

kubuntu

LibreOffice®

# Arch. Prog. Nieuprzywilejowanych



| | |
|---|---|
| **autor:** | **Jacek Marcin Jaworski** |
| **pseudonim:** | **Energo Koder Atlant** |
| **pomocnicy autora:** | **BRAK** |
| **miejsce:** | **Pruszcz Gd., Polska** |
| **utworzono:** | **2023-01-27, pią.** |
| **wersja: 2441 z dnia:** | **2025-02-16** |
| **program składu:** | **Libre Office Writer** |
| **sys. op.:** | **Triskel, Kubuntu** |

### Spis treści



## Wprowadzenie

W informie najważniejsze są 3 sprawy: arch., algorytmy i styl programów.

W programowaniu najważniejsze są 2 sprawy: programy i skrypty:

Programy są to języki kompilowane: Asembler, Fortran, C, C++, D, Paskal, Delfi, Rust.

Skrypty są to języki interpretowane: Pyton, Jawa, Jawa Skrypt, PHP, C#, Perl. Specjalną odmianą skryptów są powłoki: Bash, sh, zsh, csh, command.com, PowerShell.

**Prog. koduję gdy mają one krytyczne znaczenie (albo dla mnie, albo dla klienta).**

**Skrypty piszę, gdy nie ma znaczenia ani szybkość wykonania ani jakość kodu. W praktyce skrypty piszę by ułatwić sobie życie i przyspieszyć codzienną pracę.**

Są tylko 3 rodzaje programów: prog. główny, programy uprzywilejowane i programy nieuprzywilejowane:

**Program główny** to rdzeń każdego systemu operacyjnego: zarządza on procesami, wątkami, pamięcią i urządzeniami. Robi to za pomocą sterowników;

**Programy uprzywilejowane** to głównie demony: demony dzielą się na sieciowe i lokalne.

1 Demony sieciowe świadczą usługi sieciowe. Ale nie oznacza to, że tylko zdalne komputery mogą z nich korzystać. Wiele systemów składa się z demonów sieciowych współdziałających z programem nieuprzywilejowanym na tym samym komputerze.

2 Demony lokalne sterują określonymi elementami systemu operacyjnego. Demony lokalne to faktycznie wydzielone elementy, które powinny być w programie głównym, ale są dla bezpieczeństwa lub po prostu dla wygody poza nim;

**Program główny i programy uprzywilejowane działają w trybie uprzywilejowanym, czyli mają pełne uprawnienia (mówi się, że one z komputerem mogą zrobić wszystko).**

Programy nieuprzywilejowane, to programy używane przez użytkownika: są to polecenia konsoli, oraz programy interaktywne działające w trybie tekstowym lub graficznym.

3 **Programy nieuprzywilejowane** działają bez specjalnych uprawnień. Jednak nawet te programy mogą być szkodliwe bo standardowo mają pełen dostęp do katalogu domowego oraz standardowo mają pełen dostęp do globalnej sieci Internet. Skutek tego może być taki, że ściągnięty z sieci „darmowy programik" po uruchomieniu bez najmniejszego problemu może wesoło wysyłać w świat listy plików lub nawet całe pliki wybrane przez zdalnego operatora. Można się przed tym zabezpieczyć własnoręcznie konfigurując piaskownicę i zaporę sieciową.

W tej publikacji będę omawiać tylko architekturę programów nieuprzywilejowanych. Bo od programowania takich "zwykłych programów" należy zaczynać karierę programisty.

Tak jak nie ma jednego przepisu na świetny budynek, tak nie ma jednego przepisu na doskonały prog. Arch. programów komputerowych to wieloaspektowy temat, który postaram się tu zarysować.

**ZASADY PODAWANE W SZKOLE CZY NA UCZELNI UMOŻLIWIAJĄ ZALICZENIA PRZEDMIOTÓW, ALE NIE ZASTĄPIĄ MYŚLENIA W ŻYCIU, BO SĄ JEDYNIE WSKAZÓWKĄ JAK RADZIĆ SOBIE Z PROBLEMAMI.**

**Podobnie należy myśleć o zasadach tutaj opisanych.**

## Sktóty

| | |
|---|---|
| abs. | abstrakcyjny |
| alg. | algorytm |
| aut. | autor |
| d. | dzień |
| dok. | dokument |
| el. | element |
| ew. | ewentualnie |
| f. | funkcja |
| il. | ilość |
| int. uż. | interfejs użytkownika |
| j. | język |
| kat. | katalog |
| kl. | klasa |
| kol. | kolejny |
| l. | liczba |
| o. | obiekt |
| obl. | obliczenie |
| odp. | odpowiedź |
| sys. op. | system operacyjny |
| org. | organizacja |
| p. | punkt |
| pam. op. | pamięć operacyjna (w j. ang. RAM) |
| pub. | publiczny |
| pyt. | pytanie |
| wyr. reg. | wyrażenie regularne |
| roz. | rozdział |
| s. | strona |

# 1 Narzędzia programisty

Podst. narzędzia programisty, to:

1. Dok.: plan proj., dok. funkcjonalna, dok. techniczna, sprawozdanie z wykonania prototypu, dok. użytych prog. i bibl., sprawozdanie z podsumowaniem wykonania wcześniejszych proj. (oraz wcześniejszych wydań w ramach bieżącego proj.);
2. Kod: bibl. kupione, firmowe i proj., prototyp prog., kod prog., wtyczki do prog., konwertery danych, testy aut.;
3. Polecenia konsolowe do przetwarzania tekstu oparte na wyr. reg. to podstawa skryptów jakie na co dzień trzeba tworzyć by pchać proj. na przód;
4. Pliki z zasobami prog., dane testowe;
5. Edytor programisty (ulubiony);
6. Debuger (do uruchamiania ze śledzeniem);
7. Profiler (do wykrywania wycieków pam. i do wykrywania nadmiarowych wywołań f.).

**Zasada Inżynierska: Należy dostosowywać sobie narzędzia z dostępnym kodem źródłowym.**

**Zasada Inżynierska: Wydajność zwiększa się przez automatyzację a nie przez pośpiech.**

**W profesjonalnych proj. niedopuszczalne jest użycie gneratorów formatek (w stylu Qt Designer). Jest tak gdyż w dużych proj. dużą wagę przykłada się do pełnej kontroli nad kodem.**

# 2 Standardowa proc. tworzenia prog. opartego na wtyczkach

**Przygotowania:**

1 Plan realizacji proj.;

2 Szkolenie zespołu[1] (ze sprawozdaniem);

3 Dok. funkcjonalna proj. (wynik analizy zstępującej);

4 Dok. techniczna (wynik analizy wstępującej);

5 Prototyp (ze sprawozdaniem);

**Prototyp to luksus jaki powoduje skok świadomościowy ze sfery domniemywań do sfery wiedzy realnej na temat problemu i proj.**

6 Korekta dok. funkcjonalnej i technicznej.

**Realizacja:**

7 Kodowanie konwerterów danych i ich testów aut.;

8 Kodowanie bibl. proj. i ich testów aut.;

9 Kodowanie logik prog. i ich testów aut.;

10 PPR (Publiczny Pakiet Rozwojowy, w j. ang. SDK) z inter. abs. do tworzenia wtyczek i kodowanie wtyczek i ich testów aut.;

11 Kodowanie okien prog. i ich testy manualne;

12 Pods. proj. (ze sprawozdaniem).

# 3 Analiza zstępująca (funkcjonalna)

Analiza zstępująca mówi co ma robić prog.:

1. Na jakim sprzęcie i pod jakim sys. op. będzie pracował;
2. Jakie dane będzie przyjmował;
3. Jakie dane będzie zwracał;
4. Z czym i jak będzie współpracował;
5. jakie będą zasady jego poprawnego użycia;
6. Jakie będą jego najważniejsze opcje;
7. Jakie będą jego najważniejsze funkcje.
8. Diagramy przypadków użycia (dostarcza je klient[2]);
9. Diagramy sekwencji (dostarcza je klient);
10. Makiety wszystkich okien programu z opisem działania (dostarcza je projektant).

