Projekt zaliczeniowy

Programowanie obiektowe
Rok akademicki 2025/2026

Autorzy:
Bartosz Opioła
Emanuel Sagan
Marcel Radwański
Paweł Szulik

System Zarządzania Portfelem Inwestycyjnym

Projekt to aplikacja desktopowa stworzona w technologii C# .NET 8 z wykorzystaniem frameworka WPF oraz wzorca architektonicznego MVVM. Celem systemu jest symulacja i zarządzanie portfelem inwestycyjnym, umożliwiająca użytkownikowi śledzenie wartości różnorodnych aktywów finansowych, takich jak: akcje, obligacje, kryptowaluty, surowce oraz nieruchomości.

Podział ról

Bartosz Opioła: Testy jednostkowe, dokumentacja
Emanuel Sagan: Interfejs Użytkownika
Marcel Radwański: Pełna implementacja hierarchii, integracja systemu
Paweł Szulik: Konfiguracja bazy danych

1 MODEL MATEMATYCZNY SYMULACJI RYNKOWEJ – GEOMETRYCZNY RUCH BROWNA

Do odwzorowania zachowań rynków finansowych aplikacja wykorzystuje algorytm oparty na modelu Geometrycznego Ruchu Browna. Jest to proces losowy stosowany w ekonometrii do modelowania cen aktywów, który zakłada, że logarytmy zwrotów z akcji mają rozkład normalny, a ceny zmieniają się w czasie w sposób ciągły i losowy.

Implementacja modelu znajduje się w statycznej klasie MarketSimulator, w metodzie GenerateNewPrice. Wybór tego modelu podyktowany jest koniecznością wyeliminowania możliwości wystąpienia ujemnych cen aktywów, co jest kluczową cechą odróżniającą Geometryczny Ruch Browna od arytmetycznego ruchu Browna.

Proces wyznaczania nowej ceny za pomocą tej metody przebiega w następujący sposób:

1. **Generowanie zmiennych losowych**
   W pierwszej fazie algorytm pobiera dwie liczby losowe o rozkładzie jednostajnym (u_1,u_2). W kodzie wykorzystano właściwość Random.Shared, aby zapewnić niezależność zdarzeń rynkowych. Tradycyjne tworzenie generatora wewnątrz funkcji niesie ryzyko zainicjowania go tą samą wartością zegara systemowego przy szybkich obliczeniach, co spowodowałoby, że różne spółki zachowywałyby się identycznie. Użycie Random.Shared gwarantuje unikalny przebieg losowy dla każdego aktywa. Następnie, aby uzyskać zmienną o standardowym rozkładzie normalnym (N(0,1)), konieczną do symulacji ruchu Browna, zastosowano transformację Boxa-Mullera.

2. **Obliczanie ceny końcowej**
   Wygenerowana losowość o rozkładzie normalnym jest wykorzystana do obliczania nowej ceny za pomocą dwóch czynników:

a. Losowego zaburzenia
Wprowadza ona nieprzewidywalność rynkową, jest iloczynem zmienności aktywa (volatility) i wylosowanej wcześniej liczby z rozkładu normalnego: double shock = volatility * randStdNormal. Wartość ta może być dodatnia lub ujemna, symulując nagłe wzrosty lub spadki kursów.

b. Agregacji wykładniczej
Ostateczna cena jest obliczana poprzez pomnożenie aktualnej ceny przez funkcję wykładniczą sumy dryfu i zaburzenia: return currentPrice * Math.Exp(drift + shock);. Zastosowanie funkcji Math.Exp (eksponenty) wynika z założenia, że logarytmy zwrotów mają rozkład normalny. Gwarantuje to, że niezależnie od wielkości spadków, wygenerowana cena nigdy nie będzie ujemna.

Wzór zastosowany w aplikacji[1]:

$$S_t = S_{t-1} \times e^{\left(\mu - \frac{1}{2}\sigma^2\right) + \sigma\epsilon}$$

