



Pułapki programowania obiektowego

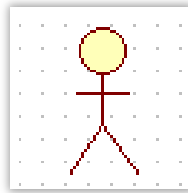
Wersja 2, 29 marca 2011

Adam Sawicki - www.asawicki.info

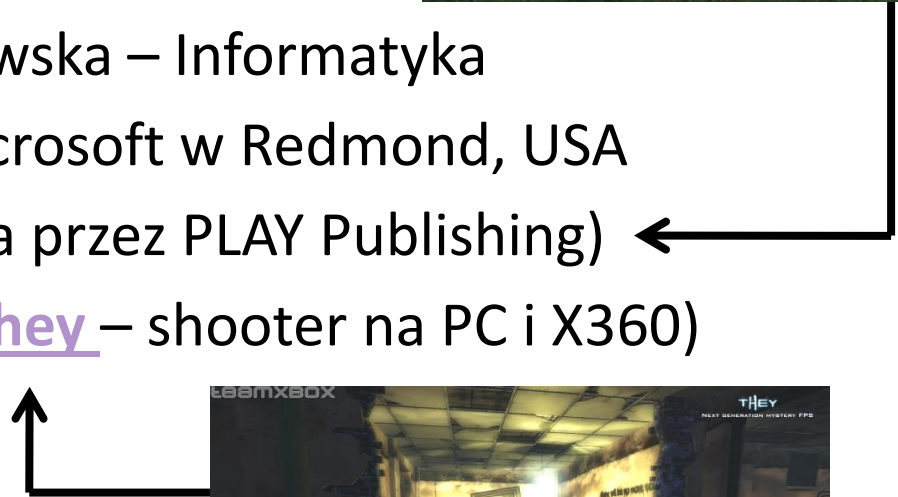
Agenda

- Kim jestem
- „Fanatyzm obiektowy”
 - Projektowanie obiektowe
 - Data-Oriented Design
- „Mania wrapowania”
 - Wrapper – na 3 przykładach
- Generalizacja
- Dziedziczenie
- Enkapsulacja
- Wzorce projektowe
 - Singleton
- Podsumowanie
 - DOD raz jeszcze
- Pytania

Adam Sawicki

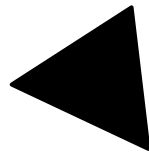


- adam@asawicki.info
- www.asawicki.info
- [@Reg](#)
- Programista
 - Politechnika Częstochowska – Informatyka
 - Praktyki w siedzibie Microsoft w Redmond, USA
 - [AquaFish 2](#) (gra wydana przez PLAY Publishing)
 - Metropolis Software ([They](#) – shooter na PC i X360)
 - Cyfrowy Polsat S.A.



Programowanie obiektowe

- Teoria: program składa się z **klas**
 - łączą **dane** (pola) i **kod** (metody)
 - Modelują byty z dziedziny problemu
 - Są od siebie niezależne
 - Nadają się do ponownego użycia
- Założenia
 - **Enkapsulacja** (hermetyzacja) – udostępnienie interfejsu, ukrycie implementacji
 - **Polimorfizm** i dziedziczenie – używanie obiektu abstrakcyjnego bez rozróżniania konkretnych typów





Fanatyzm obiektowy

Projektowanie obiektowe

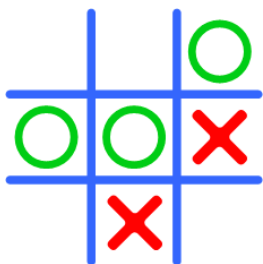
- OOP można rozważać na różnych płaszczyznach:
 - **Ideologiczna** – zasady programowania obiektowego i abstrakcyjne znaczenie jego założeń,
 - **Techniczna** – jak się używa programowania obiektowego w danym języku i jak ono działa,
 - **Praktyczna** – jak używać go w programach dla swojego pożytku, a nie tylko dla zasady.

- Tak jak ze wszystkim, nadmierna radykalność i ślepe poparcie bez myślenia rodzi **fanatyzm**, a to jest złe.