**Analiza zstępująca nic nie mówi o algorytmach ani implementacji programu.**

**Wynikiem analizy zstępującej jest "dokumentacja funkcjonalna".**

# 4 Analiza wstępująca (techniczna)

Analiza wstępująca mówi jak wewnętrznie ma działać prog.:

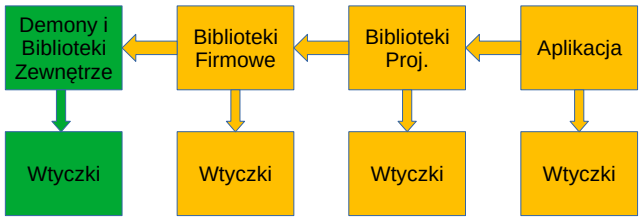
1. Dane wej. (formaty i położenie);
2. Alg. przetwarzania danych w prog.;
3. Dane wyj. (formaty i położenie na dysku);
4. Definiuje jakie będą użyte protokoły komunikacyjne;
5. Definiuje jakie mają być wtyczki (API i położenie na dysku);
6. Definiuje wymagane kl. narzędziowe;
7. Definiuje kl. logiki prog. i ich maszyny stanów;
8. Wskazuje jakie mają być użyte wzorce proj.;
9. Definiuje jak będą wyglądać testy;
10. Definiuje jaki ma być poziom bezp. i jak go osiągnąć.

**Wynikiem analizy wstępującej jest "dokumentacja techniczna".**

# 5 Arch. prog.

## 5.1 Arch. typowego prog. nieuprzywilejowanego opartego na wtyczkach

**Architektura Typowego Prog. Nieuprzywilejowanego**




4 Aplikacja zawiera kl. narzędzi, logiki i okien, natomiast bibl. zawierają jedynie dodatkowe kl. narzędzi. Ta arch. została wymyślona przeze mnie i nazwałem ją wzorcem Narzędzia-Logika-Okna. Tym wzorcem zastępuję bardziej specjalistyczny wzorzec Model-Widok-Kontroler (w j. ang. MVC).

1Jednak należy mieć na uwadze, że biegłe opanowanie nowych narzędzi i procedur zajmuje 2 lata. To wynika z niemieckich doświadczeń wojennych gdy, w latach 1939-41 przestawiali produkcję z rzemieślniczej (manufaktury opartej na stanowiskach) na seryjną (opartą na linie produkcyjne tak jak w amerykańskich fabrykach Forda).
2Nawet gdy jest on klientem wew.

5 Na obrazku są strzałki jednokierunkowe. Obrazuje to zasadę, że kod z wyższych warstw wywołuje kod z warstw niższych. Natomiast warstwy niższe nie wywołują kodu warstw wyższych. Jednak czasem warstwa wyższa musi zaczekać na coś co ma dostarczyć warstwa niższa. W tym celu korzysta się z mechanizmu f. zwrotnej (w j. ang. callback). Polega to na tym, że po prostu jawnie podaje się wsk. do f. jaki ma być wywołany gdy w warstwie niższej pojawią się dane.

**F. zwrotna będzie wywoływana przez inny wątek. Dlatego w wywołaniach zwrotnych konieczna jest sych. muteksami lub semaforami.**

W proj. mamy 3 rodzaje bibl.: dostawcy, firmowe i proj.:

1 Bibl. dostawcy: to bibl. jakie dostajemy z sys. op. oraz jakie kupuje firma;

**Jakość bibl. dostawcy są odbiciem jego kultury technicznej, ekonomicznej i osobistej jego kraju.**

2 Bibl. firmowe: to bibl. jakie zawierają narzędzia własne firmy.

**Jakość bibl. firmowych jest odbiciem kultury technicznej, ekonomicznej i osobistej twojej firmy (są to narzędzia własne firmy).**

Bibl. firmowe mają też separować proj. od świata zewnętrznego. Oto powody:

2.1 Rozwijanie bibl. firmowych jest konieczne z powodu konieczności używania darmowych bibl. czyli gratisów z SZAP. A gratis ma za zadanie psuć umysł naiwnego pasożyta.

2.2 Otoczenie firmy jest niestabilne - nawet gdy firma płaci za narzędzia programisty. Dlatego te bibl. przede wszystkim przezywają wszystkie używane w firmie typy danych w celu ew. ich podmiany, rozszerzania lub przedefiniowania;

2.3 Przestarzałe zew. bibl. w j. C trzeba opakowywać kl. C++ tak by można było ich normalnie używać.

3 Bibl. proj.: Te bibl. wyodrębnia się by w przyszłości przesunąć je do bibl. firmowych.

**Okazuje się, że bibl. firmowe i proj. korzystnie jest linkować statycznie do własnych prog.**

Wynika to z tego, że po instalacji taki prog. jest niewrażliwy na zmiany w bibl. Wtedy spokojnie można je instalować w sys. bibl. w wer. debug i w celu uruchamiania prog. ze śledzeniem. Natomiast gdy te bibl. są współdzielone, to w wer. release nie ma możliwości zajrzenia do o. kl. z bibl. a to duże utrudnienie (wtedy pomaga tylko wypisywanie komunikatów na konsolę lub zrzucanie do plików). Przekompilowanie bibl. współdzielonej w wer. debug jest z kolei niekorzystne z p. widzenia wydajnej pracy z prog. w wcześniejszej wer.

## 5.2 Arch. kat. ze źródłami

### 5.2.1 Model asemblerowy i model C

Obecnie w j. prog. stosuje się 2 modele org. kodu w kat. proj.:

1. Model asemblerowy występuje w j. Asembler i D (oraz w skryptach, ale nimi tu się nie zajmuję). Jego cechą charakterystyczną jest jedność i nierozerwalność deklaracji f. i jej definicji. To powoduje, że konieczne w nim jest udogodnienie w postaci widoczności f. w całym pliku. Z tego powodu:

**W modelu asemblerowym wywołanie f. może poprzedzić jej deklarację-definicję. Mimo, że jest to niedopuszczalne z logicznego p. widzenia.**

2. Model C: Występuje tylko w j. C i C++. Jego cechą jest możliwość rozdzielenia deklaracji i definicji f.: można wydzielać pliki nagłówkowe z deklaracjami f. i pliki źródłowe z definicjami f. Użycie f. nie może poprzedzić jej deklaracji.

**Powrót w j. D do modelu asemblerowego oznacza brak możliwości stosowania tego j. do dużych proj. Wynika to po prostu z braku plików nagłówkowych, co oznacza brak możliwości szybkiego wglądu w zawartość plików źródłowych.**

W obu sposobach organizacji kodu istnieje możliwość tworzenia f. wstawianych. W j. Asembler, C i C++ można je wstawiać makrami. Dodatkowo w C, C++ i D wybrane f. można oznaczać jako inline (jednak mimo takiej dyrektywy kompilator wcale nie musi wstawić kodu tej f. w miejscu wywołania – powody tego typu zachowania są nieznane).

### 5.2.2 Główne kat. z kodami źródłowymi

1. Kod prog.;
2. PPR: „Pub. Pakiet Rozwojowy” (w j. ang. SDK) z interfejsami API do tworzenia wtyczek;
3. Kod wtyczek prog.;
4. Skrypty;
5. Testy aut.;
6. Dane testowe;
7. Zasoby.

## 5.3 Arch. pliku z kodem

Nowsze j. prog. zwykle wprowadzają nowy styl programowania, ale często pozwalają też programować w starszych stylach. Jednak w starszych j. prog. trudno uzyskać efekty jakie wprowadzono w nowych - ogromny wysiłek jaki jest do tego konieczny całkowicie niweczy zysk. Dlatego nonsensem jest kodowanie w j. C w stylu C++.