S – cena
μ – meanReturn
σ – volatility
ϵ – wygenerowana zmienna losowa

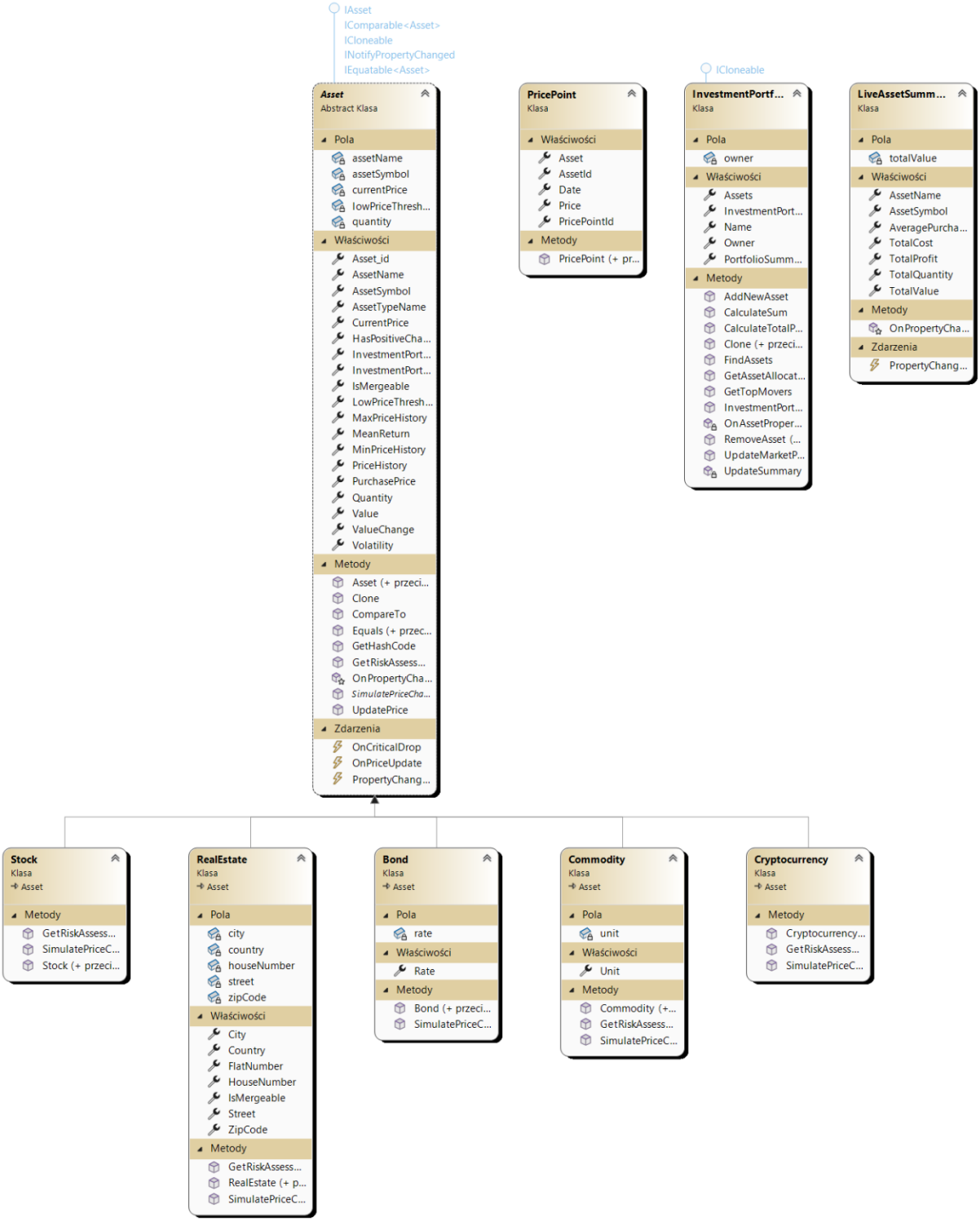
Źródło: https://en.wikipedia.org/wiki/Geometric_Brownian_motion

Film, na podstawie którego powstała implementacja: https://youtu.be/ppXN8C-efeA?si=aCPz6ZoePEafoKnM

[1] W tym wzorze pominięto Δt, ponieważ w symulacji jest to zawsze 1 dzień.

2 SZCZEGÓŁOWY OPIS WARSTWY CORE

2.1 DIAGRAM KLAS



2.2 HIERARCHIA KLAS

Podstawą modelu jest klasa abstrakcyjna Asset, definiująca wspólne cechy instrumentów finansowych, oraz klasy pochodne reprezentujące konkretne typy aktywów. Klasa abstrakcyjna Asset.

2.2.1 Klasa abstrakcyjna Asset

Opis: Klasa bazowa dla wszystkich składników portfela.

Implementuje interfejsy: IAsset, IComparable<Asset>, ICloneable, INotifyPropertyChanged, IEquatable<Asset>.

Rola w projekcie: Zapewnia polimorfizm, umożliwiając traktowanie różnych instrumentów (akcje, obligacje, nieruchomości) w jednolity sposób na listach i w obliczeniach. Przechowuje wspólną logikę walidacji, historii cen oraz powiadomień dla UI.

2.2.1.1 Właściwości

- Asset_id
    - Dostęp: public get, public set
    - Opis: Unikalny identyfikator obiektu (GUID), oznaczony atrybutem [Key].
    - Uzasadnienie: Choć logicznie ID powinno być niezmienne, publiczny setter jest wymagany przez mechanizmy Entity Framework Core oraz XmlSerializer do poprawnego mapowania klucza głównego i deserializacji obiektów z pliku/bazy.
- AssetName
    - Dostęp: public get, public set
    - Opis: Przechowuje nazwę aktywa.
    - Uzasadnienie: public set umożliwia edycję, ale dostęp jest kontrolowany przez logikę walidacyjną (rzuca AssetNameException dla pustych wartości).
- AssetSymbol
    - Dostęp: public get, public set
    - Opis: Symbol giełdowy lub skrót.
    - Uzasadnienie: Publiczny setter wymusza spójność danych poprzez konwersję na wielkie litery (ToUpper()) oraz walidację (rzuca AssetSymbolException).
- Quantity
    - Dostęp: public get, public set
    - Opis: Ilość posiadanych jednostek danego aktywa.
    - Uzasadnienie: Setter publiczny umożliwia modyfikację portfela. Zawiera walidację (ilość > 0, rzuca InvalidQuantityException) oraz wyzwala powiadomienie OnPropertyChanged dla właściwości Value.
- PurchasePrice
    - Dostęp: public get, public set
    - Opis: Cena zakupu jednej jednostki aktywa.
    - Uzasadnienie: Dostęp musi być publiczny, aby umożliwić pełną serializację XML i zapis do bazy danych.
- CurrentPrice
    - Dostęp: public get, public set
    - Opis: Aktualna cena rynkowa aktywa.
    - Uzasadnienie: Setter zawiera kluczową logikę biznesową: walidację (cena >= 0, rzuca InvalidPriceException), detekcję zmian, wyzwalanie zdarzeń OnPriceUpdate i OnCriticalDrop oraz odświeżanie UI.

