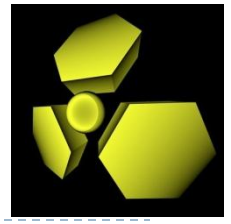
Pułapki programowania obiektowego

Adam Sawicki - www.asawicki.info – 2 stycznia 2011

Wstęp

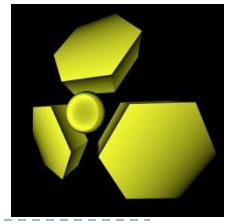


- Programowanie obiektowe (ang. object-oriented programming, OOP) – paradygmat, w którym programy tworzy się pisząc powiązane ze sobą klasy obiektów łączących kod z danymi i reprezentujących pewne byty modelowanego problemu
  - W tym: Enkapsulacja, Abstrakcja, Dziedziczenie, Polimorfizm
  - Do tego: składnia w C++, wzorce projektowe, dobre praktyki, popularne zwyczaje
- Czy zawsze najlepsze?
  - Chciałbym podjąć polemikę…

Fanatyzm obiektowy

Część I

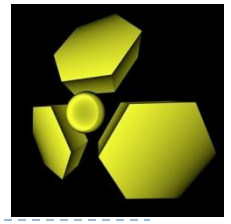
Programowanie obiektowe



- 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

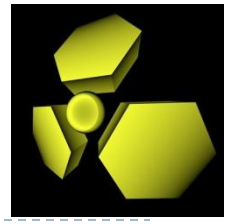
Gracz

Plansza

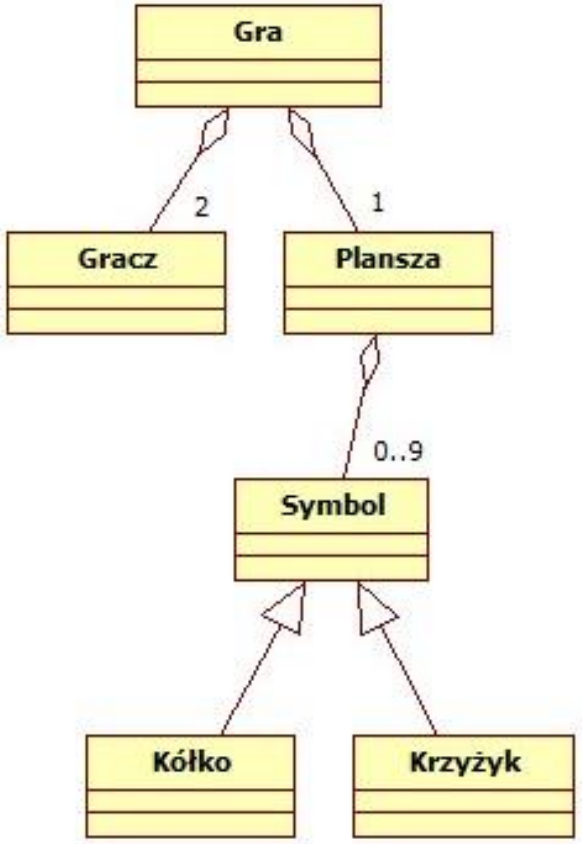
Kółko

Gra

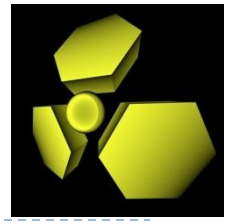
Przykład



2. Projektujemy powiązania między klasami

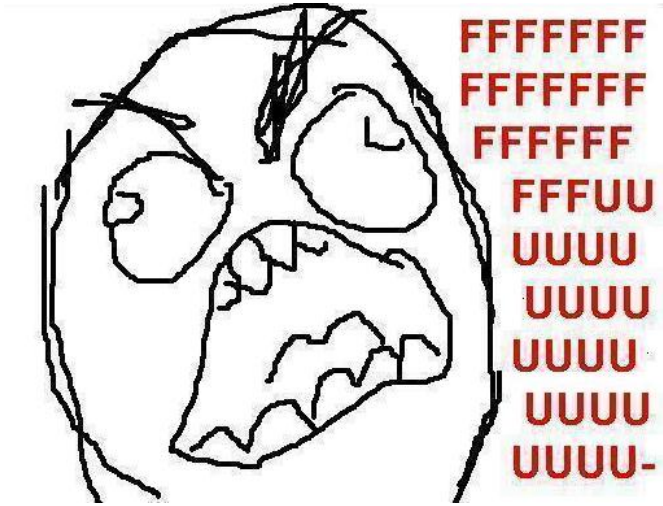


Przykład

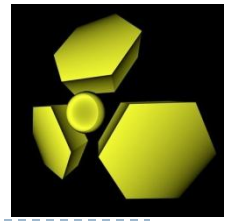


3. Implementujemy klasy

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



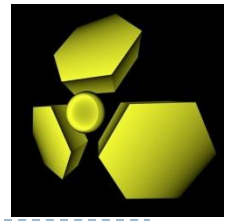
Przykład



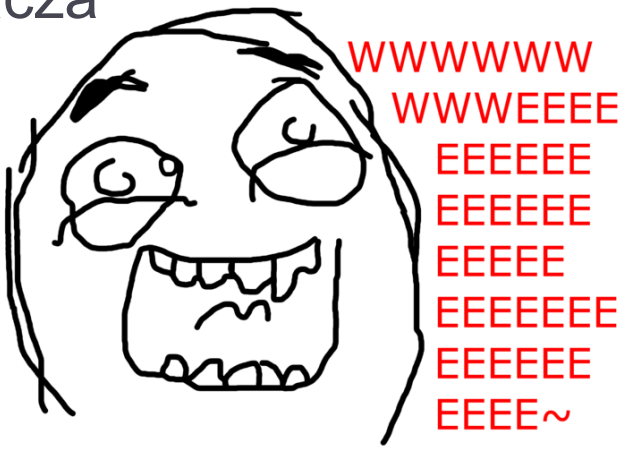
- Alternatywne podejście: Data-Oriented Design (DOD)

