. Sklep zamawia pewną ilość produktów na swój magazyn głównie na potrzeby klientów detalicznych, te produkty mają z góry znaną cenę i zamówienia są realizowane natychmiastowo.
4. Jeżeli ktoś zamawia większe ilości produktów (zmówienie hurtowe) lub skończyły się zapasy sklepowe sklep dokonuje zamówienia u producenta, cena wtedy weryfikowana jest z aktualną ceną producenta i trendami na rynku.
5. Jeżeli cena po weryfikacji mieści się w akceptowalnym zakresie uznajemy cenę poprzednią, czyli tą podaną na stronie, w przeciwnym przypadku klient jest informowany o zmianie ceny i musi się zgodzić na nową propozycję transakcji.
6. System ma szacować prawdopodobny czas dostawy uwzględniając dostawę od producenta.
7. System udziela rabatów wg ustalonych zasad, np. powyżej 100 sztuk -5%, powyżej 1000 -10% itd. tabele te tworzy kierownik w Excelu i tego nie zmieni, bo to według niego najlepszy program na świecie.
8. Klienci hurtowi terminowo rozliczający się ze sklepem mogą liczyć na dodatkowe zniżki tu dodatkowe -5% tu również dane pobieramy z Excela.
9. Niektórzy klienci hurtowi (Szalikpol, Świat czapek, Spodniex) dostają zniżki, ale obniżone o 50% gdyż podpadli szefowi, oni mają o tym nie wiedzieć to tajemnica firmy, z ich strony ma być widoczny rabat np. 2,5%.
10. Zamówienia o wartości powyżej 50 000 zł muszą być zaakceptowane przez kierownika.

Diagramy



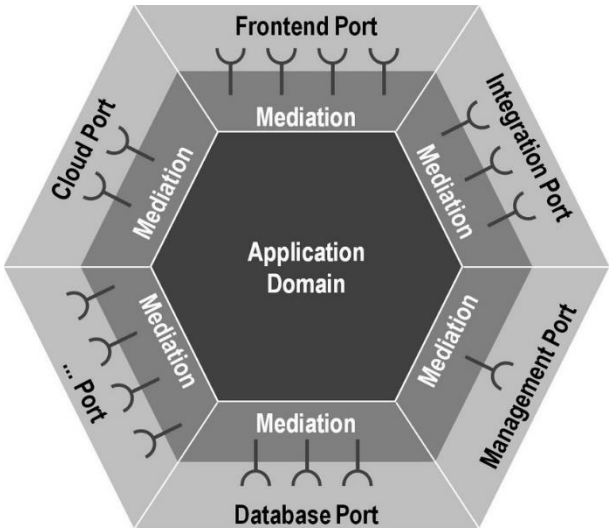
Rysunek 1 Diagram przypadków użycia UML



Rysunek 2 Diagram czynności UML

Wybrana architektura

Architektura portów i adapterów inaczej zwana heksagonalną.



Rysunek 3 Symboliczne przedstawienie architektury portów i adapterów

Alternatywna nazwa architektura heksagonalna wzięła się z prostego faktu, że sześciokąt dobrze prezentuje się na prezentacjach sama architektura w żaden sposób nie wymusza ani nie faworyzuje 6 podsystemów / modułów.

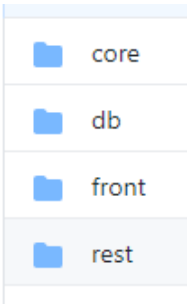
Główna nazwa architektura portów i adapterów znacznie lepiej oddaje tym czy jest ta architektura.

Architektura ta przeznaczona jest dla projektów o dużej złożoności procesu biznesowego. Główna idea to podzielenie aplikacji na moduły, w których centrum (core domain) znajduje się najważniejszy kod odpowiedzialny za proces biznesowy niezanieczyszczony frameworkami do obsługi bazy danych czy interfejsu użytkownika. Umożliwia oddzielnie złożoności domenowej od złożoności technicznej czy przypadkowej wynikającej z wybranych technologii. W prostych implementacjach sprowadza się do klasycznej architektury warstwowej.

