# OOP #2

## OBJECT-ORIENTED PROGRAMMING #2

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

# AGENDA

1. dziedziczenie
2. wielodziedziczenie
3. funkcje wirtualne
4. funkcje czysto wirtualne
5. klasy abstrakcyjne
6. interfejsy
7. polimorfizm
8. pola i metody statyczne

Możliwe, że dzisiaj nie przerobimy wszystkiego. Będzie na kolejną lekcję :)

# ZADANIA

Repo GH `coders-school/object-oriented-programming`

https://github.com/coders-school/object-oriented-programming/tree/master/module2

# KILKA PYTAŃ

- Co z zadań domowych należy jeszcze wyjaśnić?
- Czy któreś zadania były niejednoznacznie opisane?
- Ile zadań macie już wykonanych?

# PROGRAMOWANIE OBIEKTOWE

## DZIEDZICZENIE

CODERS SCHOOL

# WPROWADZENIE DO DZIEDZICZENIA

Podczas implementacji klas, często możemy zauważyć, że część cech składowych klasy można wykorzystać także w innych klasach.

Weźmy pod lupę klasę `Computer`. Jeżeli chcielibyśmy utworzyć klasy: `Laptop`, `PC`, `Tablet`, to część metod oraz składowych klasy musielibyśmy powielić.

# class Computer

```
class Computer {
public:
    void turnOn();
    void powerOff();
    void restart();

private:
    Processor processor_;
    Drive drive_;
    Motherboard motherboard_;
    GraphicsCard graphics_card_;
    Memory memory_;
};
```

# class Laptop

```
class Laptop {
public:
    void turnOn();
    void powerOff();
    void restart();
    void display();
    void getUserInput();

private:
    Processor processor_;
    Drive drive_;
    Motherboard motherboard_;
    GraphicsCard graphics_card_;
    Memory memory_;
    Screen screen_;
    Keyboard keyboard_;
};
```

# class Tablet

```
class Tablet {
public:
    void turnOn();
    void powerOff();
    void restart();
    void display();
    void getUserInput();

private:
    Processor processor_;
    Drive drive_;
    Motherboard motherboard_;
    GraphicsCard graphics_card_;
    Memory memory_;
    Screen screen_;
    Keyboard keyboard_;
};
```

# JAK UPROŚCIĆ STRUKTURĘ NASZEGO PROGRAMU?

```
class Computer {
public:
    void turnOn();
    void powerOff();
    void restart();

protected:
    Processor processor_;
    Drive drive_;
    Motherboard motherboard_;
    GraphicsCard graphics_card_;
    Memory memory_;
};


class Laptop : public Computer {
public:
    void display();
    void getUserInput();

private:
    Screen screen_;
    Keyboard keyboard_;
```

# KLASY BAZOWE I POCHODNE

Klasa, po której dziedziczymy, nazywają się **klasą bazową (base class)**.

Klasa, która dziedziczy nazywa się **klasą pochodną (derived class)**.

Inaczej, klasa, po której dziedziczymy to rodzic (parent class).

Klasa, która dziedziczy to dziecko (child class).

# CO Z METODAMI KLAS `Laptop` I `Tablet`?

## CZY MOŻNA WYDZIELIĆ KOLEJNĄ KLASĘ?

```
void display();
void getUserInput();
```

# KLASA `Screen` I `TouchScreen`

Załóżmy, że dodajemy klasę `Screen`. Klasa ta wyświetla na bieżąco interfejs użytkownika.

Chcemy też stworzyć klasę reprezentującą ekran dotykowy - `TouchScreen`, który również umożliwia odczyt akcji od użytkownika i ich wyświetlanie.

```
class Screen {
public:
    void display();

private:
    void process();

    Monitor monitor_;
};
```

```
class TouchScreen {
public:
    void display();
    void getUserInput();

private:
    void process();
    void displayKeyboard();

    Monitor monitor_;
};
```

## JAK UPROŚCIĆ POWYŻSZY KOD?

# WYKORZYSTANIE DZIEDZICZENIA DO UPROSZCZENIA KODU

```
class Screen {
public:
    void display();

private:
    void process();

    Monitor monitor_;
};
```

```
class TouchScreen : public Screen {
public:
    void getUserInput();

private:
    void displayKeyboard();
};
```

# WIELODZIEDZICZENIE

```
class Screen {
public:
    void display();

private:
    void process();

    Monitor monitor_;
};

class TouchScreen : public Screen {
public:
    void getUserInput();

private:
    void displayKeyboard();
};

class Computer {
public:
    void turnOn();
```

# WIELODZIEDZICZENIE - DISCLAIMER

Wielodziedziczenie to dziedziczenie z kliku klas bazowych.