- Pytanie: Jakie dane powinna przechowywać gra?
  - Tablica 3 x 3 pól
    ```
    Symbol Plansza[3][3];
    ```
  - Każde pole jest w jednym ze stanów: puste, kółko, krzyżyk
    ```
    enum Symbol { Pusty, Kolko, Krzyzyk };
    ```
  - Czyj jest teraz ruch: gracza 1 lub 2
    ```
    int KtoryGracz;
    ```

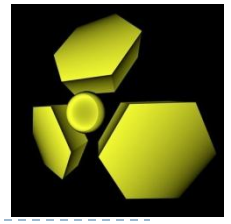
Przykład



- Następnie pomyśleć, 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



Wnioski

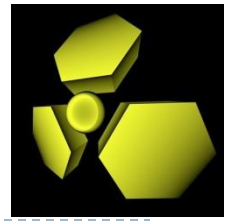


- Programowanie obiektowe nie jest idealne
- Niektórzy twierdzą nawet, że nie spełniło swoich założeń
  - Modelowanie rzeczywistych obiektów? Co modeluje manager, helper, listener, obeserver, locker i inne -ery?
  - Źle użyte działa przeciwko wydajności, jak też prostocie i czytelności kodu
- Warto je stosować, ale z rozwagą
  - Znać i doceniać alternatywne możliwości
  - Nie szukać gotowych „klocków” uciekając od myślenia

Mania wrapowania

Część II

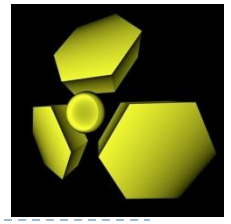
Mania wrapowania



- 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

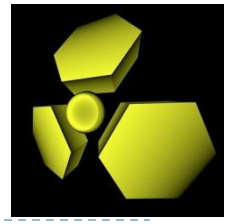


- Biblioteka dźwiękowa FMOD – wczytanie dźwięku WAV

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

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

Przykład 1 – FMOD

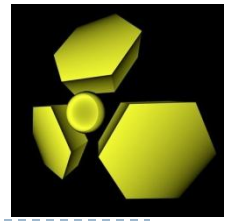


- Jimmy pisze własną klasę dźwięku, która zamyka FMOD::Sound

```
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);
}
```

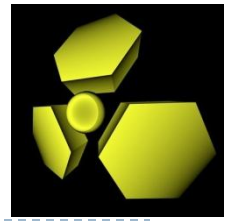
Przykład 1 – FMOD



- 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

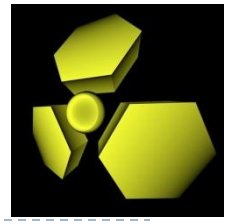


- Jimmy odkrywa odtwarzanie w dwie strony i też chce to zapewnić
  - Musi zdefiniować własny enum „LOOP_MODE”
  - Musi zamieniać swój enum na ten z FMOD

```
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



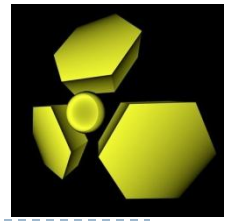
- Wreszcie duża część możliwości FMOD jest przepisana do własnego interfejsu
- Trzeba zrobić do niego własną dokumentację
- Zamiast flag FMOD trzeba pamiętać własne

- 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

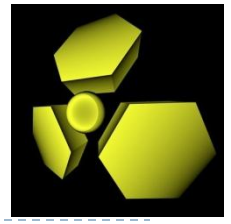


- Jakie jest prawdopodobieństwo, że
  - P_1 – skończysz kod swojej gry/silnika
  - P_2 – będziesz potrzebował mieć drugą implementację renderera
  - P_3 – uda się zaimplementować renderer w OpenGL bez większych zmian w jego interfejsie

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

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

Przykład 3 – WinAPI i MFC



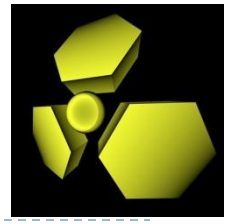
- Ktoś w Microsoft:
  WinAPI jest niefajne, bo jest strukturalne. Napiszmy obiektową otoczkę.

```
// 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.

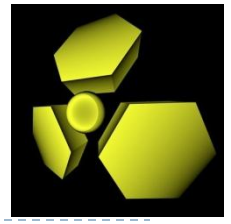
Mania wrapowania



- Jaka wobec tego jest alternatywa?
- Bardzo prosta!
  - W porę zastanowić się, czy nie wystarczy używać w swoim programie interfejsu danej biblioteki bezpośrednio.

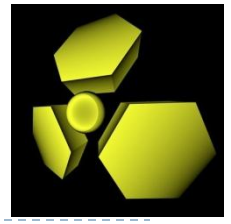
- Napisać wrapper warto, kiedy zapewnia przynajmniej jedno z poniższych:
  - Znacząco upraszcza interfejs
  - Dodaje naprawdę dużo nowej funkcjonalności
  - Wprowadza dużo wyższy poziom abstrakcji
  - Faktycznie posiada kilka różnych implementacji

Podsumowanie



- DOD służy raczej do optymalizacji wydajności:
  - kod bardziej przyjazny dla pamięci cache,
  - kod łatwiejszy do zrównoleglenia, ale
- wielu odrzuca „przedwczesną optymalizację” nadinterpretując znany cytat Donalda Knutha, więc
- pokazałem tutaj, 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
  - Lista wkrótce na mojej stronie domowej…