W skład systemu wchodzą podsystemy w omawiany przykładzie:

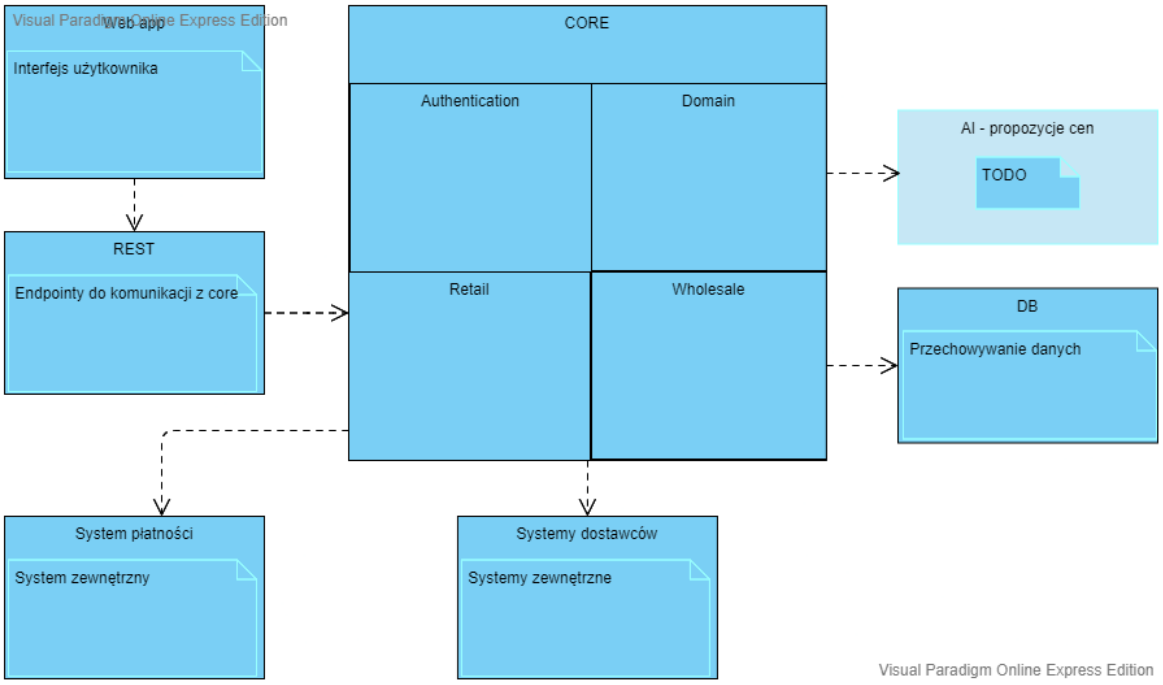
- *core* podsystem odpowiedzialny za proces biznesowy
- *db* podsystem odpowiedzialny za przechowywanie danych
- *front* podsystem z interfejsem użytkownika
- *rest* podsystem wystawiające restowe API dla podsystemu front

Podsystemy w praktyce sprowadzają się do oddzielnych katalogów



Rysunek 4 Podział na podsystemy

Podział na moduły



Rysunek 5 Podział na moduły

Sam CORE został podzielony na 4 moduły / konteksty autentyfikacja użytkownika, kontekst domenowy czyli Produkt jaki sprzedajemy, dostępne warianty jak i szeroko używane value-object y np. klasa Money umożliwia operacje na różnych walutach więc w przyszłości jest łatwa możliwość rozbudowy w tym kierunku.

```
4   export class Money {
5     private readonly value: number
6
7     constructor(value: number, private currency = 'PLN') {
8       this.value = Math.round(value * 100)
9     }
10
11  >   /**...
14    add(other: Money) {
15      if (this.currency != other.currency) throw new Error(`Waluty nie są zgodne ${this.currency} i ${other.currency}`)
16
17      return new Money(this.toNumber() + other.toNumber(), this.currency)
18    }
19
20  >   /**...
23    times(many: number) {
24      return new Money(this.toNumber() * many, this.currency)
25    }
26
27    toNumber() {
28      return this.value / 100
29    }
30
31  >   /**...
34    toString(locate = 'pl-PL') {
35      return Money.ToMoneyString(this.value, false, locate)
36    }
37
```