## 5.3.1 Sekwencyjna

Wszystkie j. prog. umożliwiają programowanie sekwencyjne.

W arch. sekwencyjnej kolejne linie prog. są wykonywane w obrębie jednej f.; po kolei z góry na dół; od pierwszej linii do ostatniej. Skoki wykonuje się wyłącznie w obrębie lokalnych pętli. Tak zaczyna się nauka programowania.

Przykład prog.[3] sekwencyjnego w j. Asembler[4]:

```
; Obliczenie N-tej wart. ciągu Fibonaciego:
format ELF64 executable
entry main

include 'import64.inc'
interpreter '/lib64/ld-linux-x86-64.so.2'
needed 'libc.so.6'
import printf,exit

segment readable executable
main:
   push rcx
   push rdx
   mov rcx, [gIloscIteracji]

; Obl. wart. ciągu Fibonaciego:
; Param rcx: nr wyr. ciągu.
; Wynik rdx: szukana wart.
   cmp rcx, 1
   jne f1
   mov rdx, 1
   ret
f1:
   cmp rcx, 2
   jne f2
   mov rdx, 2
   ret
f2:
   mov rax, 1
   mov rbx, 1
   dec rcx
   dec rcx
f3:
   mov rdx, rax
   add rdx, rbx
   mov rax, rbx
   mov rbx, rdx
   dec rcx
   jnz f3

; Drukowanie wyniku:
; Param rdx: wrart do wypisania na konsolę.
   xor rax, rax  ; Bez xmm.
   mov rdi, gWartCiaguFibonaciego
   mov rsi, [gIloscIteracji]
   call [printf]

   pop rdx
   pop rcx

; Wyj. z prog.
   xor rax, rax
   call [exit]

segment readable writable
   gIloscIteracji: dq 10
   gWartCiaguFibonaciego: db "Wynik po %d iteracjach
wynosi: %d.", 10, 0
```

## 5.3.2 Funkcyjna

J. programowania funkcyjnego jest Fortran. Natomiast j. C, C++ i D umożliwiają m. in. programowanie funkcyjne.

W arch. funkcyjnej z pierwszej f. wyodrębnia się kolejne. Robi się to w celu poprawy przejrzystości oraz w celu ponownego użycia f. Prowadzi to do tworzenia bibli. (linkowanych statycznie lub dynamicznie).

Cechą szczególną prog. funkcyjnego jest całkowita eliminacja zarządzania stanem programu. Wynika to z faktu, że wszystkie dane są przekazywane jako param. f. i przez swoją kompletność stanowią one dokładne określenie stanu prog. Ten schemat powielają niektóre (lepsze?) protokoły sieciowe, np. HTTP, jednak niegdyś popularny FTP bazuje już na jawnych stanach po s. serwera i po s. klienta.

W j. Fortran wszystkie zmienne alokuje się na stosie, czyli nic nie można zaalokować na stercie.

Przykład prog. funkcyjnego w j. D:

```
// Obliczenie silni:
import std.stdio;
import std.conv;
import std.format;
import std.string;
import core.stdc.stdlib;

ulong wczytaj()
{
   writeln("Podaj nr wyrazu ciągu silni jaki chcesz
poznać [zatw. klaw. Enter]: ");
   return to!long(readln().strip());
}

ulong oblicz(ulong n)
{
   writeln(format("Obliczam %d wyr. silni.", n));
   ulong lWynik = 1;
   for(ulong i = 1; i <= n; ++i)
      lWynik *= i;
   return lWynik;
}

void drukuj(ulong s)
{
   writeln(format("Silnia wynosi: %d", s));
}

int main(string[] a)
{
   wczytaj().oblicz().drukuj();
   return EXIT_SUCCESS;
}
```

## 5.3.3 Proceduralna

Klasycznymi j. prog. proceduralnego są j. C i Paskal. Program podzielony jest na f., a jego dane są ujęte w struktury. Struktury najczęściej są oparte o typy proste, czyli o typy wbudowane w j. (rzadziej struktury składają się z innych struktur).

Arch. proceduralna jest podobna do funkcyjnej, jednak nie przekazuje się do f. kompletu danych definiujących stan prog. Zmienna określająca stan prog. może być zm. w strukturze logiki aplikacji lub może być po prostu zm. globalną prog.

**Częsty błąd początkujących programistów prog. proceduralnych i obiektowych to brak jawnej kontroli stanu w aplikacji (brak zm. typu enum z aktualnym stanem), prowadzi to do porażki proj. w raz ze wzrostem jego komplikacji prog.**

Jawnej kontroli stanu na pewno wymaga programowanie wielowątkowe i sieciowe (wiele spotykanych prot. sieć. jest

3Wszystkie prezentowane tu przykłady kodu zostały skompilowane i uruchomione.

4Jest to polski asemebler FASM na proc. AMD64 (umożliwia też kodowanie na proc. Intel od 8080 do Pentium włącznie). UWAGA: aby skompilować kod trzeba skopiować do kat. bieżącego pliki import64.inc i `elf.inc` z kat. `/usr/share/fasm/examples/elfexe/dynamic` .

stanowych mimo, że znacznie komplikuje to demony sieciowe i programy klienckie).

Przykład prog. proceduralnego w j. C:

```
/* Przykład proceduralnego przetwarzania danych: */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Struktury:::::::::::::::::::::::::::::::::::::*/
struct Dane
{
   int cX, cY, cWysokosc;
   double cPaliwo;
   double cLadunek;
   /* Tu zmienne... */
};

/* Funkcje:::::::::::::::::::::::::::::::::::::::*/
struct Dane* wczytajDane()
{
   struct Dane* d = malloc(sizeof(struct Dane));
   memset(d, 0, sizeof(struct Dane));
   /* Tu kod... */
   return d;
}
int sprDane(struct Dane* d)
{
   /* Tu kod... */ return 0;
}
int oblicz(struct Dane* d)
{
   /* Tu kod... */ return 0;
}
int wypiszWyniki(struct Dane* d)
{
   /* Tu kod... */ return 0;
}
int zwolnijPamiecDanych(struct Dane* d)
{
   free(d);
   return 0;
}

int main(int n, char** w)
{
   /* [Pominięte dla czytelności:] Wczytywanie opcji:
         param. lini komend i plików ustawień. */

   struct Dane* lDane = wczytajDane();
   int lWynik = sprDane(lDane);

   if(!lWynik)
      lWynik = oblicz(lDane);

   if(!lWynik)
      lWynik = wypiszWyniki(lDane);

   zwolnijPamiecDanych(lDane);

   if(lWynik)
   {
      fprintf(stderr, "Wystąpił błąd! Nr: %d\n", lWynik);
      exit(lWynik);
   }

   printf("Wykonanie prawidłowe!\n");
   exit(EXIT_SUCCESS);
}
```

## 5.3.4 Obiekty w proceduralnym j. C

W tym miejscu można podpowiedzieć kombinatorom trik pozwalający uzyskać efekt dziedziczenia struktur w j. C. Pomysł polega na kapsułkowaniu rodzica w potomku. Jest to uproszczone dziedziczenie jednobazowe. Działa to tylko dzięki temu, że w j. C odwzorowanie struktur w pam. komp. Jest dokładnie takie jak w deklaracji struktury – nic nie jest dodawane przez kompilator. Schemat kapsułkowania w celu uzyskania efektu dziedziczenia pokazuje poniższy kod:

```
#include <stdio.h>
#include <stdlib.h>

struct StrukturaA
{
   char a;
   int b;
};

struct StrukturaB
{
   struct StrukturaA A;
   long long c;
};

int main(int n, char** p)
{
   struct StrukturaB b;
   b.A.a = 100;
   b.A.b = 101;
   b.c = 102;

   struct StrukturaA* a = (struct StrukturaA*) &b;

   printf("a.a=%d, a.b=%d, b.c=%lld\n", a->a, a->b, b.c);

   return EXIT_SUCCESS;
}
```

Trzeba jednak pamiętać, że C++ pozwala na wielobazowe dziedziczenie (można dziedziczyć po wielu kl. jednocześnie) i wygodne f. wirt (w strukturach C można je emulować używając wsk. do f.).