- Value
    - Dostęp: public get
    - Opis: Całkowita wartość pozycji (Quantity * CurrentPrice).
    - Uzasadnienie: Właściwość wyliczalna, zawsze spójna z aktualnym stanem obiektu.
- PriceHistory
    - Dostęp: public get, public set
    - Opis: Kolekcja (ObservableCollection<PricePoint>) przechowująca historię cen.
    - Uzasadnienie: Setter musi być publiczny dla deserializacji XML. Kolekcja jest Observable, aby UI automatycznie reagowało na nowe punkty danych.
- Volatility
    - Dostęp: public get, public set
    - Opis: Współczynnik zmienności ceny.
    - Uzasadnienie: Umożliwia konfigurację modelu symulacji (Geometryczny Ruch Browna) dla konkretnego aktywa.
- MeanReturn
    - Dostęp: public get, public set
    - Opis: Średnia oczekiwana stopa zwrotu.
    - Uzasadnienie: Parametr modelu symulacyjnego, określający długoterminowy trend.
- LowPriceThreshold
    - Dostęp: public get, public set
    - Opis: Próg cenowy dla alertów krytycznych.
    - Uzasadnienie: Właściwość konfiguracyjna, zmiana wyzwala OnPropertyChanged.
- IsMergeable
    - Dostęp: public virtual get
    - Opis: Flaga określająca, czy aktywa można łączyć (grupować).
    - Uzasadnienie: Domyślnie true. Tylko do odczytu, gdyż wynika z typu klasy.
- InvestmentPortfolioId
    - Dostęp: public get, public set
    - Opis: Klucz obcy wskazujący na portfel właściciela.
    - Uzasadnienie: Wymagane przez Entity Framework do obsługi relacji jeden-do-wielu.
- AssetTypeName
    - Dostęp: public get
    - Opis: Zwraca nazwę typu klasy, oznaczona [NotMapped].
    - Uzasadnienie: Właściwość pomocnicza dla DataGrid w WPF, ułatwiająca wyświetlanie typu aktywa bez konwerterów.
- Właściwości pomocnicze (ValueChange, HasPositiveChange, MinPriceHistory, MaxPriceHistory)
    - Dostęp: public get
    - Opis: Właściwości wyliczalne, oznaczone [NotMapped]/[XmlIgnore].
    - Uzasadnienie: Służą wyłącznie warstwie prezentacji (kolorowanie wyników, tooltipy z historią) i nie są zapisywane w bazie.

2.2.1.2 Metody

- Asset()

Konstruktor bezparametrowy wymagany przez serializatory.

- Asset(string name, string symbol, double quantity, double purchasePrice, double volatility)
  Konstruktor protected dla klas pochodnych, inicjalizujący stan początkowy.
- UpdatePrice(double newPrice)
  Publiczna metoda do ręcznej aktualizacji ceny z zewnątrz (np. przez symulator).
- SimulatePriceChange(DateTime simulationDate)
  Metoda abstrakcyjna wymuszająca implementację logiki zmiany ceny w klasach pochodnych.
- Clone()
  Tworzy głęboką kopię aktywa, resetując ID i powiązanie z portfelem.
- GetRiskAssessment()
  Metoda wirtualna zwracająca ocenę ryzyka (domyślnie Medium).

2.2.2 Klasa Stock

Opis: Reprezentuje akcję spółki giełdowej. Dziedziczy po Asset.

Rola w projekcie: Podstawowy typ aktywa o wysokiej zmienności.

2.2.2.1 Metody

- Konstruktor: Ustawia domyślną zmienność (Volatility) na 2% (0.02).
- SimulatePriceChange: Wykorzystuje MarketSimulator (Ruch Browna) do generowania nowej ceny.
- GetRiskAssessment: Nadpisuje metodę bazową, zwracając RiskEnum.High.

2.2.3 Klasa Bond

Opis: Reprezentuje obligacje o stałym dochodzie. Dziedziczy po Asset.

Rola w projekcie: Aktywo bezpieczne o przewidywalnym wzroście wartości.

2.2.3.1 Właściwości

- Rate
  - Dostęp: public get, public set
  - Opis: Roczna stopa procentowa.
  - Uzasadnienie: Setter zawiera walidację (nie może być ujemna, rzuca BondRateException).

2.2.3.2 Metody

- Konstruktor: Ustawia zmienność na 0%.
- SimulatePriceChange: Implementuje model deterministyczny – cena rośnie liniowo o dzienne odsetki (Rate / 365).

2.2.4 Klasa Cryptocurrency

Opis: Reprezentuje cyfrowe aktywo o bardzo dużej zmienności. Dziedziczy po Asset.

Rola w projekcie: Element portfela generujący ekstremalne ryzyko i potencjalnie wysokie zyski.

2.2.4.1 Metody

- Konstruktor: Ustawia wysoką zmienność na 8% (0.08).
- GetRiskAssessment: Zwraca RiskEnum.ExtremelyHigh.
- SimulatePriceChange: Wykorzystuje standardowy symulator stochastyczny.

2.2.5 Klasa RealEstate

Opis: Reprezentuje fizyczną nieruchomość z danymi adresowymi. Dziedziczy po Asset.

Rola w projekcie: Aktywo o niskiej płynności i specyficznym zachowaniu symulacyjnym.

2.2.5.1 Właściwości

- IsMergeable: Nadpisana, zwraca false. Każda nieruchomość jest unikalna i nie może być sumowana z innymi.
- Street, HouseNumber, City, ZipCode, Country
    - Dostęp: public get, public set
    - Opis: Dane adresowe.
    - Uzasadnienie: Zawierają walidację (rzucają InvalidAddressException lub InvalidZipCodeException).
- FlatNumber: Opcjonalny numer lokalu.

2.2.5.2 Metody

- Konstruktor: Ustawia Symbol na "PROP", ilość na 1, niską zmienność (0.005) oraz niski MeanReturn.
- SimulatePriceChange: Aktualizuje cenę tylko pierwszego dnia miesiąca (symulacja niskiej płynności rynku).
- GetRiskAssessment: Zwraca RiskEnum.Low.

2.2.6 Klasa Commodity

Opis: Reprezentuje towary takie jak złoto czy ropa. Dziedziczy po Asset.

Rola w projekcie: Wprowadza pojęcie jednostek miary (Unit).

2.2.6.1 Właściwości

- Unit
    - Dostęp: public get, public set
    - Opis: Jednostka miary (enum UnitEnum: Ounce, Barrel, Ton itp.).
    - Uzasadnienie: Setter sprawdza czy wartość należy do enuma (rzuca InvalidUnitException).