Rysunek 6 Object value - Money

Autoryzacja

Autoryzacja jest funkcjonalnością przewijającą się w prawie wszystkich modułach, interfejs użytkownika odpowiada za możliwość wpisania danych, wstępnie je waliduje po czym przygotowuje zapytanie do modułu REST który kontaktuje się z modułem CORE który pobiera dane z modułu DB (dane użytkownika o podanym loginie) jeżeli taki użytkownik nie jest znaleziony zwraca błąd autoryzacji który moduł REST interpretuje i zwraca komunika tu „Login lub hasło nieprawidłowe” który można wyświetlić jako błąd, gdy zaś użytkownik został znaleziony moduł CORE (w zasadzie core/authorization) dokonuje sprawdzenie poprawności hasła jeżeli hasła nie są zgodne zwraca błąd podobnie jak przy nie znalezieniu użytkownika zaś gdy jest wszystko w porządku zwraca zalogowanego użytkownika

oraz generuje jego indywidualny jsonweb token dzięki któremu będzie można później rozpoznać sesję logowania. Moduł REST jest odpowiedzialny za przekazanie w odpowiedni sposób tokena tutaj umieszczenie go w ciasteczku.

```
public signIn = async (req: Request, res: Response, next: NextFunction) => {
  const userData: SignInUserDto = req.body

  try {
    const { cookie, user } = await this.authService.signIn(userData)

    res.setHeader("Set-Cookie", [cookie])
    res
      .status(200)
      .json({ data: user.dto(), message: "sign-in" } as IAuthSignIn)
  } catch (error) {
    next(error)
  }
}
```

Rysunek 7 Kontroler REST/Auth

```
public async signIn(userData: ILoginUserDto): Promise<ILoginData> {
  const user = await this.userRepository.withLoginOptional(userData.login)

  if (!user) throw new AuthExeception(401, `Login lub hasło nieprawidłowe`)

  const isPasswordMatching = await bcrypt.compare(userData.password, user.password)

  if (!isPasswordMatching) throw new AuthExeception(401, 'Login lub hasło nieprawidłowe')

  const tokenData = this.createToken(user.id, userData.rememberMe ? 24 * 60 : 60)
  const cookie = this.createCookie(tokenData)

  return { cookie, user }
}
```

Rysunek 8 Serwis CORE/Authentication

Wygląda to jak procedura rozbita między warstwy jak w klasycznej architekturze warstwowej i tu tak dokładnie jest. Jedyną różnicą jest to że w klasycznej architekturze warstwowej często spotyka się wymieszanie warstwy obsługującej protokół komunikacji tu rest z warstwą odpowiedzialną za samą logikę logowania, w tym podejściu warstwa rest nie wie jak weryfikowany jest użytkownik a warstwa core nie wie skąd pochodzą dane o użytkowniku.