Przykład

Gra w kółko i krzyżyk – podejście obiektowe

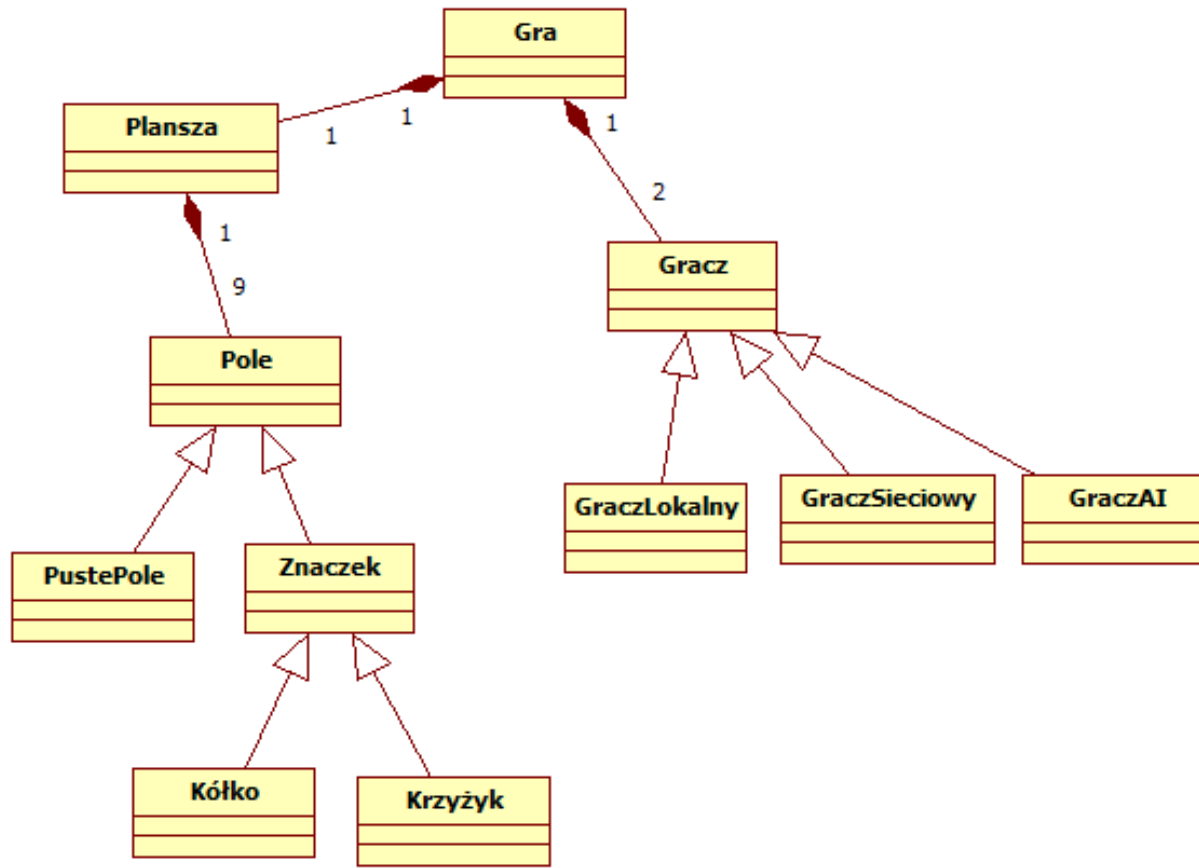
1. Znajdujemy klasy identyfikując rzeczowniki



Krzyżyk
Gracz Plansza
Kółko Gra

Przykład

2. Projektujemy powiązania między klasami



Przykład

3. Implementujemy klasy

- Co umieścić w tych klasach, żeby nie były puste???
- W której klasie umieścić potrzebne dane i kod???

Analysis Paralysis – poszukiwanie idealnego rozwiązania

Przykład

3. Implementujemy klasy

- Co umieścić w tych klasach, żeby nie były puste???
- W której klasie umieścić potrzebne dane i kod???

Analysis Paralysis – poszukiwanie idealnego rozwiązania



Data-Oriented Design

- Alternatywne podejście:

- Data-Oriented Design (DOD)**

- Myśl o strukturze danych w pamięci

- Pytanie: Jakie dane powinna przechowywać gra?