2.2.6.2 Metody

- Konstruktor: Wymusza podanie jednostki miary, ustawia zmienność na 1.5%.
- GetRiskAssessment: Zwraca RiskEnum.Medium.

2.3 KLASA PRICEPOINT

Opis: Klasa pomocnicza reprezentująca pojedynczy punkt na wykresie historii.

Rola w projekcie: Encja bazodanowa przechowująca historię notowań dla relacji jeden-do-wielu z Asset.

2.3.1.1 Właściwości

- PricePointId: Klucz główny (PK).
- Date: Data notowania.
- Price: Cena w danym dniu.
- AssetId: Klucz obcy do aktywa.

2.4 KLASA MARKETEVENT

Opis: Klasa modelująca losowe zdarzenia rynkowe, które wpływają na parametry symulacji.

2.4.1 Właściwości

- Title, Description (String): Opis zdarzenia wyświetlany użytkownikowi.
- DurationTicks (int): Czas trwania zdarzenia w cyklach symulacji.
- TargetPredicate (Func<Asset, bool>): Delegat określający, na które aktywa wpływa zdarzenie.
- VolatilityMultiplier, MeanReturnModifier (double): Modyfikatory wpływające na matematykę symulacji (zmienność i trend).

Wszystkie właściwości są publiczne, aby mogły wpływać na inne klasy.

2.5 KLASA LIVEASSETSUMMARY

Opis: Klasa służy jako agregat danych. Jej głównym celem jest przechowywanie podsumowania dla grupy aktywów tego samego typu (np. wszystkich posiadanych akcji Apple kupionych w różnych transakcjach). Dzięki niej możliwe jest otrzymanie jednej, zbiorczej pozycji z uśrednioną ceną zakupu i sumarycznym zyskiem.

2.5.1 Właściwości

- AssetSymbol oraz AssetName
  - **Typ:** string
  - **Dostęp:** public get, public set
  - **Opis:** Przechowują identyfikator (symbol) oraz nazwę grupy aktywów.
- TotalQuantity
  - **Typ:** double
  - **Dostęp:** public get, public set
  - **Opis:** Sumaryczna ilość jednostek danego aktywa w portfelu.
- AveragePurchasePrice
  - **Typ:** double
  - **Dostęp:** public get, public set
  - **Opis:** Średnia ważona cena zakupu.
- TotalCost
  - **Typ:** double
  - **Dostęp:** public get, public set
  - **Opis:** Całkowity koszt zakupu (zainwestowany kapitał) dla danej grupy aktywów.
- TotalValue
  - **Typ:** double
  - **Dostęp:** public get, public set
  - **Opis:** Aktualna wartość rynkowa wszystkich posiadanych jednostek tego aktywa.

        - **Logika:** Setter przypisuje wartość do prywatnego pola totalValue. Wywołuje OnPropertyChanged() dla samej siebie i wywołuje OnPropertyChanged(nameof(TotalProfit)), aby automatycznie odświeżać w widoku nie tylko wartość, ale i wynikowy zysk, gdy tylko zmieni się cena rynkowa.
    - TotalProfit
        - **Typ:** double
        - **Dostęp:** public get
        - **Logika:** Zwraca wynik odejmowania: TotalValue - TotalCost. Zawsze jest wartością wyliczalną.

2.5.2 Zdarzenia

- PropertyChanged
    - **Typ:** PropertyChangedEventHandler?
    - **Opis:** Implementacja interfejsu INotifyPropertyChanged.
    - Uzasadnienie: Umożliwia wiązanie danych w WPF. Dzięki temu widok automatycznie reaguje na zmiany wartości wewnątrz obiektu.
- OnPropertyChanged([CallerMemberName] string? propertyName = null)
    - **Dostęp:** protected void
    - **Opis:** Metoda pomocnicza do bezpiecznego wywoływania zdarzenia PropertyChanged.
    - **Uzasadnienie (**protected**):** Przeznaczona wyłącznie do użytku wewnętrznego klasy (używana w setterze TotalValue). Obiekt sam powinien informować o zmianie swojego stanu.

2.6 KLASA INVESTMENTPORTFOLIO

Opis: Klasa pomocnicza która nie stanowi encji bazodanowej. Jej głównym celem jest przechowywanie dynamicznie obliczonego podsumowania dla grupy aktywów tego samego typu (np. łączna pozycja dla wszystkich akcji Apple, niezależnie od liczby transakcji). Służy jako warstwa pośrednia dla widoku (WPF), umożliwiając łatwe wiązanie danych

2.6.1 Właściwości

AssetSymbol oraz AssetName

- **Typ:** string
- **Dostęp:** public get, public set
- **Opis:** Przechowują symbol giełdowy oraz pełną nazwę grupy aktywów.
- **Uzasadnienie:** Właściwości są publiczne, aby umożliwić ich ustawienie przez logikę grupującą w klasie InvestmentPortfolio oraz odczyt przez kontrolki w interfejsie użytkownika. O

TotalQuantity

- **Typ:** double
- **Dostęp:** public get, public set
- **Opis:** Sumaryczna ilość jednostek danego aktywa w portfelu.

AveragePurchasePrice

- **Typ:** double

- **Dostęp:** public get, public set
- **Opis:** Średnia ważona cena zakupu dla całej pozycji.

TotalCost

- **Typ:** double
- **Dostęp:** public get, public set
- **Opis:** Całkowity koszt zakupu (zainwestowany kapitał). Obliczany jako suma iloczynów ceny zakupu i ilości dla poszczególnych aktywów składowych.

TotalValue

- **Typ:** double
- **Dostęp:** public get, public set
- **Opis:** Aktualna, łączna wartość rynkowa wszystkich posiadanych jednostek tego aktywa. Setter jest rozbudowany – przypisuje wartość do pola prywatnego, a następnie wywołuje metodę OnPropertyChanged() dla właściwości TotalValue oraz, co kluczowe, dla właściwości TotalProfit. Dzięki temu zmiana wyceny rynkowej automatycznie odświeża w widoku informację o zysku, bez konieczności ręcznego przeliczania.