Ogólnie można powiedzieć, że wszystko co jest możliwe w j. C++ jest też możliwe w j. C. Można dodać, że pierwsze kompilatory C++ były konwerterami kodu do j. C. Z tego co pamiętam tak było w przypadku kompilatora FSF GNU g++.

## 5.3.5 Obiektowa

Klasycznymi j. prog. obiektowego są C++ (C z klasami), D i Delfi (Paskal z klasami). Arch. obiektowa jest rozwinięciem arch. proceduralnej:

W j. C++ obok struktur mogą występować klasy. Klasa od struktury różni się jedynie tym, że domyślnie wszystko jest w niej private, a w strukturze domyślnie wszystko jest public. Czyli to czysta kosmetyka.

6 W odróżnieniu do struktur w j. C, kl. w C++ często składają się ze zmiennych innych klas. Kl. można dziedziczyć w celu implementacji interfejsów (wtyczki) lub w celu rozszerzania lub przedefiniowania zachowania starszych kl.

7 W j. C++ kl. i struktury w oprócz zmiennych mogą mieć f. Warto przyjąć zasadę, że w kl. wszystkie f. pub. są wirt. Natomiast w strukturach wszystkie f. nie są wirt. Wynika to z tego że tylko wtedy można te kl. rozszerzać. Jednak f. wirt. są wolniejsze niż „normalne” (bo wywołuje się je za pośrednictwem tab. f. wirt.). Dlatego aby uzyskać pełną prędkość w strukturach, można by przyjąć, że wszystkie f. pub. nie są wirt. (takimi strukturami powinny być np. kl. QRect, QPoint, QSize itp.).

8 W j. C++ istnieje pojęcie kl. abstrakcyjnej, służą one do definiowania interfejsów. Kl. abs. jest klasa która ma co najmniej jedną deklarację f. wirt. bez jej implementacji. To się przydaje przy tworzeniu pakietów ppr (czyli w j. ang. SDK) do tworzenia wtyczek.

**W C++ każda kl. powinna być deklarowana w osobnym pliku *.h++ i implementowana w osobnym pliku *.c++.**

Uzasadnienie: Czytelność w sensie łatwości analizy proj.

Przykład prog. obiektowego w j. C++:

```
// Przykład obiektowego przetwarzania danych:
#include <iostream>
#include <exception>

// Klasy:::::::::::::::::::::::::::::::::::::::
struct Dane                    // Kl. typu struktura.
{
public:
   int cX = 0, cY = 0, cWysokosc = 0;
   double cPaliwo = 0;
   double cPredkosc = 0;
   double cLadunek = 0;
   /* Tu zmienne... */

public:
   void wczytajDane() { /* Tu kod... */ }
};

class PrzetwarzanieDanych    // Kl. typu usługa.
{
public:
   PrzetwarzanieDanych(const Dane& d)
      : cDane(d) {}

public:
   virtual void przetwarzaj()
   {
      sprDane();
      oblicz();
      wypiszWyniki();
   }

protected:
   void sprDane() { /* Tu kod... */ }
   void oblicz() { /* Tu kod... */ }
   void wypiszWyniki() { /* Tu kod... */ }

protected:
   const Dane& cDane;
};

// Funkcje:::::::::::::::::::::::::::::::::::::
int main(int n, char** w)
{
   try
   {
      // [Pominięte dla czytelności:] Wczytywanie opcji:
      //    param. lini komend i plików ustawień.

      Dane d;
      d.wczytajDane();
      PrzetwarzanieDanych p(d);
      p.przetwarzaj();
   }
   catch(const std::exception& pWyjatek)
   {
      std::cerr << "Wystąpił wyjatek! Treść: "
         << pWyjatek.what() << std::endl;
      exit(EXIT_FAILURE);
   }

   std::cout << "Wykonanie prawidłowe!" << std::endl;
   exit(EXIT_SUCCESS);
}
```

## 5.4 Wtyczki w proj.

Rodzaje wtyczek w proj.:

1. Wtyczki w bibl. narzędzi służą do obsługi różnych formatów plików lub do obsługi różnych protokołów;
2. Wtyczki w Logice prog. służą do obsługi różnych alg. pracy. Są ich 2 rodz.:

   2.1. Reagujące na zmianę stanu logiki;

   2.2. F. spec.: jawnie wywoływane przez logikę.

Generalnie największą elastyczność dają f. reagujące na zmianę stanu logiki. Jednak opierając się na samych zmianach stanu logiki nie zawsze jest możliwe zachowanie spójności zachowania aplikacji, stąd f. spec. wywoływane dodatkowo między zmianami stanów (niekiedy użycie f. spec. powoduje radykalne uproszczenie kodowania wtyczki).

3. Wtyczki w oknach prog. standardowo dodają nowe el. interfejsu co wzbogaca funkcjonalność prog.

Zasady tworzenia wtyczek:

1. Wtyczki w bibl. narzędziowych działają pojedynczo (np. do ładowania pliku w danym formacie). Często te wtyczki są ładowane aut. na podstawie kontekstu wynikającego z bieżącej pracy prog., np. wybór pliku o określonym roz.;
2. By użyć równolegle 2 wtyczek z bibl. narzędziowej konieczne jest utworzenie dwóch o. kl. narzędziowej. Tak jest np. w przypadku otwarcia 2 różnych plików lub w przypadku jednoczesnej obsługi 2 różnych protokołów w jednym celu, np. do obsługi bazy danych.
3. Należy jawnie zdefiniować kol. ładowania wtyczek. Ta kol. określa też kol. wywołań f. z tych wtyczek. Bez znajomości kol. wywołań też nie było by wiadomo co się dzieje w prog. Oznacza to, że:
4. Wewnętrznie wtyczka powinna działać synchronicznie i powinna powodować synchroniczne skutki w prog. To zabezpiecza przed chaosem który powstaje gdy używa się darzeń do oddziaływania na prog.

   Jedyną asynchroniczną akcją jaka jest dopuszczalna w przypadku wtyczek to zdarzenie w prog. które po kolei wyzwala wtyczki reagujące na tą akcję.

**Można dodać, że coś takiego jak plik json we wtyczkach Qt jest kompletnie nie na miejscu, bo on służy do określenia jakie wtyczki są wymagane przez daną wtyczkę.**

## 5.5 Prog. konsolowe wywoływane przez prog.

Są powody by używać poleceń konsolowych w prog.:

1. Są łatwiejsze do zrozumienia;
2. Mają lepszą dokumentację;
3. Są mniej kłopotliwe;
4. Są bardziej żywotne;
5. Łatwo je testować;
6. Są b. szybkie gdy dłużej pracują niż wynosi czas ich uruchomienia.

## 5.6 Przenośność programu

Przenośność też wpływa na arch. prog. Sam kod C, C++ i D jest w pełni przenośny. Jednak trzeba go dopasowywać do różnych bibl. w różnych sys. op. Oto wskazówki jak dbać o przenośność kodu:

1. Izolacja kodu: przezywanie typów i klas;
2. Tworzenie interfejsów obiektowych (wzorzec proj. konwerter) do interfejsów w czystym C;
3. Wydzielanie plików *.uniks.c++, *.mak.c++ i *.okna.c++. W tych plikach należy umieszczać wszystkie f. w których pojawia się nieprzenośny kod. Natomiast skrypt CMake warunkowo włącza te pliki w zależności od sys. op. na jaki się kompiluje prog.
4. Dyrektywy warunkowe: #ifdef Q_OS_UNIX, #ifdef Q_OS_MACOS i #ifdef Q_OS_WIN są zakazane, bo robią jatkę w programie (a tym samym również w głowie programisty)!