- Tablica 3 x 3 pól
 - Każde pole jest w jednym ze stanów: puste, kółko, krzyżyk
 - Czyj jest teraz ruch: gracza 1 lub 2

```
enum POLE {  
    POLE_PUSTE,  
    POLE_KOLKO,  
    POLE_KRZYZYK,  
};  
POLE Plansza[3][3];  
int CzyjRuch;
```

DOD – przykład

- Następnie pomyśl, co po kolei kod powinien robić z tymi danymi
 - Wykonanie ruchu przez gracza
 - Sprawdzenie, czy ruch jest prawidłowy
 - Wstawienie symbolu do tablicy
 - Sprawdzenie, czy gracz wygrał
 - Rozpoczęcie tury przeciwnego gracza



DOD – przykład

- Następnie pomyśl, co po kolei kod powinien robić z tymi danymi
 - Wykonanie ruchu przez gracza
 - Sprawdzenie, czy ruch jest prawidłowy
 - Wstawienie symbolu do tablicy
 - Sprawdzenie, czy gracz wygrał
 - Rozpoczęcie tury przeciwnego gracza



Projektowanie obiektowe – wnioski

- Podejście obiektowe **nie jest idealne**
- Można nawet stwierdzić, że **nie spełniło swoich założeń**
 - Modelowanie rzeczywistych obiektów? Co modeluje manager, helper, listener, observer, locker i inne -ery?
 - Źle użyte działy przeciwko wydajności, jak też prostocie i czytelności kodu
- Warto je stosować, ale **z rozwagą**
 - Znaj i doceniaj inne możliwości
 - Nie szukaj gotowych „klocków” uciekając od myślenia

Mania wrapowania



Mania wrapowania

- Wielu programistów **czuje potrzebę**, żeby naukę czy wykorzystanie w swoim programie jakiejś biblioteki zacząć od napisania własnej otoczki (wrappera) zamykającego jej interfejs.
 - Robią to niemal **odruchowo** i bez zastanowienia, czy to potrzebne.
- Jakie argumenty za tym stoją?

Przykład 1: FMOD

Biblioteka dźwiękowa FMOD – wczytanie dźwięku WAV

```
FMOD::Sound *Sound;  
FmodSystem->createSound(  
    wavFileName,  
    FMOD_LOOP_OFF | FMOD_2D | MOD_SOFTWARE,  
    0,  
    &Sound);
```

- Jimmy jest przerażony, chce **uproszczyć interfejs**
 - *Te parametry są niepotrzebne*
 - *Chcę mieć mniej parametrów*
 - *Nie chcę sięgać do dokumentacji FMOD – wolę własny interfejs*



Przykład 1: FMOD

Jimmy pisze własną klasę dźwięku, która zamyka FMOD::Sound

```
class CSound {
private:
    FMOD::Sound *Sound;
public:
    void Load(const char *WavFileName);
};

void CSound::Load(const char *WavFileName) {
    FmodSystem->createSound(
        WavFileName,
        FMOD_LOOP_OFF | FMOD_2D | FMOD_SOFTWARE,
        0,
        &Sound);
}
```



Przykład 1: FMOD

- Jimmy potrzebuje możliwości zapętlenia dźwięku
 - Musi dodać parametr Loop
 - Musi zamieniać parametr ze swojej postaci do postaci FMOD

```
void CSound::Load(const char *WavFileName, bool Loop) {  
    FmodSystem->createSound(  
        WavFileName,  
        (Loop ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF) |  
        FMOD_2D | FMOD_SOFTWARE,  
        0,  
        &Sound);  
}
```



Przykład 1: FMOD

- Jimmy odkrywa odtwarzanie w dwie strony i też chce to umożliwić
 - Musi zdefiniować własny enum LOOP_MODE
 - Musi zamieniać swój enum na ten z FMOD

```
enum LOOP_MODE {  
    LOOP_MODE_NONE, LOOP_MODE_NORMAL, LOOP_MODE_BIDI };  
  
void CSound::Load(const char *WavFileName, LOOP_MODE LoopMode) {  
    unsigned LoopFlag = 0;  
    switch (LoopMode) {  
        case LOOP_MODE_NONE:    LoopFlag = FMOD_LOOP_OFF;    break;  
        case LOOP_MODE_NORMAL:  LoopFlag = FMOD_LOOP_NORMAL; break;  
        case LOOP_MODE_BIDI:    LoopFlag = FMOD_LOOP_BIDI;   break;  
    }  
    FmodSystem->createSound(WavFileName,  
        LoopFlag | FMOD_2D | FMOD_SOFTWARE, 0, &Sound);  
}
```