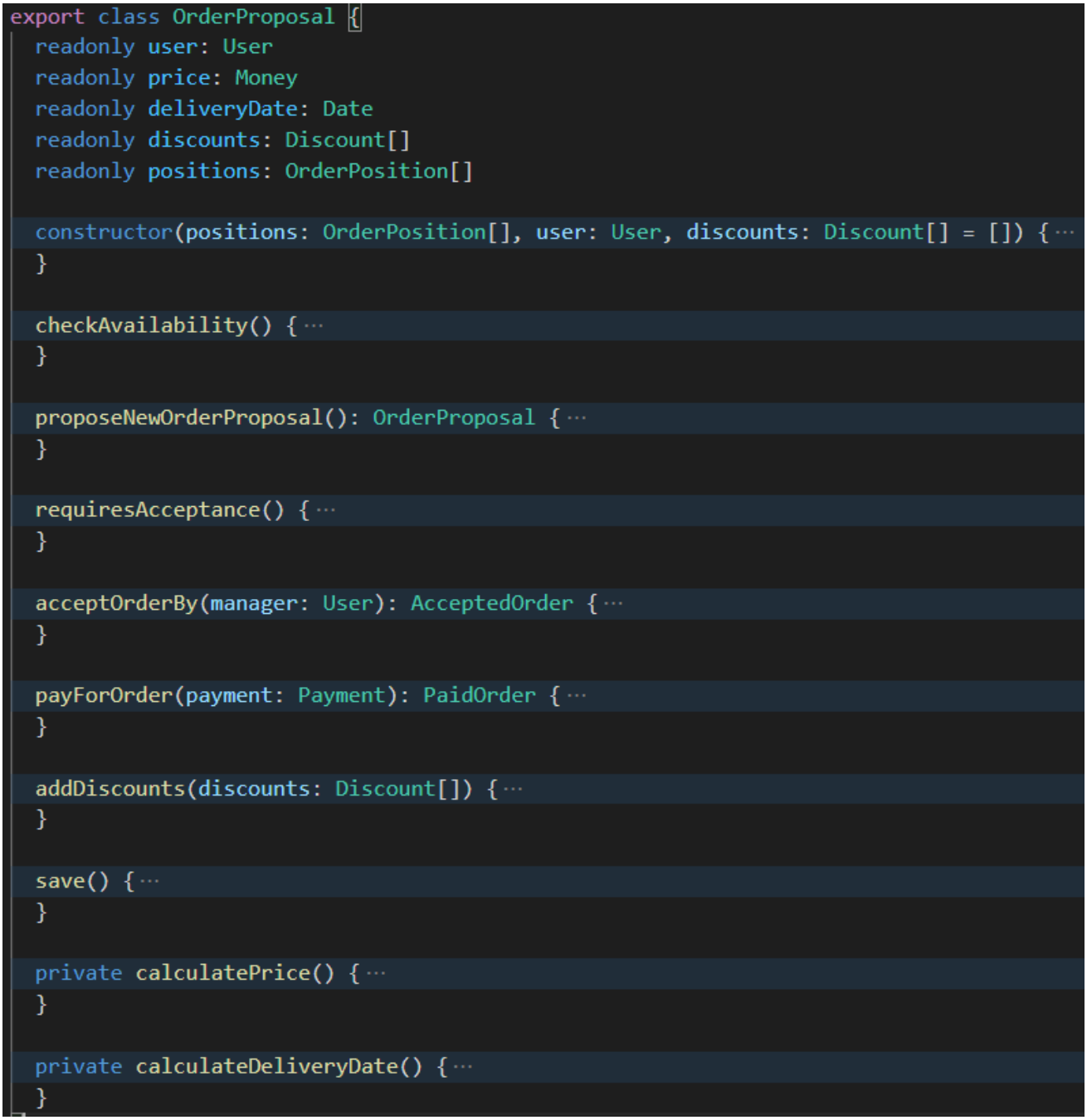
Obsługa zamówienia – komunikacja między modułami

Warstwa CORE wystawia serię obiektów domenowych czyli spełniających wymagania postawione przez modelem, następnie kontroler dla konkretnego przypadku użycia używa ich wykonując swoje zadanie. Jeżeli zmienimy zasady np. przyznawania rabatów zmiana nastąpi tylko w podmodule rabatów nie zmieni to reszty aplikacji, przez zastosowanie wyraźnych granic kompetencji modułów zmiany mają efekt możliwie lokalny.

Każdy rodzaj zamówienia ma swoją klasę (Propozycja Zamówienia, Zamówienie zapłacone, Zamówienie zaakceptowane) ma to na celu zwiększenie utrzymywalności kodu, od razu widać czym skutkuje wywołanie danej metody, dla zamówienia już opłaconego nie ma sensu ponowne pobranie płatności takie rozwiązanie wychwytuje tego typu błędu już w czasie kompilacji.

```
public validateOrder = (
  req: RequestWithUser,
  res: Response,
  next: NextFunction
) => {
  const user = req.user

  const order = new OrderProposal(this.loadOrderPosition(), user)
  order.save()

  if (order.checkAvailability()) {
    if (order.requiresAcceptance()) {
      const toAccept = order.acceptOrderBy(UserService.Menager)
      toAccept.save()

      res.status(200).json({ data: toAccept.dto(), message: "order-requires-acceptance-by-menager" })
    }
    else {
      const orderWithDiscounts = order.addDiscounts(this.discountsService.discountsFor(user))
      orderWithDiscounts.save()

      res.status(200).json({ data: orderWithDiscounts.dto(), message: "order-accepted" })
    }
  }
  else {
    const proposition = order.proposeNewOrderProposal()
    proposition.save()

    res.status(200).json({ data: proposition.dto(), message: "order-proposition" })
  }
}
```