Wybór implementacji zależy od programisty.

Nie zawsze wielodziedziczenie będzie lepszym rozwiązaniem.

Należy się zawsze zastanowić czy dziedziczenie po konkretnej klasie uprości nam program i czy nie będzie powodować żadnych komplikacji w dalszym procesie rozbudowy naszego programu.

Najwyżej trzeba będzie refaktoryzować program ;)

# DZIEDZICZENIE - PROBLEMY

```
struct Bird {
    fly();
    makeSound();
};

struct Penguin {
    swim();
    makeSound();
};

// Hummingbird is the type of bird which makes so little sound so it
// can be said that it makes no sound.
struct Hummingbird {
    fly();
};
```

# DZIEDZICZENIE - ZASADA LSP

Jeżeli spróbujemy teraz uprościć klasę poprzez dziedziczenie pojawi się problem:

```
struct Bird {
    fly();
    makeSound();
};

struct Penguin : public Bird {
    fly();  // But I can't fly!
    swim();
    makeSound();
};

struct Hummingbird : public Bird {
    fly();
    makeSound();  // But I don't make sound!
};
```

Jeszcze bardziej utrudnimy sytuacje, gdy w przyszłości dodamy sobie kolejne klasy jak Struś. Zawsze przed implementacją musimy się zastanowić jak podzielić odpowiedzialność na poszczególne klasy, aby uniknąć podobnych problemów.

# DLA CIEKAWSKICH

Poczytajcie o zasadzie Liskov Substitution Principle (LSP). Mówi ona jak powinno / nie powinno się projektować kodu obiektowego. Ta zasada została złamana w ostatnim przykładzie.

Możecie też poczytać o wszystkich zasadach SOLID.

# Q&A

# PROGRAMOWANIE OBIEKTOWE

## METODY WIRTUALNE, INTERFEJSY, KLASY ABSTRAKCYJNE

CODERS SCHOOL

# METODY CZYSTO WIRTUALNE

```
class Bird {
public:
    size_t getWeight();
    size_t getHeight();
    size_t getName();

    // Pure virtual function without implementation
    virtual void eat() = 0;
    virtual void sleep() = 0;

protected:
    size_t weight_;
    size_t height_;
    std::string name_;
};
```

Metody `eat()` oraz `sleep()` to tzw. metody czysto wirtualne. Świadczy o tym `= 0;`. Oznacza to, że nigdzie nie znajdziemy ich implementacji dla klasy `Bird`. Klasy które dziedziczą po `Bird` będą ją musiały zaimplementować same.

Znaczenie słowa `virtual` na razie przemilczymy. Jedyne co trzeba na teraz wiedzieć, to aby metoda była czysto wirtualna `= 0;` to musi być przed nią słowo `virtual`.

# INTERFEJSY

Jednym z pomysłów na rozwiązanie problemu wielodziedziczenia jest tworzenie tzw. interfejsów, ich dziedziczenie oraz przeciążanie implementacji metod z klas bazowych. Interfejsy określają "umiejętności" i łatwo je ze sobą łączyć.

```
class Flyable {
public:
    virtual void fly() = 0;

private:
    Wings wings_;
};

class Swimmable {
public:
    virtual void swim() = 0;
};

class Soundable {
public:
    virtual void makeSound() = 0;
};
```

# UŻYCIE INTERFEJSÓW

```
class Penguin : public Bird, public Swimmable {
public:
    // Override from Bird
    void eat() override;
    void sleep() override;

    // Override from Swimmable
    void swim() override;
};

class Hummingbird : public Bird,
                    public Flyable,
                    public Soundable {
public:
    // Override from Bird
    void eat() override;
    void sleep() override;

    // Override from Soundable
    void makeSound() override;
```

# CO TO JEST INTERFEJS?

Interfejs to zestaw funkcji, które klasa implementująca go musi zaimplementować (masło maślane).

Interfejs to zestaw funkcji, które klasa dziedzicząca po nim musi zaimplementować.

Brak implementacji funkcji czysto wirtualnej to błąd linkera (undefined reference).

# DEFINICJA INTERFEJSU

W pełni poprawna interfejsu to część publiczna klasy / zestawu funkcjonalności. Mogą to być zarówno metody, pola, typu, ale najczęściej będziemy słowa interfejs używać w odniesieniu do publicznych metod klasy.