Przykład 1: FMOD

- Wreszcie duża część możliwości FMOD jest przepisana do własnego interfejsu
- Trzeba zrobić do niego własną dokumentację
- Zamiast flag FMOD trzeba pamiętać własne

- Jaka to różnica?
- **Warto było??**

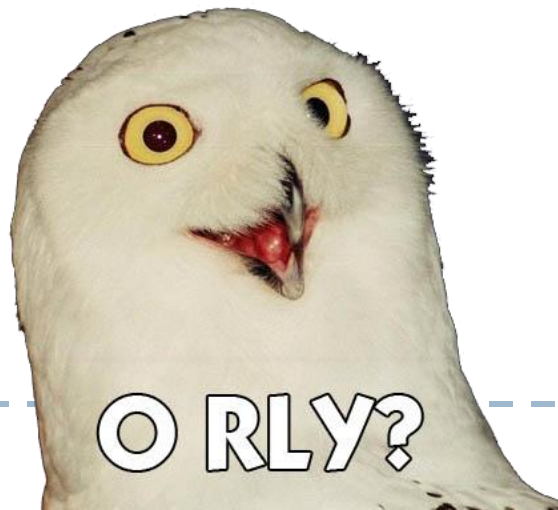
Przykład 2: DirectX

- *Zrobię wrapper na DirectX, to potem będę mógł **wymienić** renderer na OpenGL bez modyfikowania kodu gry.*



Przykład 2: DirectX

- Zrobię wrapper na DirectX, to potem będę mógł **wymienić** renderer na OpenGL bez modyfikowania kodu gry.



Przykład 2: DirectX

- Jakie jest prawdopodobieństwo, że
 - P_1 – skończysz kod swojej gry/silnika
 - P_2 – będziesz potrzebował mieć drugą implementację renderera
 - P_3 – uda się zaimplementować renderer w OpenGL bez większych zmian w jego interfejsie

- $P_1 \cdot P_2 \cdot P_3 \approx 0$

- Znasz już dobrze zarówno DirectX, jak i OpenGL?
 - Jeśli nie $\rightarrow P_3 = 0$

Przykład 3: WinAPI i MFC

- Ktoś w Microsoft:
WinAPI jest niefajne, bo jest strukturalne. Napiszmy obiektową otoczkę.

```
// WinAPI
HDC hdc = GetWindowDC(hwnd);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 100, 100);
ReleaseDC(hdc);
```

```
// MFC
CDC *dc = wnd->GetDC();
dc->MoveTo(0, 0);
dc->LineTo(100, 100);
wnd->ReleaseDC(dc);
```

- Czy to zrobiło aż tak dużą różnicę? **Warto było???**
- Efekt?
 - MFC to tylko cienka otoczka na WinAPI. Nikt go nie lubi.

Wrapper – wnioski

- Jaka wobec tego jest alternatywa?
 - W porę zastanowić się, czy nie wystarczy używać w swoim programie interfejsu danej biblioteki bezpośrednio.
- Warto napisać wrapper, kiedy zapewnia przynajmniej jedno z poniższych:
 - **Znacząco** upraszcza interfejs
 - Dodaje **naprawdę** dużo nowej funkcjonalności
 - Wprowadza **dużo** wyższy poziom abstrakcji
 - **Faktycznie** posiada kilka różnych implementacji