TotalProfit

- **Typ:** double
- **Dostęp:** public get
- **Opis:** Bieżący zysk lub strata na całej pozycji. Właściwość wyliczalna (read-only), zwracająca różnicę: TotalValue - TotalCost. Jej wartość jest zawsze aktualna, ponieważ zależy bezpośrednio od TotalValue.

2.6.2 Zdarzenia i metody

PropertyChanged

- **Typ:** PropertyChangedEventHandler?
- **Opis:** Zdarzenie wymagane do obsługi powiadomień w technologii WPF.
- **Uzasadnienie:** Pozwala na automatyczną aktualizację interfejsu użytkownika (np. tabeli portfela) w momencie, gdy w tle zmieniają się wartości liczbowe (np. podczas symulacji rynkowej).

OnPropertyChanged([CallerMemberName] string? propertyName = null)

- **Dostęp:** protected void A
- **Opis:** Metoda pomocnicza służąca do podnoszenia zdarzenia PropertyChanged.
- **Uzasadnienie:** Wykorzystuje atrybut [CallerMemberName], co upraszcza kod (nie trzeba podawać nazwy właściwości w postaci łańcucha znaków). Jest oznaczona jako protected, aby była dostępna tylko wewnątrz klasy.

2.7 KLASA STATYCZNA MARKETSIMULATOR

Metoda public static double GenerateNewPrice(double currentPrice, double meanReturn, double volatility)

Opis: Implementuje matematyczny model **Geometrycznego Ruchu Browna**.

Parametry: currentPrice – aktualna cena aktywa, meanReturn – oczekiwana stopa zwrotu, volatility – siła wahań losowych.

Algorytm:

1. Generowanie losowości: Metoda potrzebuje liczby losowej o rozkładzie normalnym, komputerowe generatory dają rozkład jednostajny. Aby to przekształcić, zastosowano Transformację Boxa-Mullera:
   double randStdNormal = Math.Sqrt(-2.0 * Math.Log(u1)) * Math.Sin(2.0 * Math.PI * u2);
2. Obliczenie składników wzoru GBM:
   a. Drift (Trend deterministyczny): meanReturn - (0.5 * volatility^2). Odzwierciedla długoterminowy kierunek ceny.
   b. Shock (Element losowy): volatility * randStdNormal. Jest to nieprzewidywalny ruch w górę lub w dół.
3. Wyliczenie nowej ceny: Cena zmienia się wykładniczo: currentPrice * Math.Exp(drift + shock). Użycie funkcji wykładniczej (Exp) gwarantuje, że cena nigdy nie spadnie poniżej zera.

2.8 INTERFEJSY

2.8.1 IAsset

Definiuje on fundamentalny kontrakt (wspólny zestaw zachowań) dla każdego instrumentu finansowego w systemie. Dzięki niemu aplikacja może zarządzać różnymi typami aktywów (akcje, obligacje, nieruchomości) w sposób polimorficzny.

Metody:

- void SimulatePriceChange(DateTime simulationDate) – Wymusza na klasie implementującej zdefiniowanie unikalnej logiki zmiany ceny w czasie (np. losowej dla akcji, stałej dla obligacji).
- RiskEnum GetRiskAssessment() – Wymusza zaimplementowanie metody zwracającej poziom ryzyka inwestycyjnego (Low, Medium, High, ExtremelyHigh), co pozwala na automatyczną ocenę profilu ryzyka całego portfela.

2.8.2 IDataService

Jest to interfejs publiczny, kluczowy dla warstwy dostępu do danych (Data Access Layer). Realizuje wzorzec odwrócenia zależności (Dependency Inversion) – logika aplikacji (ViewModel) zależy od tej abstrakcji, a nie od konkretnej klasy obsługującej pliki XML czy bazę SQL. Umożliwia to łatwą podmianę źródła danych bez ingerencji w kod aplikacji.

Metody:

- void SavePortfolio(InvestmentPortfolio portfolio) – Metoda odpowiedzialna za trwały zapis stanu całego portfela. Przyjmuje obiekt InvestmentPortfolio, co pozwala na zapisanie zarówno metadanych (Właściciel, Nazwa), jak i powiązanej z nim listy aktywów, zachowując spójność relacyjną.
- InvestmentPortfolio LoadPortfolio() – Odpowiada za odczyt danych i materializację obiektów domenowych. Zwraca pełny obiekt portfela wraz z załadowanymi aktywami, gotowy do wyświetlenia w aplikacji.

2.9 KLASA ASSETRISKCOMPARER

Komparator umożliwiający sortowanie aktywów według ich poziomu ryzyka.

Metoda public int Compare(Asset? x, Asset? y)

Opis: Implementuje interfejs IComparer<Asset>. Porównuje dwa obiekty aktywów (x i y) pod kątem ich oceny ryzyka.

Logika działania:

1. **Obsługa nulli:** Sprawdza, czy któryś z obiektów nie jest nullem (if (x == null || y == null) return 0;). Jest to zabezpieczenie przed błędem NullReferenceException podczas sortowania listy zawierającej puste referencje.
2. **Pobranie ryzyka:** Wywołuje metodę GetRiskAssessment() na obu obiektach.
3. **Porównanie enumów:** Zwraca wynik porównania wartości enumów (CompareTo), porównanie odbywa się zgodnie z kolejnością zdefiniowaną w RiskEnum (Low < Medium < High < ExtremelyHigh).