Zobacz interfejs wektora na cppreference.com

Znajdziesz tam opis jego publicznych metod (member functions), typów wewnętrznych (member types) oraz luźnych funkcji, które go wykorzystują (non-member functions).

# KLASA ABSTRAKCYJNA

Nie można utworzyć obiektu klasy, która posiada **co najmniej jedną funkcję czysto wirtualną**.

Funkcja czysto wirtualna nie ma implementacji, więc nie może istnieć obiekt, dla którego linker nie znajdzie implementacji jednej z jego funkcji.

Możemy przechowywać wskaźnik wskazujący na typ tej klasy, ale nie możemy stworzyć jej instancji (obiektu), ponieważ nie mamy zdefiniowanych jej zachowań.

Klasa taka nazywa się **klasą abstrakcyjną** i służy tylko do ujednolicania interfejsu, a nie tworzenia obiektów.

Dopiero obiekt klasy pochodnej, która zaimplementuje wszystkie brakujące metody, może zostać utworzony.

# SŁOWO `virtual` i `override`

Co to za słowa? Co one robią? O tym za chwilę ;)

# Q&A

# ZADANIE 1

Przekształć klasę `Cargo` w interfejs z 4 czysto wirtualnymi metodami.

```
virtual size_t getPrice() const = 0;
virtual std::string getName() const = 0;
virtual size_t getAmount() const = 0;
virtual size_t getBasePrice() const = 0;
```

# ZADANIE 1 CD.

Utwórz 3 pochodne klasy `Cargo`:

- `Fruit`
- `Alcohol`
- `Item`

Klasa `Fruit` powinna mieć dodatkową zmienną określającą czas do zepsucia oraz `operator--` który będzie odejmował ten czas o 1. Metoda `getPrice()` powinna redukować cenę odpowiednio wraz z czasem psucia naszego owocu.

Klasa `Alcohol` powinna mieć dodatkową zmienną określającą procentowy udział spirytusu. Metoda `getPrice()` powinna być proporcjonalnie wyższa w zależności od mocy alkoholu. Należy ustalić bazową cenę za spirytus 96%.

Klasa `Item` powinna mieć dodatkową zmienną enum określającą rzadkość przedmiotu (common, rare, epic, legendary). Metoda `getPrice()` powinna być adekwatnie wyliczana od poziomu rzadkości przedmiotu.

# ZADANIE 2

Wykorzystując wspólną klasę bazową `Cargo` spróbuj przechowywać wszystkie towary w jednym wektorze w klasie `Ship`.

Dodaj funkcję `void load(std::shared_ptr<Cargo> cargo)`, która dodaje towar i (dla chętnych) `void unload(Cargo* cargo)`, która usuwa towar z obiektu klasy `Ship`.

# PROGRAMOWANIE OBIEKTOWE

## POLIMORFIZM

CODERS SCHOOL

# SŁOWO KLUCZOWE `virtual`

Jeżeli chcemy, aby przy używaniu wskaźników lub referencji na klasę bazową, jakaś metoda zachowywała się inaczej w zależności od prawdziwego typu obiektu, to należy ją oznaczyć słowem kluczowym `virtual`. Jest to tzw. funkcja wirtualna.

# FUNKCJA NIE-WIRTUALNA

```cpp
#include <iostream>

struct Bird {
    void sing() { std::cout << "tweet, tweet\n"; }
};

struct Sparrow : Bird {
    void sing() { std::cout << "chirp, chirp\n"; }
};

int main() {
    Sparrow sparrow;
    Bird& bird = sparrow;
    bird.sing();
    return 0;
}
```

Co pojawi się na ekranie?

`tweet, tweet`

# FUNKCJA WIRTUALNA

```
#include <iostream>

struct Bird {
    virtual void sing() { std::cout << "tweet, tweet\n"; }
};

struct Sparrow : Bird {
    void sing() { std::cout << "chirp, chirp\n"; }
};

int main() {
    Sparrow sparrow;
    Bird& bird = sparrow;
    bird.sing();
    return 0;
}
```

Co pojawi się na ekranie?