Rysunek 9 Użycie Zamówienia w module REST

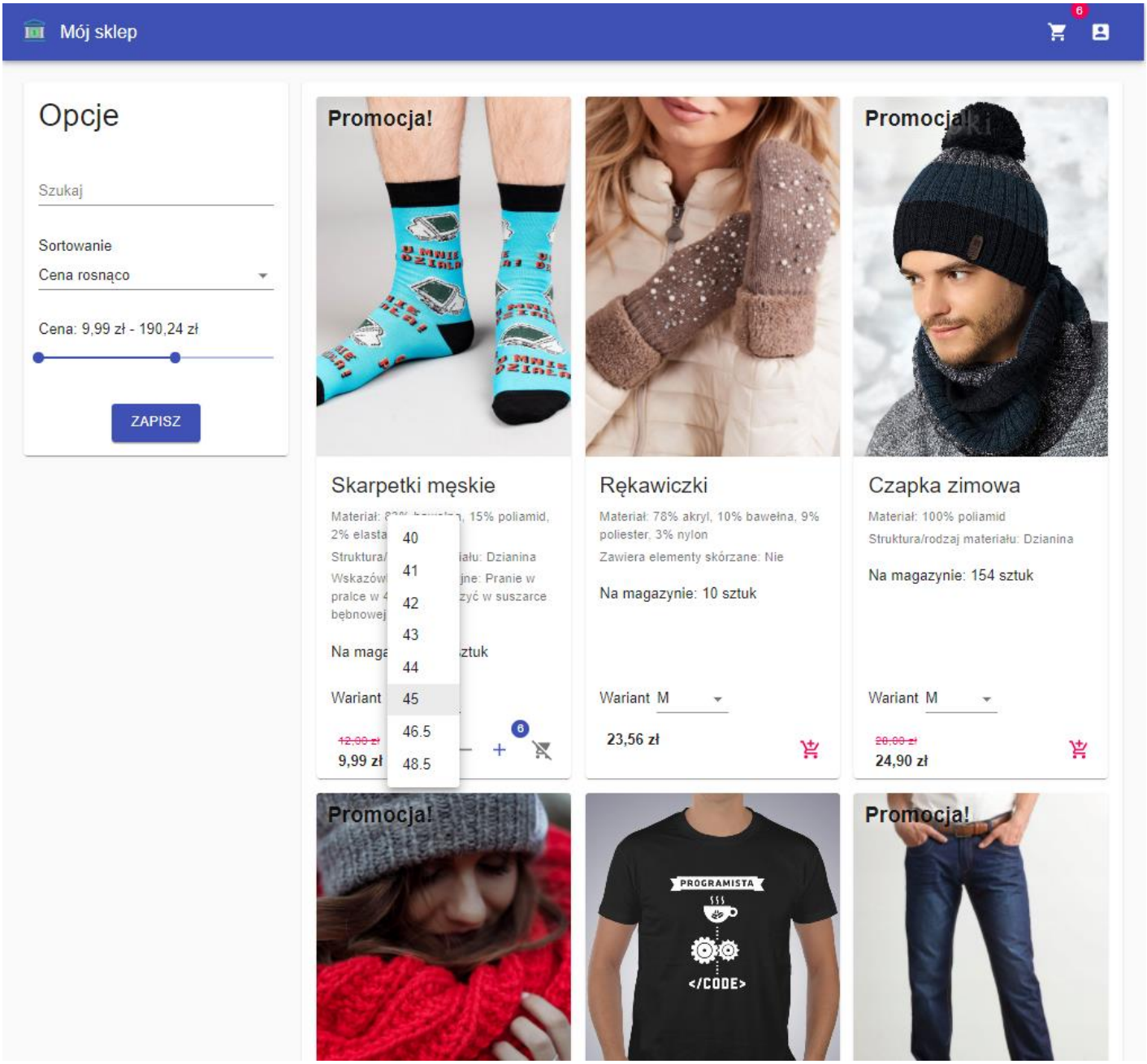


```
export class OrderProposal {
  readonly user: User
  readonly price: Money
  readonly deliveryDate: Date
  readonly discounts: Discount[]
  readonly positions: OrderPosition[]

  constructor(positions: OrderPosition[], user: User, discounts: Discount[] = []) {···
  }

  checkAvailability() {···
  }

  proposeNewOrderProposal(): OrderProposal {···
  }

  requiresAcceptance() {···
  }

  acceptOrderBy(manager: User): AcceptedOrder {···
  }

  payForOrder(payment: Payment): PaidOrder {···
  }

  addDiscounts(discounts: Discount[]) {···
  }

  save() {···
  }

  private calculatePrice() {···
  }

  private calculateDeliveryDate() {···
  }
```