## 5.7 Logika w programie

Logika w programie musi:

1. Obsługiwać stany aplikacji i na ich podst. kontr. jej działanie;
2. Ładować i zapisywać opcje (pliki INI, linia komend, okno z parametrami);
3. Ładować wtyczki prog.;
4. Tworzyć i niszczyć okna prog.;
5. Spr. niezmienniki;
6. Łapać i obsługiwać wyjątki (rzucane z logik i z narzędzi).

## 5.8 Kontrola stanu Logiki

Typowo stan prog. można kontrolować na 2 sposoby:

1 Bezstanowo: ma dwie odmiany:

1.1 Prog. bazujące na kontekście: w prostych sekwencyjnych prog. z niewielką liczbą zmiennych jest oczywiste co się dzieje. Dlatego "jakoś to działa" bez specjalnego myślenia o stanach prog.

1.2 Prog. funkcyjne: Wywoływana f. dostaje wszystkie potrzebne dane do jej działania (nie używa ona żadnych zmiennych globalnych, ani nie używa zm. kl. do której należy).

2 Stanowo: Maszyna stanów to po prostu enum z jawnie zdefiniowaną listą możliwych stanów.

Zasady projektowania maszyny stanów:

1. Maszyny stanów należy projektować wspierając się programami do wizualizacji takimi jak dot z pakietu Graphiz.
2. W enum nie należy używać heksów do definiowania podstanów. Podstany należy definiować jako odrębne maszyny stanów.

Czyli zamiast:

```
enum Stan { oczekiwanie = 0, start=1, platnosc=2,
wysyłanie=4, pytanieOPodpis=8,  drukowanieParagonu=16,
drukowaniePotwierdzenia=32, ...};
```

należy zdefiniować 2 stany:

```
enum StanTermianala { oczekiwanie, start, platnosc,
raportDobowy, ...};

enum StanPlatnosci { oczekiwanie, start, wysyłanie
pytanieOPodpis, drukowanieParagonu,
drukowaniePotwierdzenia, ...};
```

3. Należy zakodować f. stanDoNapisu() w calu wypisywania czytelnego komunikatu o próbie nielegalnej zmiany stanu (co oznacza awaryjne zamknięcie prog.).

Działanie f. zmiany stanu:

1. Spr. czy przejście do innego stanu jest dopuszczalne?
2. Zmienia stan na nowy;
3. Reaguje na zmianę stanu;
4. Informuje wtyczki (i inne obiekty) o zmianie stanu.

## 5.9 Spr. niezmienników

Niezmienniki są to warunki jakie muszą zawsze być spełnione aby prog. działał prawidłowo. Za prawidłowe działanie prog. odpowiada jego logika więc tylko w logice należy spr. niezmienniki. Niezmienniki należy spr. na końcu konstruktora i na końcu każdej f. publicznej z wyj. destruktora;

## 5.10 Obsługa wyjątków w Logice

Wyjątki należy łapać w prog. we wszystkich f. pub. kl. logiki. Wynika to z faktu, że tylko na poziomie logiki wiadomo co z tymi wyjątkami robić.

W celu automatyzacji typowych wyjątków które kończą się zamknięciem prog. należy zrobić makro w którym:

1. Komunikat jest zapisywany do pliku z listą błędów (zawsze);
2. Komunikat jest wypisywany na konsolę (zawsze);
3. Komunikat jest wyświetlany w okienku gdy jest to prog. graf. (należy to spr. makrem takim jak #ifdef QT_GUI_LIB).

Takie makro automatyzujące łapanie wyjątków możliwe jest jedynie w prog. (a nie w bibl) – wynika to z warunkowej kompilacji #ifdef QT_GUI_LIB, której nie można zastosować bez rekompilacji biblioteki). Jest to argument by logiki były tylko i wyłącznie w prog. (a nie w bibl.).

## 5.11 Sposób interakcji z prog.

Do podstawowych sposobów interakcji z programem należą:

1. Zadania wsadowe: To typowo seria poleceń połączonych rurkami w którym pierwsze polecenie przyjmuje dane wej. a ostatnie wypisuje wynik na konsolę. Interakcja z użytkownikiem ogranicza się do podania danych wej., wywołania i odbierania wyniku w formie tekstowej.
2. Czarodzieje (w j. ang. wizzards) tekstowe lub graficzne: Zadają kol. pyt. i użytkownik musi na nie odp.;
3. Programy okienkowe tekstowe lub graficzne: Interakcja odbywa się za pomocą zwykłych okien;
4. Programy zajmujące cały ekran: Zwykle tak działają gry komputerowe. To po prostu okno na cały ekran bez możliwości jego nagłego zamknięcia.

9 Od s. technicznej można wyróżnić prog.:

1. Bez interakcji (wsadowe) – wypisują wyniki jako tekst na konsolę;
2. Tekstowe oparte o f. getchar(): To f. bibl. j. C. Umożliwia ona prostą interakcję w terminalu polegającą na wywołaniu w pętli f. getchar() w celu pobrania znaku lub linii tekstu od użytkownika. Normalnie użycie f. getchar() wstrzymuje pracę programu na czas wpisywania znaków.
3. Pętla zdarzeń (np. bibl. Qt): Najczęściej stosowana jest w prog. graficznych zwanych "programami użytkowymi". Jej działanie polega na monitorowaniu deskryptorów wybranych urządzeń (np. klawiatury, myszy, karty sieciowej, itd.) w celu pobrania od nich danych gdy się one pojawią. Wtedy następuje propagacja takiego zdarzenia po obiektach w oknie prog. mogących je odbierać. W przypadku myszy będzie to kontrolka interaktywna która zostanie aktywowana (dostanie ognisko - w j. ang. focus). W przypadku klawiatury będzie to aktywna kontrolka interaktywna (mająca ognisko).
4. Pętla gry (np. bibl. SDL i NCurses: Najczęściej stosowana w grach komputerowych oraz w prog. czasu rzeczywistego. W pętli gry oblicza się położenie obiektów na ekranie oraz fizykę wirtualnego świata (kolizje i uszkodzenia). Mierzy się czas przerysowania każdej klatki w celu przewidzenia czasu koniecznego do przerysowania kolejnej klatki, a to jest konieczne by wyliczyć stan obiektów w grze w kolejnej klatce (o ile się ruszyły postacie i jakie zaszły zjawiska fizyczne).

   Pętla gry charakteryzuje systemy czasu rzeczywistego, które muszą być przewidywalne w sensie czasowym. Pętla zdarzeń, nie spełnia tych wymagań, bo nikt nie gwarantuje kiedy zdarzenie będzie obsłużone.

**W źle napisanych grach nawet wejście w menu oznacza 100% obciążenia procesora. Dlaczego tak jest? Bo w grach nawet menu jest rysowane przez silnik gry. I jak komp. Jest zbyt wolny do danej gry, to menu też przycina i są slajdy.**

**Mimo, że na sys. Linuks i Windows są biblioteki z pętlą gry to jednak te sys. op. nie są systemami czasu rzeczywistego. Dlatego gry na nich działają wolno (tak samo zresztą wszystkie inne uruchamiane na nich prog.). Wynika to z faktu pisania powolnych prog. - to dlatego od końca lat 90. XXw. komputery działają tak samo wolno (mimo, że teoretycznie od tamtych czasów kompy i sieci znacznie przyspieszyły). Szybkie sys. op. i szybkie prog. koduje się SZAP w tzw. tajnych czarnych projektach.**

Częstym błędem początkujących programistów próbujących używać pętli zdarzeń lub z pętli gry jest sytuacja gdy kontrolka graf. jednocześnie służy do wyświetlania i do wprowadzania danych. Wtedy jest ona jednocześnie kontrolowana przez użytkownika oraz przez urządzenie współpracujące z programem - to jest sytuacja konfliktowa. Dlatego zawsze należy używać osobnych kontrolek do wyświetlania i do ustawiania wart. (ta druga może pojawiać się tylko w razie potrzeby) - mimo, że jest to nieeleganckie, ale ma tą zaletę, że działa normalnie.