`chirp, chirp`

Sprawdź na ideone.com

# SŁOWO KLUCZOWE `override`

Jeżeli w klasie pochodnej **nadpisujemy** metodę wirtualną, czyli zmieniamy jej zachowanie, to należy dodać słowo `override`.

```
class Interface {
public:
    virtual void doSth() = 0;
};

class SomeClass : public Interface {
public:
    doSth() override;   // there should be an implementation in cpp file
};

int main() {
    Interface interface;    // Compilation error, Interface is pure virtual
    SomeClass someClass;    // OK
    Interface* interface = &someClass;  // OK, we hold a pointer
}
```

# MAŁA UWAGA

`override` jest opcjonalne. Jeśli go nie podamy za sygnaturą funkcji klasy pochodnej to metoda z klasy bazowej i tak zostanie nadpisana.

Jego użycie jest jednak dobrą praktyką, bo dzięki niemu kompilator sprawdzi czy faktycznie przeciążamy metodą z klasy bazowej i jeśli nie, to program się nie skompiluje.

Bez `override` mogłaby zostać utworzona nowa metoda w klasie pochodnej, która nie nadpisuje niczego z klasy bazowej.

Metody wirtualne **nadpisujemy**, nie przeciążamy.

# NADPISYWANIE METOD - `override`

Wracając do przykładu o ptakach, klasy `Penguin`, `Hummingbird` oraz `Goose` to klasy pochodne, które dziedziczą po pewnych klasach bazowych jak `Bird`, `Flyable`, `Soundable`, `Swimmable` oraz nadpisują kilka ich metod jak:

- `void eat() override`
- `void sleep() override`
- `void makeSound() override`
- `void fly() override`
- `void swim() override`

Nadpisanie takich metod powoduje, że możemy zmienić ich implementacje.

# override

```
class Soundable {
public:
    virtual void makeSound() = 0;
};
```

```
class Goose : public Soundable {
public:
    void makeSound() override { std::cout << "Honk! Honk!"; }
};
```

```
class Hen : public Soundable {
public:
    void makeSound() override { std::cout << "Cluck! Cluck!"; }
};
```

```
class Duck : public Soundable {
public:
    void makeSound() override { std::cout << "Quack! Quack!"; }
};
```

# WSPÓLNA KLASA BAZOWA

Ponieważ wspólnym rodzicem wszystkich klas jest klasa `Soundable` możemy przechowywać w kontenerze wskaźniki typu `Soundable`.

```
std::vector<std::shared_ptr<Soundable>> birds_;
```

# JAKIE DANE OTRZYMAMY NA WYJŚCIU?

```
std::vector<std::shared_ptr<Soundable>> birds_;
birds_.push_back(std::make_shared<Goose>());
birds_.push_back(std::make_shared<Hen>());
birds_.push_back(std::make_shared<Duck>());

std::cout << birds_[0]->makeSound() << '\n';
std::cout << birds_[1]->makeSound() << '\n';
std::cout << birds_[2]->makeSound() << '\n';
```

# POLIMORFIZM

Zjawisko, które właśnie zaobserwowaliśmy, nazywa się polimorfizmem.

Polimorfizm pozwala funkcji przybrać różne formy (implementacje), tak jak na przykładzie.

Dlatego, jeżeli utworzymy kolejno obiekty `Goose`, `Hen` i `Duck` w zależności od obiektu zostanie wywołana jego wersja metody `makeSound`.

Polimorfizm włącza się, gdy mamy funkcje wirtualne i używamy wskaźników lub referencji na typ bazowy.

## KTO GRAŁ LUB CZYTAŁ WIEDŹMINA?

# DOPPLER :)

W uniwersum wykreowanym przez naszego rodzimego pisarza Andrzeja Sapkowskiego, występuje pewna intrygująca i ciekawa rasa zwana Dopplerami.

Rasa ta potrafi przyjąć, postać różnych form życia, może stać się człowiekiem, elfem, krasnoludem. Zmienia w ten sposób swoje cechy jak głos, kolor włosów, a nawet ubranie!

Pomimo że rasa ta jest typu Doppler, potrafi w różnych okolicznościach podszywać się pod inne rasy jak elf, krasnolud czy człowiek.

Z punktu widzenia C++ nasz Doppler podlega zjawisku polimorfizmu.

```cpp
class Doppler {
public:
    virtual sayHello() { std::cout << "I'm Doppler!"; }
};

class Dwarf : public Doppler {
public:
    virtual sayHello() { std::cout << "I'm Dwarf!"; }
};

class Elf : public Doppler {
public:
    virtual sayHello() { std::cout << "I'm Elf!"; }
};

class Human : public Doppler {
public:
    virtual sayHello() { std::cout << "I'm Human!"; }
};

int main() {
    std::shared_ptr<Doppler> doppler1 = std::make_shared<Dwarf>();
    std::shared_ptr<Doppler> doppler2 = std::make_shared<Elf>();
    std::shared_ptr<Doppler> doppler3 = std::make_shared<Human>();
```

Jak widzimy, nasz Doppler może przyjąć różne formy i zachowywać się tak jak one. Wskaźnik jest typu Doppler, ale program dobrze wie, kiedy Doppler podszywa się pod człowieka, kiedy pod krasnoluda, a kiedy pod elfa.