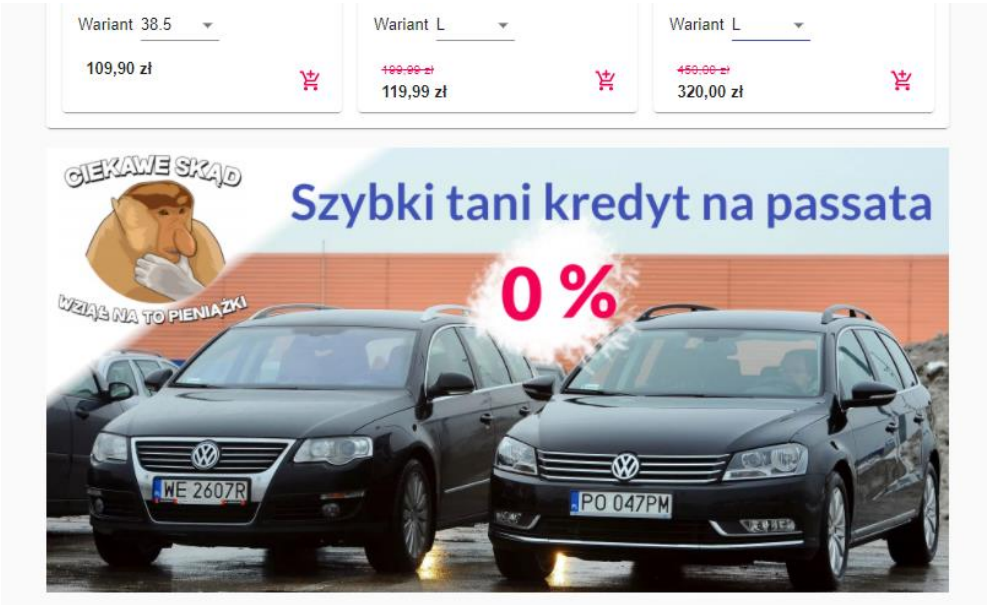
Rysunek 10 CORE/Zamówienie

front - Interfejs użytkownika

Popularna aplikacja typu SPA prosta nowoczesna jak i dobrze znana stylistyka material design. Zapewnia funkcjonalności znane z popularnych sklepów online takie jak wyszukiwanie produktu, sortowanie po wybranych wartościach filtrowanie po cenie itp. Dla konkretnego produktu mamy możliwość wyboru wariantu oraz ilość jaką chcemy kupić, możemy kupić ten sam produkt w kilku wariantach np. 2 koszuli w rozmiarze M oraz 3 i rozmiarze L.