## 5.12 Sposób przetwarzania danych w prog.

Niezależnie od sposobu przetwarzania mam nast. zasady:

1. Używam grup wyspecjalizowanych prog., które współdziałają ze sobą w celu realizacji moich zad.;
2. Prog. narzędziowy koduję tylko wtedy gdy muszę.
3. Nigdy nie tworzymy monstrualnych omnibusów, które mają zastąpić wszystkie inne prog. (syndrom Qt Creator).

### 5.12.1 Przetwarzanie strumieniowe danych tekstowych

Pobiera się dane z stand. wej. (lub z pliku) i przetwarza. Wynik zwykle wypisuje się na stand. wyj. To przetwarzanie występuje w dwóch odmianach:

1. Linijkowe: przetwarzanie odbywa się linia po linii;
2. Blokowe: wczytuje się całość i przetwarza na raz (hurtem). Ten model stosuje się przy wieloliniowych wyr. reg. Oczywiście, w porównaniu do modelu linijkowego, ten model ma dużo większe wymagania pamięciowe. Dlatego należy go stosować w ostateczności - najlepiej gdy użytkownik sam o tym zadecyduje wybierając taką opcję.

**Zwraca uwagę nacisk na przetwarzanie linijkowe w poleceniach konsolowych sys. Linuks GNU. Jednak jest to tylko uciążliwa konwencja, gdyż są polecenia które nie mogą jej wypełniać - np. sort - więc również sed i grep mogłyby mieć opcjonalny tryb przetwarzania blokowego (odpowiednik multiline z normalnych wyr. reg.).**

### 5.12.2 Praca z plikami ładowanymi w całości do pam. op.

Są to wszelkiego rodzaju edytory plików w ściśle zdefiniowanych formatach: dźwiękowe, obrazy 2D i 3D, filmy, schematy el., lub arch. i wiele innych.

### 5.12.3 Przetwarzanie bazodanowe

Opiera się na pracy z bazą danych (która w szczególności może być silnikiem SQL, który w szczególności może być na serwerze sieciowym). W tym modelu ważne jest by nie dać się ogłupić specyficznemu silnikowi bazy danych. Dlatego należy specyficzne interfejsy bazodanowe przykrywać własnymi kl. pośredniczącymi by móc w przyszłości łatwo podmienić jeden silnik bazodanowy na inny.

### 5.12.4 Przetwarzanie sieciowe

Najczęściej dotyczy synchronizacji danych w bazach. Takie zadania należy realizować w wątkach wg wzorca proj. Producent-Konsument. Zarówno po stronie aplikacji jak i po stronie demona sieciowego.

### 5.12.5 Przetwarzanie wielowątkowe

Gdy mowa o wątkach zwykle mówi się o muteksach i semafoarach - czyli o tym jak ważne jest chronienie współdzielonych zasobów. Jednak można unikać muteksów i semaforów kopiując dla każdego wątku komplet danych roboczych. W prog. wielowątkowym trzeba jeszcze pamiętać o tym, że:

**Podstawą jakiejkolwiek pracy w wątkach jest maszyna stanów jaka definiuje nam gdzie jesteśmy, kiedy i gdzie się udamy.**

## 5.13 Sposób wykrywania błędów

Generalnie można wyróżnić nast. etapy wykrywania błędów przez programistę:

1. Podczas przeglądu kodu:

   Każdą f. zmienianą przez siebie przejrzyj 2x (od góry do dołu).

   Każdą f. zmienianą przez innych przejrzyj 3x (od góry do dołu).

   Każde włączenie kodu do gałęzi głównej powinny zaakceptować co najmniej 2 osoby[5].

2. Automaty do spr. popr. kodu należy stosować wszędzie gdzie to możliwe. Nie musi być to SI!;
3. W kl. logiki w f. pub. spr. niezmienniki zgodnie z roz. Spr. niezmienników;
4. W kl. przyjmujących dane z zew. stosujemy prog. kontraktowe. Są to kl.: opcje (pliki ini, param. linii komend), pobieranie danych przez int. uż., wczytywanie danych z dysku oraz pobieranie danych z sieci.

**Niezmienniki kl. różnią się od prog. kontraktowego tym, że niezmiennik musi być spełniony w każdej sytuacji (na wyjściu z każdej f. pub. danej kl.) natomiast testy w prog. kontraktowym związane są ściśle z daną f. (z formatem wczytywanych lub/i zwracanych danych).**

5. Podczas testów jednostkowych:

**Testy jednostkowe zwykle zajmują drugie tyle kodu co sam prog. Dlatego na kodowanie tych testów należy przeznaczać co najmniej tyle czasu co na kodowanie samego prog.**

6. Testy manualne:

**Gdy kodujesz coś nowego i już wszystko gra, wymyśl i zrób jeszcze 2 dodatkowe testy manualne.**

**Programowanie sterowane przez testy należy odrzucić gdyż jego istotą jest odwrócenie procesu twórczego i zaczęcie go od końca, czyli od testów. Jest to kolejna podpucha z SZAP.**

# 6 Styl prog.

## 6.1 Izolacja proj. od świata zewnętrznego

W głowie ojca dyrektora wszystko się może zdarzyć, podobnie jak wszystko może się zdarzyć w głowach bankierów co płacą za rozwój darmowego oprogramowania. Dlatego zgodnie z Ustawą o Ochronie Zdrowia Psychicznego trzeba się zabezpieczać przed ich sabotażem. W tym celu należy przestrzegać tych zasad:

1 Zachowanie dyskrecji:

1.1 Ze światem zew. komunikuje się wył. kiero. proj.;

1.2 Klienta nie wtajemnicza się w szczegóły stosowanych rozw. tech.;

5W dok. Biblia Tysiąclecia, copyright Wydawnictwo Pallottinum, PISMO-SW 3.0 BETA (26 lutego 2002 roku), copyright 1994-2002 by Piotr Kłosowski klosowski@iele.polsl.gliwice.pl: Pwt 17,06: "Na słowo dwu lub trzech świadków skaże się na śmierć; nie wyda się wyroku na słowo jednego świadka.". Ten cytat znaczy tyle: **Żadna ważna sprawa nie będzie oparta na mowie jednego świadka. Dlatego jak tylko jeden będzie mówił, że zachowałeś się w porządku, to uczciwy sędzia mu nie uwierzy.**

1.3 O postępach prac wie wyłącznie zespół i klient.

2 Praca offline:

2.1 Zespół pracuje w sieci lokalnej odciętej od Internetu;

2.2 W sieci lokalnej jest repo z lustrzanym serwerem pakietów używanej dystrybucji (wszyscy muszą pracować na tym samym distro w tej samej wer.);

2.3 W razie potrzeby zespół wyszukuje info w sieci Internet za pomocą sprytnych tel.;

2.4 Dane do sieci Internet wysyła się kopiując dane na patyki USB. Mimo, że krajowymi siłami niewiele więcej można zrobić, to należy pamiętać, że:

**Odnośnie patyków USB konieczna jest wzmianka, że znany jest przypadek przenoszenia za ich pomocą wirusów. To właśnie za pomocą zhakowanych irańskich patyków USB Żydzi wprowadzili wirusy do wydzielonej irańskiej sieci komputerowej (prawdopodobnie w elektrowni jądrowej). Wirusy te zmieniły kod sterujący wirówek do do wzbogacania uranu (miał on posłużyć do stworzenia irańskiej bomby atomowej) i w efekcie wirówki te uległy uszkodzeniu lub zniszczeniu.**

3 Izolacja kodu:

3.1 Należy ograniczać il. zew. bibl.;

3.2 Wszystkie typy proste należy przezwać własnymi. Są ku temu co najmniej 2 powody: typy te zmieniają się między kompilatorami i sys. op., może zaistnieć kiedyś konieczność zmiany typu na inny.