# NIE-WIRTUALNE DESTRUKTORY

Bardzo ważne w przypadku tworzenia metod wirtualnych i dziedziczenia jest tworzenie wirtualnych destruktorów. Jeżeli korzystamy z dobroci polimorfizmu i nie oznaczymy destruktor jako `virtual` to destruktor ten nie zostanie wywołany.

```
#include <iostream>
#include <string>

class Parent {
public:
    Parent() { std::cout << "PARENT C'tor called\n"; }
    ~Parent() { std::cout << "PARENT D'tor caller\n"; }
};

class Child : public Parent {
public:
    Child() { std::cout << "CHILD C'tor called\n"; }
    ~Child() { std::cout << "CHILD D'tor caller\n"; }
};

int main() {
    Child child;    // ok, object on stack, not a pointer
}
```

# NIE-WIRTUALNE DESTRUKTORY - PROBLEM

```cpp
#include <iostream>
#include <memory>
#include <string>

class Parent {
public:
    Parent() { std::cout << "PARENT C'tor called\n"; }
    ~Parent() { std::cout << "PARENT D'tor caller\n"; }
};

class Child : public Parent {
public:
    Child() { std::cout << "CHILD C'tor called\n"; }
    ~Child() { std::cout << "CHILD D'tor caller\n"; }
};

int main() {
    // Problem
    std::unique_ptr<Parent> child = std::make_unique<Child>();

    // But shared_ptr will cleanup properly
```

# WIRTUALNY DESTRUKTOR

```cpp
#include <iostream>
#include <memory>
#include <string>

class Parent {
public:
    Parent() { std::cout << "PARENT C'tor called\n"; }
    virtual ~Parent() { std::cout << "PARENT D'tor caller\n"; }
};

class Child : public Parent {
public:
    Child() { std::cout << "CHILD C'tor called\n"; }
    ~Child() override { std::cout << "CHILD D'tor caller\n"; }
};

int main() {
    std::unique_ptr<Parent> child2 = std::make_unique<Child>();
}
```

# Q&A

# ZADANIE 3

Napisz klasę `DryFruit`, która dziedziczyć będzie po klasie `Fruit`.

Klasa ta powinna przeciążać metody `getPrice()`, `getName()` oraz `operator--`.

`operator--` powinien odejmować zużycie raz na 10 wywołań.

Metoda `getPrice()` powinna zwracać trzykrotnie większą wartość w porównaniu do ceny bazowej.

Przetestuj wywołania polimorficzne oraz podziel się wnioskami.

# PROGRAMOWANIE OBIEKTOWE

## ZMIENNE I FUNKCJE STATYCZNE

CODERS SCHOOL

# "ZMIENNA LUB STAŁA KLASY"

Czasami chcielibyśmy przypisać jakąś stałą cechę do klasy. Nie konkretnych obiektów, a klasy samej w sobie. Np. każdy obiekt klasy ma nazwę "Object".

```
class Object {
public:
    std::string GetName() const { return name_; }

private:
    const std::string name_ = "Object";
};
```

W celu otrzymania nazwy tego obiektu, musimy najpierw utworzyć obiekt, a następnie zawołać `getName()`.

```
int main() {
    Object obj;
    std::cout << obj.getName() << '\n';
}
```

Nawet jeżeli obiekt zajmowałby dużo miejsca w pamięci a my chcielibyśmy tylko jego nazwę i tak musimy go utworzyć, ponieważ tylko na nim możemy zawołać metodę `getName()`.

# static

Rozwiązaniem tej uciążliwości jest `static`. Co więcej, problem ten możemy rozwiązać na 2 sposoby. Nie musimy w ten sposób tworzyć specjalnie obiektu, aby dostać się do cechy klasy, jaką jest jej nazwa.

```
class ObjectA {
public:
    static std::string getName() { return "ObjectA"; }
};

class ObjectB {
public:
    static std::string name_;
};

std::string ObjectB::name_{"ObjectB"};

int main() {
    std::cout << ObjectA::getName() << '\n';
    std::cout << ObjectB::name_ << '\n';

    return 0;
}
```

# Q&A

# ZADANIE 4

Przekształć klasę bazową `Coordinates`, tak aby miała funkcję statyczną

```
static size_t distance(const Coordinates& lhs, const Coordinates& rhs)
```

Funkcja ta powinna zwracać dystans pomiędzy dwoma pozycjami.