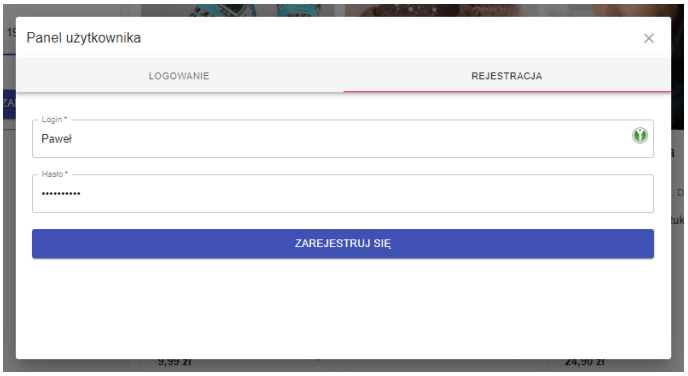
Rysunek 11 Strona główna



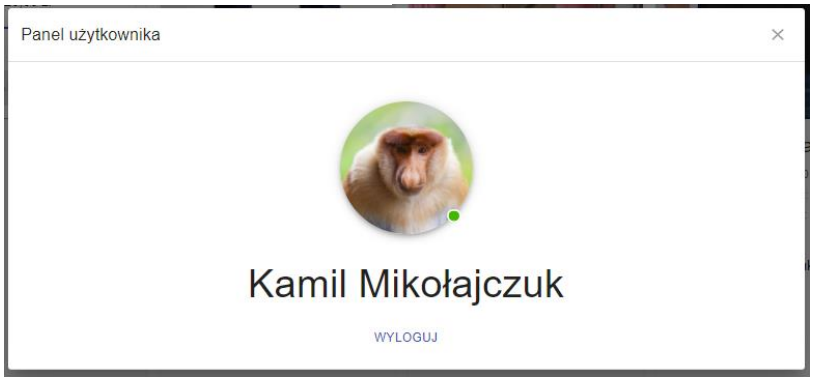
Rysunek 12 Dodatkowe źródło przychodu - reklamy

Wymaganie funkcjonalne – logowanie i rejestracja.

Rysunek 13 Panel użytkownika - Logowanie



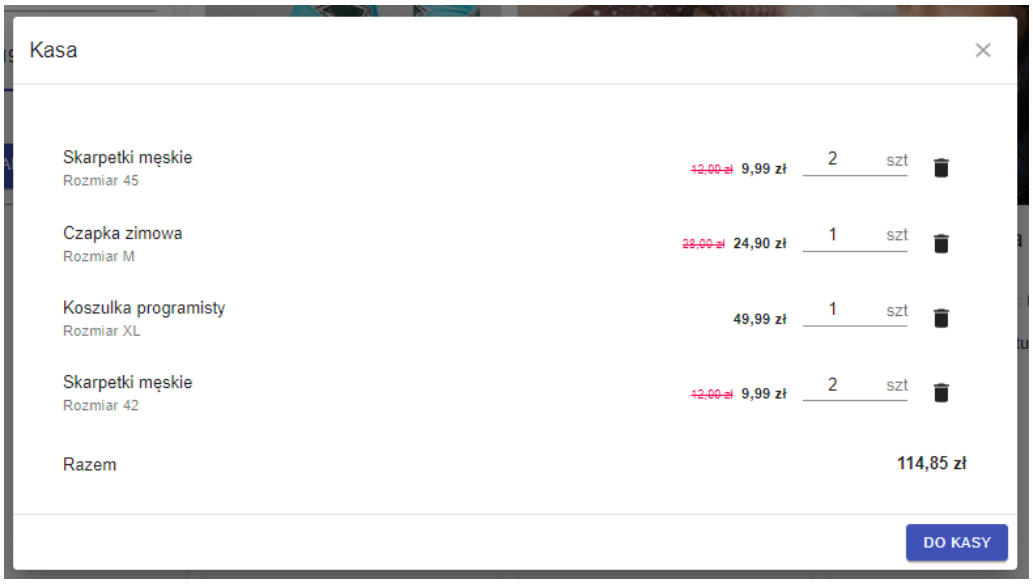
Rysunek 14 Panel użytkownika - Rejestracja