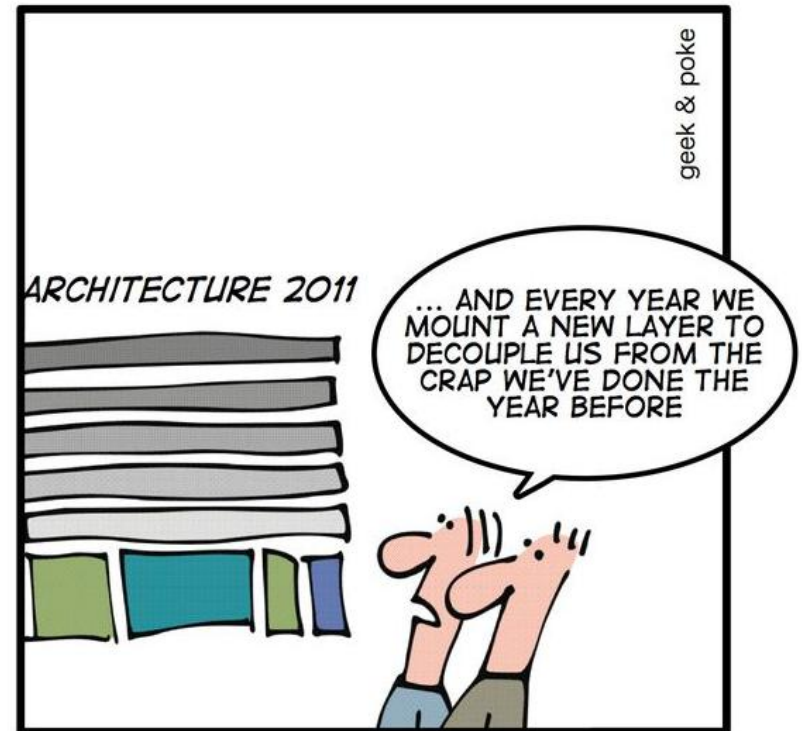
Warstwy

The object-oriented version of "Spaghetti code" is, of course, "Lasagna code". (Too many layers.) – Roberto Waltman

All problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection. – David Wheeler

BEST PRACTICES IN APPLICATION ARCHITECTURE

TODAY: USE LAYERS TO DECOUPLE



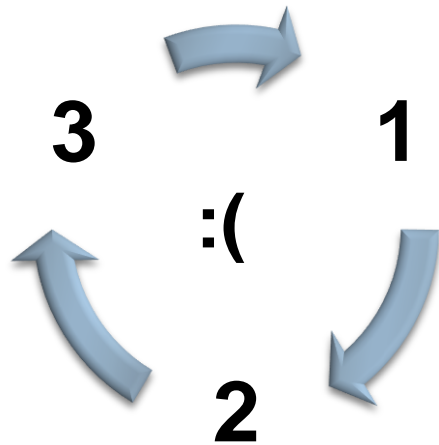
ANNUAL RINGS

Generalizacja

Generalizacja – błędne koło

1. Chcemy, żeby każdy pisany kod był jak najbardziej ogólny i uniwersalny,

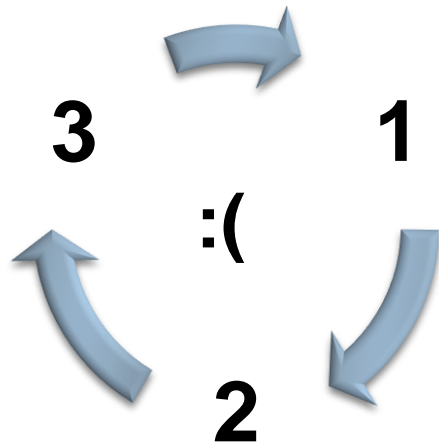
aby nie trzeba było wprowadzać w nim zmian, kiedy zmienią się wymagania.



Generalizacja – błędne koło

2. Unikamy wprowadzania zmian w kodzie,

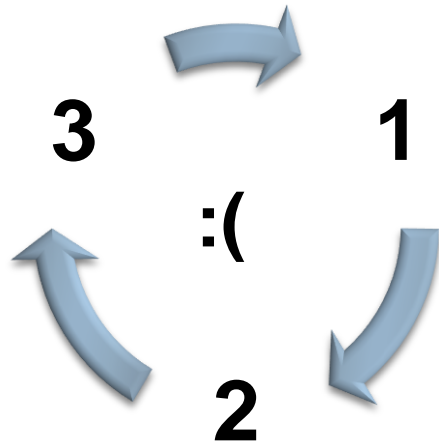
bo każda, nawet miała zmiana, wymaga bardzo dużo pracy.