2.10 TYPY WYLICZENIOWE

- RiskEnum
  - **Opis:** Definiuje skalę ryzyka inwestycyjnego. Jest wykorzystywany przez metodę GetRiskAssessment() w klasach aktywów oraz przez AssetRiskComparer. Kolejność wartości ma znaczenie (od najmniejszego do największego ryzyka).
  - **Wartości:** Low (0), Medium (1), High (2), ExtremelyHigh (3)
- UnitEnum
  - **Opis:** Służy do określania jednostek dla klasy Commodity.
  - **Wartości:**
    - Ounce (Uncja) – np. dla złota, srebra.
    - Barrel (Baryłka) – np. dla ropy naftowej.
    - Ton (Tona) – np. dla węgla, stali.
    - Kilogram – uniwersalna jednostka wagowa.
    - Gram – dla metali rzadkich.
    - Bushel – dla zbóż.
    - Liter – dla płynów innych niż ropa.
    - MWh (Megawatogodzina) – dla energii elektrycznej lub gazu.

3 OPIS TESTÓW JEDNOSTKOWYCH

3.1 KLASA ASSETTESTS

Cel

Klasa odpowiada za testy jednostkowe fundamentalnych komponentów systemu: klasy bazowej Asset oraz jej pochodnych (Stock, Cryptocurrency). Jej głównym zadaniem jest weryfikacja poprawności obliczeń finansowych, walidacja danych wejściowych oraz sprawdzenie mechanizmów agregacji danych.

Rola w projekcie

Zapewnia integralność danych na najniższym szczeblu architektury. Poprawna walidacja w konstruktorach oraz precyzyjne wyliczanie wartości aktywów są krytyczne dla wiarygodności raportów finansowych całego portfela. Testy grupowania gwarantują natomiast poprawność logiczną widoków podsumowujących stan posiadania użytkownika.

Kluczowe przypadki testowe

3.1.1 Value_ShouldCalculateCorrectly_WhenQuantityAndPriceAreSet
Weryfikuje, czy właściwość Value poprawnie wylicza łączną wartość aktywa jako iloczyn ilości i ceny jednostkowej. Jest to kluczowy parametr dla wyceny całego portfela.

3.1.2 Constructor_ShouldThrowException_WhenQuantityIsNegative
Sprawdza mechanizmy obronne systemu. Test potwierdza, że próba utworzenia aktywa z ujemną ilością skutkuje wyrzuceniem wyjątku InvalidQuantityException.

3.1.3 Constructor_ShouldThrowException_WhenNameIsEmpty
Gwarantuje, że system nie dopuści do utworzenia obiektu o nieprawidłowym stanie (brak nazwy), wymuszając rzucenie wyjątku AssetNameException.

3.1.4 AddNewAsset_ShouldGroupSameAssetsInSummary
Testuje logikę biznesową agregacji. Sprawdza, czy przy dodaniu powtarzających się aktywów (np. tej samej kryptowaluty), system poprawnie sumuje ich wolumen i oblicza średnią ważoną cenę zakupu.

3.1.5 FindAssets_ShouldReturnOnlyMatchingAssets
Potwierdza skuteczność silnika filtrowania, sprawdzając, czy metody wyszukiwania zwracają wyłącznie rekordy spełniające zadane kryteria cenowe.

Implementacja:

Pola i modyfikatory dostępu:

private InvestmentPortfolio _portfolio: Pole zdefiniowane jako prywatne w celu zapewnienia pełnej enkapsulacji. Dzięki temu stan portfela jest odizolowany i niedostępny dla innych klas testowych, co zapobiega efektom ubocznym podczas wykonywania testów.

Inicjalizacja (Setup): Obiekt _portfolio jest inicjalizowany w metodzie oznaczonej atrybutem [SetUp]. Gwarantuje to, że każdy test jednostkowy operuje na świeżej instancji obiektu, co zapewnia pełną izolację testów i powtarzalność wyników.

3.2 KLASA EVENTTESTS

Cel:

Weryfikacja mechanizmów komunikacji międzyobiektowej opartej na zdarzeniach. Testy skupiają się na reakcji systemu na dynamiczne zmiany cen rynkowych.

Rola w projekcie

Gwarantuje niezawodność krytycznych powiadomień finansowych (np. stop-loss, alerty cenowe). Klasa potwierdza, że zdarzenie OnCriticalDrop jest emitowane natychmiast po przekroczeniu zdefiniowanego progu cenowego, co zapewnia stabilność warstwy reaktywnej aplikacji.

Główny przypadek testowy

3.2.1 OnCriticalDrop_ShouldFire_WhenPriceDropsBelowThreshold

Symuluje gwałtowny spadek ceny aktywa (np. kryptowaluty) poniżej progu LowPriceThreshold. Test weryfikuje poprawność subskrypcji poprzez sprawdzenie stanu flagi eventFired.

Implementacja

Klasa bezstanowa. Wszystkie metody testowe są publiczne, co umożliwia ich automatyczną detekcję przez runnera testów (np. xUnit, NUnit).

3.3 KLASA INVESTMENTPORTFOLIOTESTS

Cel

Klasa ta koncentruje się na weryfikacji logiki biznesowej głównego agregatu systemu — portfela inwestycyjnego (InvestmentPortfolio). Testy sprawdzają poprawność zarządzania kolekcją aktywów, mechanizmy sumowania ich wartości oraz walidację danych identyfikacyjnych właściciela.

Rola w projekcie

InvestmentPortfolio pełni funkcję centralnego kontenera danych. Testy te są krytyczne dla zapewnienia integralności finansowej systemu (metoda CalculateSum), gwarantując, że zagregowana wartość portfela zawsze odzwierciedla stan faktyczny. Dodatkowo, walidacja pola owner za pomocą wyrażeń regularnych (Regex) zapewnia spójność bazy danych i chroni przed wprowadzaniem niekompletnych lub błędnych informacji.

Kluczowe przypadki testowe

3.3.1 AddNewAsset_ShouldIncreaseAssetCount

Test integralności kolekcji. Weryfikuje, czy po dodaniu nowego obiektu (np. typu Stock), licznik elementów wewnątrz portfela jest poprawnie inkrementowany, a sam obiekt zostaje trwale umieszczony w strukturze danych.