3.3 W razie stosowania zew. bibl. kl. należy je przezywać aliasami (oczywiście tylko te kl. których się używa). Robi się tak by w razie potrzeby można było taki alias zastąpić własną, ROZSZERZONĄ klasą bez ruszania jego wszystkich wystąpień w kodzie.

3.4 W razie stosowanie zew. bibl. f. w j. C należy je opakowywać własnymi kl. w celu uproszczenia ich użycia i możliwości wymiany danej bibl. w j. C (bez zmiany reszty kodu).

3.5 Należy się bronić przed j. SQL stosując go w minimalnym stopniu: do serializacji obiektów i do ich wyszukiwania w bazie. Nie należy stosować SQL do żadnego przetwarzania danych, bo to j. skryptowy i ma nienormalną składnię (psuje umysł).

## 6.2 Przetwarzanie etapowe (zamiast matactw)

W ks. [c++podróż] na 210s. Jest taki przykład kodu (nie spr. czy się kompiluje, bo na razie nie mam kompilatora C++20):

```
void user(forward_range auto& r)
{
   int count = 0;
   for (int x : r)
      if (x %2) {
         cout << x << ' ';
         if (++count == 3) return;
      }
}
```

Ten kod jest złą praktyką, bo to matactwo, bo mimo że są w nim 3 proste etapy przetwarzania, to są one zaplątane w jedną pętlę for. Ten przykład powinien wyglądać tak (też nie spr.):

```
void user(forward_range auto& r)
{
   // Filtrowanie nieparzystych
   vector<int> lResult;
   for(int x : r)
      if(x % 2)
         lResult << x;

   // Wybranie pierwszych 3:
   lResult = copy(lResult.begin(), lResult.begin() + 3);

   // Wyświetlenie wyniku:
   for(int x : lResult)
      cout << x << ' ';
}
```

A co z efektywnością? Oczywiście jest ona istotna, jednak czytelność kodu jest ważniejsza. Po za tym optymalizacja na poziomie kodu jest drugorzędna w porównaniu do optymalizacji na poziomie algorytmów. Tak więc warto utrzymać czytelność 99% kodu (dla dobra własnego i kolegów po fachu) i optymalizować tylko to co naprawdę konieczne.

## 6.3 Projektowanie i kodowanie wg optymistycznego scenariusza

Projektowanie prog. (w dok. funkcjonalnej) należy prowadzić wg prostej zasady: realizujemy "optymistyczny scenariusz" w którym wszystko się udaje. Proj. wg optymistycznego scenariusza jest prosty i naturalny, bo stanowi logiczną ścieżkę od startu do zakończenia.

Jak się okazuje można również kodować wg tej zasady. Ma to tą zaletę, że kod jest wiernym odbiciem proj. - czyli jest zrozumiały dla każdego kto poznał dok. funkcjonalną.

Kodowanie wg optymistycznego scenariusza jest trochę niezwykłe: należy kodować z minimalną liczbą zagnieżdżeń. Instrukcje "if" dotyczą przede wszystkim spr. czy wystąpił błąd. Jednak mimo, że takie kodowanie jest logiczne z projektowego p. widzenia, to jednak wprowadza konieczność znajomości kontekstu w jakim występuje dana linia kodu - może to być utrudnieniem dla recenzentów i dla następców.

Programując optymistyczny scenariusz po wykryciu błędu po prostu przerywamy optymistyczny scenariusz. W kodzie oznacza to zwrócenie kodu błędu (w j. Asembler i C) lub rzucenie wyjątku (w j. C++ i D).

Aby to wyjaśnić przedstawię 2 f. w j. C jedną zgodną z optymistycznym scenariuszem, a drugą w stylu "byle jak" (kod skompilowany i uruchomiony ale okrojony dla potrzeb publikacji):

```
[...]
```

```
enum KodBledu
{
   eSukces,
   eDaneWe,
   eZuzyciePaliwa,
   eDrukowanie
};

[...]

int funkcjaOptymistycznyScenariusz(struct Dane* d)
{
   if(sprDaneWe(d) != eSukces)
   {
      fprintf(stderr, "%s", "Błąd danych wej.!\n");
      return eDaneWe;
   }

   if(obliczZuzyciePaliwa(d) != eSukces)
   {
      fprintf(stderr, "%s", "Błąd zużycia paliwa!\n");
      return eZuzyciePaliwa;
   }

   if(drukuj(d) != eSukces)
   {
      fprintf(stderr, "%s", "Błąd drukowania!\n");
      return eDrukowanie;
   }

   return eSukces;
}

int f_byle_jak(struct Dane* d)
{
   if(sprDaneWe(d) != eSukces)
   {
      fprintf(stderr, "%s", "Błąd danych we.!\n");
      return eDaneWe;
   }
   else
   {
      if(obliczZuzyciePaliwa(d) != eSukces)
      {
         fprintf(stderr, "%s", "Błąd zużycia paliwa!\n");
         return eZuzyciePaliwa;
      }
      else
      {
         if(drukuj(d) != eSukces)
         {
            fprintf(stderr, "%s", "Błąd drukowania!\n");
            return eDrukowanie;
         }
         else
            return eSukces;
      }
   }
}

[...]
```

## 6.4 Sposób prezentacji wyników

Dla porządku wymienię tu sposoby prezentacji danych programu:

1. Wyjście na konsolę: typowy sposób działania poleceń linii komend. Mimo, że jest to bardzo prosta prezentacja wyniku działania, to jest ona nadal bardzo użyteczna. Sprawdza się ona równie dobrze w codziennej pracy programisty jak i w warunkach domowych przy przetwarzaniu plików tekstowych.

**W sys. Windows programy graf. nie mogą pisać na konsolę - przy pierwszej próbie pisania na konsolę Windows ubije ten prog. W sys. Linuks nie ma tego ograniczenia.**

2. Interfejs tekstowy: ma 2 odmiany: linijkowy i pełnoekranowy.
   - 2.1 Interfejs linijkowy polega na wyświetleniu pyt. i oczekiwania na odp. wpisywaną z klawiatury.
   - 2.2 Interfejs pełnoekranowy: działa podobnie do interfejsów graficznych. Jednak ma inne skróty klawiszowe i inną obsługę myszy. Do pełnoekranowych interfejsów tekstowych często stosuje się bibl. NCurses (pętla gry).
3. Interfejs graficzny: To interfejs głównie do klikania. Od dziesięcioleci bezkonkurencyjne bibl.[6] do prog. interfejsów graf. jest Qt (pętla zdarzeń) i SDL (pętla gry).

## 6.5 Sposób zgłaszania błędów

Jak się okazuje nawet sposób zwracania błędów przez f. ma wpływ na architekturę prog. Są 3 główne sposoby zwracania błędów:

### 6.5.1 Wartość zwracana z f.

Po prostu wartość zwracana zawiera kod błędu. W sytuacji gdy wszystko gra f. zwraca 0 (tak samo jest w programach powłoki) w przeciwnym wypadku war. zw. jest kodem błędu.