Generalizacja – błędne koło

3. Każda zmiana w kodzie wymaga bardzo dużo pracy,

ponieważ wszystko piszemy jak najbardziej ogólnie i uniwersalnie.



Generalizacja – błędne koło

■ Rozwiązanie

- Pisz w sposób **prosty, bezpośrednio** to co ma być napisane i nie więcej.
- Nie wyprzedzaj przyszłości. Kiedy pojawi się taka potrzeba, wtedy przerobisz kod na bardziej ogólny.

Dziedziczenie

Dziedziczenie

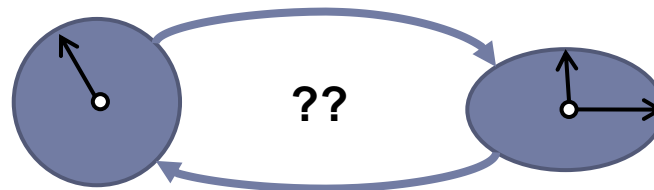
- Teoria
 - Modeluje relację „jest rodzajem”
 - Umożliwia abstrakcję i polimorfizm
 - Dzieli obszerny kod na fragmenty mające część wspólną

Dziedziczenie

- Wady
 - Kod jednej funkcjonalności jest **rozproszony** między wiele klas i plików
 - Tworzy ściśle **powiązania** między klasami
 - Zmiana w jednym miejscu powoduje **nieoczekiwane efekty** w innych miejscach
 - Często **trudno zaprojektować** dobry układ klas, ich pól, metod i zależności między nimi

Dziedziczenie

- Przykład: **okrąg** i **elipsa**
 - Matematyk mówi: okrąg jest rodzajem elipsy, która ma dwa promienie równe!
 - Programista obiektowy mówi: elipsa jest rodzajem okręgu, który dodaje drugi promień!



Dziedziczenie

Rozwiązanie:

- Myśl o **danych**
- Rozważ inne możliwości
 - enum Type
 - Kompozycja (agregacja)
 - Podejście sterowane danymi (data-driven)
 - Architektura komponentowa
- Stosuj dziedziczenie jak najrzadziej, nie jak najczęściej
- Patrz na dziedziczenie jak na mechanizm językowy, nie jak na ideę do modelowania każdej relacji „jest rodzajem”

Enkapsulacja

Enkapsulacja

- Teoria
 - Udostępnia **interfejs**, ukrywa szczegóły **implementacji**
 - Pozwala **zmienić** implementację bez zmian w interfejsie i w innych częściach kodu

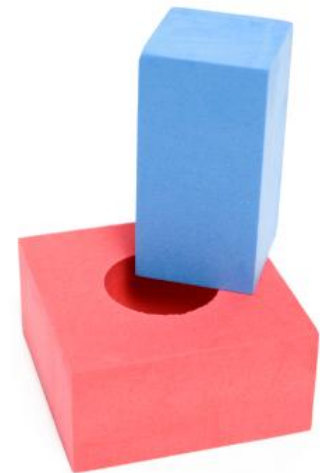
```
class Foo {  
public:  
    int GetValue() { return Value; }  
    void SetValue(int v) { Value = v; }  
private:  
    int Value;  
};
```

Enkapsulacja

■ Problem

- Czym się różni **getter + setter** od uczynienia pola publicznym?
- Czy naprawdę wierzysz, że możesz zmienić implementację tego pola (zmienić jego typ, wyliczać za każdym razem, pobierać z innego obiektu) **bez zmian** w interfejsie i w innych częściach kodu?

```
class Foo {  
public:  
    int GetValue() { return (int)Value; }  
    void SetValue(int v) { Value = (float)v; }  
private:  
    float Value;  
};
```



Enkapsulacja

Rozwiązanie:

- Nie bój się używać **struktur** z publicznymi polami tam, gdzie chodzi o paczkę danych
- Rozważ inne metody dostępu do danych, np. parametry przekazywane podczas inicjalizacji, a potem dostępne **tylko do odczytu**
 - Descriptor, immutability

```
struct FooDesc {
    int Value;
};

class Foo {
public:
    Foo(const FooDesc &params);
    void GetParams(FooDesc &out_params);
};
```



Wzorce projektowe: Singleton

Singleton

■ Teoria

- Tylko jedna instancja klasy
- Dostępna globalnie
- Inicjalizowana przy pierwszym użyciu

```
SoundSystem::GetInstance().PlaySound("Boom!");
```

■ Wady

- Brak jawnej kontroli nad kolejnością inicjalizacji i finalizacji
- Narzut na każde wywołanie
- Problem z bezpieczeństwem wątkowym

■ Czy na pewno system dźwiękowy powinien się inicjalizować w momencie, kiedy chcemy odegrać pierwszy dźwięk?