3.3.2 CalculateSum_ShouldReturnTotalValueOfAllAssets

Sprawdza poprawność algorytmu sumowania wartości całego portfela. Test łączy różne typy aktywów (np. akcje i obligacje) o różnych wycenach, weryfikując, czy metoda CalculateSum precyzyjnie agreguje dane finansowe.

3.3.3 Owner_ValidationRegex_ShouldWorkCorrectly

Test parametryczny (DataRow), który kompleksowo sprawdza walidację nazwiska właściciela. Scenariusze testowe obejmują:

- Dane poprawne (Imię i Nazwisko).
- Błędy formatowania (brak wielkich liter).
- Zbyt krótkie ciągi znaków.
- Znaki niedozwolone (cyfry, symbole specjalne).

Zapewnia to, że pole Owner spełnia standardy biznesowe i techniczne systemu.

Implementacja

Pola i modyfikatory dostępu:

(takie same jak w klasie AssetTest)

private InvestmentPortfolio _portfolio: Pole zdefiniowane jako prywatne w celu zapewnienia pełnej enkapsulacji. Dzięki temu stan portfela jest odizolowany i niedostępny dla innych klas testowych, co zapobiega efektom ubocznym podczas wykonywania testów.

Inicjalizacja (Setup): Obiekt _portfolio jest inicjalizowany w metodzie oznaczonej atrybutem [SetUp]. Gwarantuje to, że każdy test jednostkowy operuje na świeżej instancji obiektu, co zapewnia pełną izolację testów i powtarzalność wyników.

3.4 KLASA SIMULATIONTESTS

Cel

Klasa ta służy do weryfikacji poprawności algorytmów symulujących dynamikę rynkową. Skupia się na badaniu zmian wartości aktywów w czasie oraz monitorowaniu stabilności numerycznej silnika obliczeniowego, eliminując ryzyko wystąpienia błędów w implementacji wzorów matematycznych.

Rola w projekcie

W systemie zaimplementowano zróżnicowane modele wzrostu dla poszczególnych klas aktywów (np. deterministyczny wzrost obligacji vs. zmienność akcji). Testy te stanowią barierę ochronną przed błędami typu NaN (Not a Number), Infinity czy wartościami ujemnymi, które mogłyby doprowadzić do destabilizacji modułu finansowego i błędnych raportów.

Kluczowe przypadki testowe

3.4.1 UpdateMarketPrices_Bond_ShouldIncreaseDeterministically

Weryfikuje logikę wzrostu obligacji. Test potwierdza, że cena aktywa rośnie w sposób przewidywalny i liniowy, zgodnie z założonym oprocentowaniem i upływem czasu (wzrost deterministyczny).

3.4.2 UpdateMarketPrices_Stock_ShouldNotBeNaN_Or_Infinity

Test odpornościowy (Stress Test). Przeprowadza wielokrotne iteracje (symulacje) zmian cen akcji, aby upewnić się, że algorytm – nawet przy dużej zmienności – nie generuje błędnych wartości numerycznych. Zapewnia to, że obliczenia pozostają w bezpiecznych granicach matematycznych.

Implementacja

Pola i modyfikatory dostępu:

(takie same jak w klasie AssetTest)

private InvestmentPortfolio _portfolio: Pole zdefiniowane jako prywatne w celu zapewnienia pełnej enkapsulacji. Dzięki temu stan portfela jest odizolowany i niedostępny dla innych klas testowych, co zapobiega efektom ubocznym podczas wykonywania testów

4 OPIS KONFIGURACJI BAZY DANYCH

4.1 KLASA INVESTMENTPORTFOLIODBCONTEXT

Klasa dziedziczy po klasie DbContext. Pełni rolę pośrednika pomiędzy kodem aplikacji opartej o Entity Framework Core, a bazą danych postawioną na MS SQL Server.

4.1.1 Zdefiniowane w niej są 3 tabele:

- **DbSet<InvestmentPortfolio> Portfolios**
  która przechowuje nagłówki portfeli inwestycyjnych użytkowników.
- **DbSet<Asset> Assets**
  przechowuje wszystkie informacje o posiadanych aktywach (i ich typach) użytkownika
- **DbSet<PricePoint> PricePoints**
  przechowuje historię notowań cen aktywów

4.1.2 Klasa DbContext zawiera również metody:

- **protected override void OnConfiguring(DbContextOptionsBuilder options)**
  odpowiedzialna za konfigurację połączenia z bazą danych. Domyślnie używany jest w niej lokalny SQL Server użytkownika.
- **protected override void OnModelCreating(ModelBuilder modelBuilder)**
  jej zadaniem jest definiowanie reguł mapowania encji na strukturę bazy.

Wszystkie klasy, które dziedziczą po klasie Asset, są przechowywane w tabeli Assets. Za rozróżnienie typu aktywa, odpowiada kolumna AssetType. Z możliwych typów mamy: Stock, Bond, Cryptocurrency, RealEstate, Commodity.

4.2 KLASA SQLDATABASESERVICE

Klasa implementuje wcześniej wspomniany interfejs IDataService. Jest ona komunikatorem pomiędzy aplikacją, a bazą danych. Obsługuje ona wczytywanie i zapisywanie danych do bazy. Zawiera również metodę odpowiedzialną, za wygenerowanie przykładowych danych, w wypadku kiedy baza danych jest pusta.

Opis poszczególnych metod:

4.2.1 public void SavePortfolios(List<InvestmentPortfolio> portfolios)

sprawdza czy dany portfel już istnieje w bazie, w przypadku gdy go nie ma, dodaje nowy portfel, aktualizuje już istniejące portfele, po dokonaniu zmian; dodawanie, usuwanie i aktualizowanie stanu aktyw.

4.2.2 public List<InvestmentPortfolio> LoadAllPortfolios()

metoda odpowiedzialna za załadowanie danych z bazy do naszej aplikacji; pobiera ona wszystkie istniejące portfele, łączy z nimi przypisane im aktywa i ich historię cen, zwraca strukturę obiektów gotową, do użycia w aplikacji.