# PROGRAMOWANIE OBIEKTOWE

## PODSUMOWANIE

CODERS SCHOOL

# CO PAMIĘTASZ Z DZISIAJ?

## NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

1. dziedziczenie
2. wielodziedziczenie
3. funkcje wirtualne
4. funkcje czysto wirtualne
5. klasy abstrakcyjne
6. interfejsy
7. polimorfizm
8. pola i metody statyczne

# PRE-WORK

- Dowiedzcie się czym jest problem diamentowy
- Poczytajcie o zasadach SOLID, dotyczących pisania dobrego kodu obiektowego
- Lektura o wzorcach projektowych z przykładami w C++ - refactoring.guru
- Spróbujcie w grupie metodą Copy & Paste dorzucić system budowania cmake do projektu. W tym celu popatrzcie na dotychczasowe zadania domowe i plik CMakeLists.txt.

# PROJEKT GRUPOWY

Wykorzystajcie kod napisany podczas zajęć. Możecie też skorzystać z kodu w katalogu solutions

Projekt grupowy - kontynuacja. Możecie zmienić grupę jeśli chcecie ;)

# ORGANIZACJA PRAC

- Jak wyglądało wasze daily?
- Czy Code Review nie jest zaniedbane?
- Czy współpraca idzie gładko?
- Zróbcie sobie retrospektywę :)

# PUNKTACJA

- 3 pierwsze zadania - 5 punktów
- zadania 4, 5, 6 - 8 punktów
- 20 punktów za dostarczenie wszystkich 6 zadań przed 05.07.2020 (niedziela) do 23:59
- brak punktów bonusowych za dostarczenie tylko części zadań przed 05.07
- 6 punktów za pracę w grupie dla każdej osoby z grupy.

# ZADANIE 1

Napisz klasę `Store`, która będzie umożliwiała dokonywanie zakupów. Wykorzystaj poniższy enum i funkcje.

```
enum class Response {done, lack_of_money, lack_of_cargo, lack_of_space};

Response buy(Cargo* cargo, size_t amount, Player* player);
Response sell(Cargo* cargo, size_t amount, Player* player);
```

# ZADANIE 2

W klasach `Alcohol`, `Fruit`, `Item` dopisz brakujące metody oraz ich implementacje.

```
// override from Cargo
size_t getPrice() const override;
std::string getName() const override { return name_; }
size_t getAmount() const override { return amount_; }
size_t getBasePrice() const override { return base_price_; }
Cargo& operator+=(size_t amount) override;
Cargo& operator-=(size_t amount) override;
bool operator==(Cargo& cargo) const override;
```

# ZADANIE 3

Dopisz do klasy `Ship`, `Cargo` oraz `Stock` metodę `nextDay()`

- Klasa `Ship`: Metoda powinna odejmować po 1 sztuce monety za każdego członka załogi.
- Klasa `Cargo`: Metoda powinna powodować psucie się towarów.
- Klasa `Stock`: Metoda powinna zmieniać ilość towaru w sklepach.

# ZADANIE 4 (DLA AMBITNYCH)

Spróbuj napisać klasę `Time`, która będzie odpowiadać za zarządzanie czasem w grze.

Klasa ta powinna informować inne klasy, takie jak `Cargo`, `Ship`, `Stock` o upłynięciu każdego dnia.

Poczytaj czym jest wzorzec projektowy `Observer`.

# ZADANIE 5 (DLA AMBITNYCH)

Napisz zaprzyjaźniony operator wypisywania do strumienia

```
friend std::ostream& operator<<(std::ostream& out, const Store& store);
```

Ma on w przystępny sposób wypisywać towar, jaki znajduje się w danym sklepie.

# ZADANIE 6 (DLA AMBITNYCH)

Napisz klasę `Game`, która zarządzać będzie całą rozgrywką.

Dodaj jej jedną metodę publiczną `startGame()`.

Finalnie plik main powinien wyglądać tak:

```cpp
#include "Game.hpp"

constexpr size_t start_money = 1'000;
constexpr size_t game_days = 100;
constexpr size_t final_goal = 2'000;

int main() {
    Game game(start_money, game_days, final_goal);
    game.startGame();

    return 0;
}
```

# CODERS SCHOOL