Singleton

Rozwiązanie:

- Jawnie tworzyć i niszczyć obiekty globalne w określonym miejscu programu

```
g_SoundSystem = new SoundSystem();
```

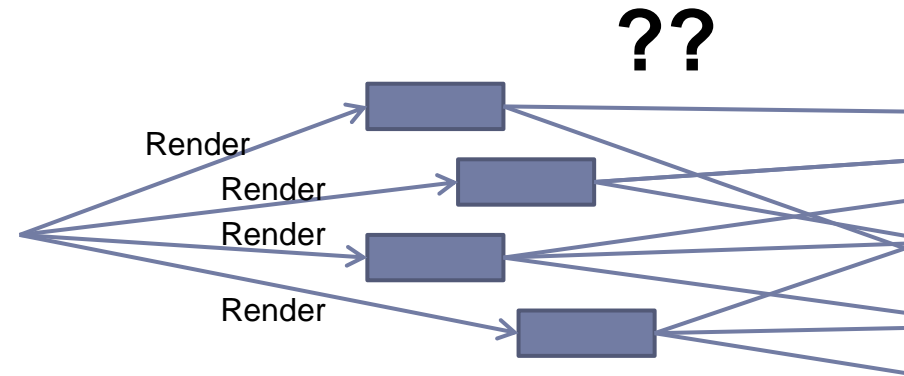
*Every time you make a singleton, God kills a startup.
Two if you think you've made it thread-safe.* – popularna
sentencja z Twittera

Podsumowanie

OOP kontra DOD

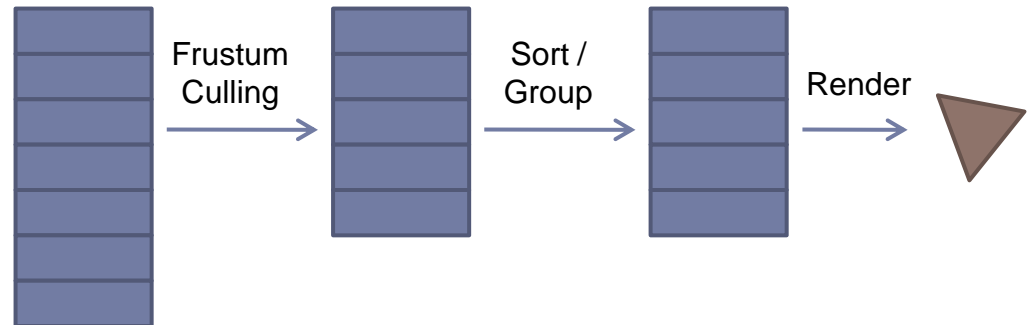
:(

```
class BaseObject {  
private:  
    float3 position;  
public:  
    virtual void Render() = 0;  
};
```



:)

```
struct RenderBatch {  
    float3 position;  
    Mesh *mesh;  
    Material *material;  
};
```



Podsumowanie

- DOD służy raczej do **optymalizacji wydajności**
 - Kod bardziej przyjazny dla pamięci cache
 - Kod łatwiej się zrównolegla
- Jednak wielu odrzuca „przedwczesną optymalizację” nadinterpretując znany cytat Donalda Knutha
- Dlatego tutaj pokazałem, jak krytyczne spojrzenie na OOP może pomóc także w **uproszczeniu kodu**.

Bibliografia

- Fanatyzm obiektowy, Adam Sawicki
 - http://www.asawicki.info/Download/Misc/Fanatyzm_obiektowy.html
- Materiały o Data-Oriented Design
 - http://www.asawicki.info/news_1422.html (lista linków)

Pytania?