5 SZCZEGÓŁOWY OPIS WARSTWY PREZENTACJI (GUI)

Warstwa prezentacji została zrealizowana w technologii **WPF (Windows Presentation Foundation)** przy użyciu wzorca **MVVM**. Taka architektura zapewnia separację logiki biznesowej od interfejsu użytkownika, co pozwala na niezależne testowanie komponentów i łatwą rozbudowę interfejsu.

5.1 KLASY INFRASTRUKTURALNE MVVM I DANE

5.1.1 Klasa ViewModelBase

- **Opis:** Klasa abstrakcyjna stanowiąca fundament dla wszystkich modeli widoku. Implementuje interfejs INotifyPropertyChanged.
- **Rola w projekcie:** apewnia mechanizm powiadamiania warstwy widoku o zmianach w danych. Dzięki niej każda zmiana ceny w symulatorze lub aktualizacja komunikatu o zdarzeniu rynkowym jest natychmiast odzwierciedlana w interfejsie bez konieczności ręcznego odświeżania okien.
- **Uzasadnienie:** Metoda OnPropertyChanged posiada modyfikator protected, co umożliwia klasom pochodnym zgłaszanie aktualizacji własnych właściwości przy zachowaniu hermetyzacji mechanizmu powiadomień.

5.1.2 Klasa RelayCommand

- **Opis:** Implementacja interfejsu ICommand oparta na delegatach.
- **Rola w projekcie:** Pozwala na mapowanie akcji użytkownika (np. kliknięcie przycisku „Dodaj” lub „Usuń”) na konkretne metody w klasie MainViewModel.
- **Uzasadnienie:** Wykorzystuje delegaty **Action** (logika wykonawcza) oraz **Predicate** (warunek dostępności komendy). Modyfikatory public dla metod Execute i CanExecute umożliwiają frameworkowi WPF sterowanie aktywnością kontrolek w zależności od stanu logicznego aplikacji.

5.1.3 Klasa FileDataService

- **Opis:** Klasa implementująca interfejs IDataService, odpowiedzialna za operacje wejścia/wyjścia.
- **Rola w projekcie:** Zarządza trwałym przechowywaniem portfela inwestycyjnego. Obsługuje serializację do formatu XML oraz odczyt danych z dysku.
- **Uzasadnienie:** Wydzielenie logiki do osobnej klasy serwisowej pozwala na łatwą podmianę źródła danych (np. przejście z plików lokalnych na bazę danych SQL) bez modyfikacji kodu w warstwie prezentacji.

5.2 WIDOKI (VIEWS) I OKNA

5.2.1 Klasa App

- **Opis:** Główna klasa aplikacji dziedzicząca po System.Windows.Application.
- **Rola w projekcie:** Zarządza cyklem życia aplikacji i zasobami globalnymi.

- **Uzasadnienie:** W pliku App.xaml zdefiniowano wspólne style dla kontrolek oraz zasoby kolorystyczne (Brushes), co zapewnia spójność wizualną (UI Consistency) całego systemu.

5.2.2 Klasa MainWindow

- **Opis:** Główne okno aplikacji.
- **Rola w projekcie:** Pełni funkcję kontenera nadrzędnego. Zawiera główne menu nawigacyjne, obszar ContentControl służący do dynamicznego wyświetlania Dashboardu lub Portfela stopkę z paskiem wiadomości rynkowych.
- **Uzasadnienie:** Wykorzystuje bindowanie właściwości CurrentView, co umożliwia płynną nawigację między modułami wewnątrz jednego okna. Dedykowany pasek na dole okna eksponuje pilne komunikaty z symulatora.

5.2.3 Klasa PortfolioView

- **Opis:** Komponent typu UserControl prezentujący szczegółową listę posiadanych aktywów.
- **Rola w projekcie:** Umożliwia przeglądanie i modyfikację składników portfela. Wykorzystuje kontrolkę DataGrid z warunkowym formatowaniem (kolorowanie zysków i strat).
- **Uzasadnienie:** Klasa typu partial pozwala na rozdzielenie definicji interfejsu (XAML) od kodu inicjalizującego, co ułatwia zarządzanie projektem.

5.2.4 Klasa Dashboard

- **Opis:** Widok podsumowujący (UserControl).
- **Rola w projekcie:** Prezentuje kluczowe statystyki całego portfela, takie jak całkowita wartość rynkowa oraz podział procentowy aktywów.
- **Uzasadnienie:** Agreguje dane z wielu obiektów w skondensowaną formę, ułatwiając użytkownikowi szybką ocenę sytuacji finansowej.

5.2.5 Klasa AddAssetWindow

- **Opis:** Okno dialogowe służące do wprowadzania nowych transakcji.
- **Rola w projekcie:** Zapewnia dedykowany interfejs dla formularza dodawania aktywa.
- **Uzasadnienie:** Zdefiniowane jako osobne okno (Window), aby wymusić skupienie użytkownika na procesie wprowadzania danych i umożliwić walidację formularza przed jego zatwierdzeniem.

5.3 Modele Widoku (ViewModels)

5.3.1 Klasa MainViewModel

- **Opis:** Główny model widoku zarządzający stanem całej aplikacji.
- **Rola w projekcie:** Przechowuje kolekcję ObservableCollection<Asset>, która zasila widok portfela, oraz zarządza logiką przełączania widoków (Dashboard/Portfolio). Obsługuje wszystkie komendy użytkownika przesyłane z interfejsu. Obsługuje DispatcherTimer, który cyklicznie aktualizuje ceny rynkowe. Implementuje logikę

losowych zdarzeń, zarządzając ich czasem trwania i wpływem na parametry aktywów. Udostępnia właściwość NewsMessage, która steruje paskiem informacyjnym w głównym oknie.

- **Uzasadnienie:** Centralizacja logiki w MainViewModel ułatwia zarządzanie wspólnymi danymi (np. całkowitą sumą portfela) i upraszcza komunikację między różnymi komponentami GUI.