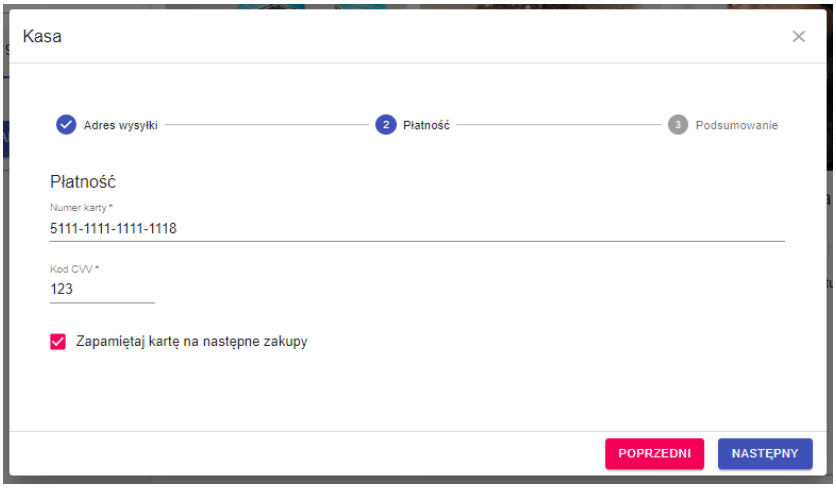
Rysunek 15 Panel użytkownika - Zalogowany

Koszyk i kasa

Warto zauważyć że w kontekście zamówienia Produkt zmienia swoje znaczenie w porównaniu do Produktu w kontekście przeglądania oferty czy tym ogólnym jeżeli mówimy o Produkcie w tej domenie, różnica jest oczywiście w wariantach tu Produkt ma 1 konkretny wariant a Produkt z innym wariantem to inny Produkt. Wnioskiem z tej obserwacji jest że zamówienia to inny moduł posługujący się inną klasą Produkt.



Rysunek 16 Koszyk



Rysunek 12 Kasa

Testowanie

Warto zauważyć że ta architektura sama z siebie promuje testowanie nie trzeba robić prawie nic ponad to co już mamy na potrzeby aplikacji, bierzemy klasę Propozycja Zamówienia wykonujemy np. akceptację czy płatność i sprawdzamy czy efekt jest zgodny z oczekiwaniami czy to że płatność na inną kwotę powinna zostać odrzucona. Nie dotykamy wtedy kodu technicznego, co mogło by być trudne w przypadku prostym podejściu 3 warstwowym. W skrócie wyraźny podział na logikę biznesową oraz logikę techniczną bardzo ułatwia testowanie.

```
describe("OrderProposal", () => {
  test("Order worth above 50 000 PLN should require approval", () => {
    const order = giveOrderWithProductWorth(55000)
    const requiresAcceptance = order.requiresAcceptance()
    expect(requiresAcceptance).toBeTruthy()
  })

  test("Order worth below 50 000 PLN should not require approval", () => {
    const order = giveOrderWithProductWorth(45000)
    const requiresAcceptance = order.requiresAcceptance()
    expect(requiresAcceptance).not.toBeTruthy()
  })

  test("Payment worth equal to the value of the order should accept the order", () => {
    const order = giveOrderWithProductWorth(100)
    const payment = givePaymentWorth(100)
    const paidOrder = order.payForOrder(payment)
    expect(paidOrder).toBeInstanceOf(PaidOrder)
  })

  test("Payment worth not equal to the value of the order should not accept the order", () => {
    const order = giveOrderWithProductWorth(100)
    const payment = givePaymentWorth(50)
    expect(() => order.payForOrder(payment)).toThrow(OrderException)
  })
```

Rysunek 13 Przykład testów jednostkowych

Podsumowanie

Architektura portów i adapterów jest okazała się dobrym wyborem, dla prostych zastosowań takich jak samo zamówienie detaliczne jest nieco naddatkowa wtedy spokojnie wystarczy prosta architektura trój warstwowa, dla rozbudowanych projektów za to daje duże możliwości zachowując czystość i klarowność kodu.