**Problem ze zwracaniem kodu błędu w wyniku f. jest taki, że nie ma możliwości normalnego zwracania jej wyniku.**

```
/* Zwracanie kodu błędu przez war. zw.:*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Zmienne globalne:::::::::::::::::::::::::::::*/
enum KodyBledow
{
   eOK,
   eDrukowanie
};

const char gJakisFajnyTekst[] = { "Jakiś fajny tekst!" };

/* Funkcje:::::::::::::::::::::::::::::::::::::::*/
int drukuj(const char* pNapis)
{
   int cosNieTak = (printf("%s\n", pNapis) == 0);

   if(cosNieTak)
      return eDrukowanie;

   return eOK;
}

int main()
{
   int lKodBledu = drukuj(gJakisFajnyTekst);

   if(lKodBledu)
   {
      fprintf(stderr, "Wystąpił błąd! Kod: %d\n",
lKodBledu);
      exit(EXIT_FAILURE);
   }

   printf("Wykonanie prawidłowe!\n");
   exit(EXIT_SUCCESS);
}
```

6Można powiedzieć, że to stagnacja ponieważ nie są one jakieś nadzwyczajne.

## 6.5.2 errno

Cytat: "errno jest definiowana przez standard ISO C jako modyfikowalna l-wartość typu int, która nie może zostać jawnie zadeklarowana; errno może być makrem. Wartość errno jest lokalna w obrębie wątku, jej zmiana w jednym wątku nie wpływa na wartość w innym."[7]

**Koncepcja enrno jest nietrafiona, bo nie jest znany używany zakres wart. errno. Dlatego by unikać konfliktów do nr kodu błędu konieczna jest nazwa bibl. która go zgłasza. Wtedy programista mógłby sobie ustawić jako nazwę bibl. nazwę swojej aplikacji i samodzielnie zdefiniować jej kody błędów.**

```
/* Przykład użycia errno do zwracania kodów błędów. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/* Zmienne globalne:::::::::::::::::::::::::::::*/
enum KodyBledow
{
   eOK,
   eMalloc,
   eSprntf
};

const char gJakisFajnyTekst[] = { "Jakiś fajny tekst!" };

/* Funkcje:::::::::::::::::::::::::::::::::::::::*/
const char* konwertuj(const char* pNapis)
{
   char* lWynik = malloc(100);

   if(!lWynik)
   {
      errno = eMalloc;
      return pNapis;
   }

   if(sprintf(lWynik, "%s %s\n", pNapis
         , "I jakiś fajny dodatek!") == 0)
   {
      free(lWynik);
      errno = eSprntf;
      return pNapis;
   }

   return lWynik;
}

int main()
{
   printf("%s", konwertuj(gJakisFajnyTekst));

   if(errno)
   {
      fprintf(stderr, "Wystąpił błąd! errno: %d\n",
errno);
      exit(EXIT_FAILURE);
   }

   printf("Wykonanie prawidłowe!\n");
   exit(EXIT_SUCCESS);
}
```

## 6.5.3 Wyjątki

W j. C++ wprowadzono wyjątki (oczywiście j. D też je ma). Wyjątki składają się z dwóch el.: najpierw deklaruje się dwa bloki kodu:

```
try
{}
catch()
{}
```

Następnie w bloku try wywołuje się f. która potencjalnie może rzucić wyjątek (wyjątek może rzucić f. wywołana z bloku try bezpośrednio lub pośrednio). Wyjątek rzuca się dyrektywą throw. Wtedy następuje zwinięcie stosu (wyskoczenie z f.) do bloku try i przekazanie kontroli do bloku catch.

W bloku obsługi wyjątków (blok catch) normalnie działa polimorfizm, tak więc można swobodnie definiować typy wyjątków i w dowolnych miejscach można je łapać.

Złapane wyjątki można dalej propagować. Poniżej pokazałem propagację wyjątku z dodaniem dodatkowej informacji.

**Podobnie jak w przypadku errno z j. C tak samo koncepcja wyjątków w C++ jest nietrafiona. Wynika to z faktu, że Komitet Centralny C++ zaleca tworzenie nowej kl. dla każdego rodz. wyjątku.**

Różnica jest jedynie taka, że errno nie można poprawić, a wyjątki tak. Oto co trzeba w tym celu zrobić:

1. Należy stworzyć własną kl. Wyjątek której konstruktor będzie pobierał nast. param.: nazwę modułu, kod błędu, treść błędu, plik, nazwa f. (razem z nazwą kl.), nr linii;
2. Należy stworzyć makro które będzie pobierało nazwę modułu (bibl. lub prog.), kod błędu i treść błędu (reszta danych jest pobierana makrami typu __FILE__). To makro należy stosować w plikach definicji (C++);
3. Należy stworzyć makro które będzie pobierało nazwę modułu i treść błędu (wywołuje on makro z p. 2 z kodem błędu -1). To makro należy stosować w plikach deklaracji (H++), np. w szablonach lub w f. wstawianych/inline.

Ogólnie bardzo nieufnie przyjmowano nowinki C++. Brak wiary w wyjątki pokutuje do dziś, np. w bibl. Qt wer. 6 z 2020r. nadal nie używa się wyjątków.

Perfidnym rozwiązaniem w bibl. Qt uniemożliwiającym wygodną, globalną obsługę wyjątków jest łapanie wyjątków w kodzie obsługi pętli zdarzeń w Qt. To jest tym bardziej perfidne, że Qt Group oficjalnie głosi propagandę, że Qt nie używa wyjątków. To perfidne rozw. działa tak, że w razie wystąpienia takiego nieprzechwyconego wyjątku program Qt wypisuje na konsolę komunikat i kończy działanie. Co w normalnym przypadku jest całkowitym zaskoczeniem dla użytkownika prog.

```
// Przykład użycia wyjątków:
#include <exception>
#include <string>
#include <iostream>

using namespace std;

// Zmienne globalne:::::::::::::::::::::::::::::
enum KodyBledow
{
   eOK,
   eDrukowanie
};

char gJakisFajnyTekst[] = { "Jakiś fajny tekst!" };

// Klasy:::::::::::::::::::::::::::::::::::::::
class Wyjatek : public exception
```

7man errno

```
{
public:
   Wyjatek(string pKomunikat, int pKod)
      : cKomunikat(pKomunikat), cKod(pKod)
   {
   }

   const char* what() const noexcept
   {
      return cKomunikat.c_str();
   }
   int kod() const
   {
      return cKod;
   }

protected:
   const string cKomunikat;
   int cKod;
};

// Funkcje::::::::::::::::::::::::::::::::::::
void drukuj(string pNapis)
{
   try
   {
      cout << pNapis << endl;
   }
   catch(const exception& pWyjatek)
   {
      string lTresc = "Coś jest nie tak z cout w f.
drukuj()!";
      lTresc += pWyjatek.what();
      throw Wyjatek(lTresc, eDrukowanie);
   }
}

int main(int n, char** w)
{
   try
   {
      drukuj(gJakisFajnyTekst);
   }
   catch(const Wyjatek& pWyjatek)
   {
      cerr << "Wystąpił wyjatek! Treść: "
         << pWyjatek.what()
         << " Kod błędu: " << pWyjatek.kod() << endl;
      exit(EXIT_FAILURE);
   }
   catch(const exception& pWyjatek)
   {
      cerr << "Wystąpił wyjatek! Treść: "
         << pWyjatek.what() << endl;
      exit(EXIT_FAILURE);
   }
   catch(...)
   {
      cerr << "Wystąpił nieznany wyjatek!" << endl;
      exit(EXIT_FAILURE);
   }

   cout << "Wykonanie prawidłowe!" << endl;
   exit(EXIT_SUCCESS);
}
```

## 6.6 Komentarze

Komentarze dzielimy na 2 rodz.:

1. Komentarze dokumentujące, z których generowana jest dokumentacja dla użytkowników (gł. Doxygen);
2. Komentarze techniczne, czyli wyjaśniające szczegóły programistom utrzymującym kod.

Kod powinien być samoopisowy tak jak to tylko możliwe. Kod nie powinien wymagać komentarzy. Komentarzy należy unikać po to by ułatwiać analizę i utrzymanie kodu.

Komentarze powinny wyłącznie dokumentować udostępniane interfejsy API. Tylko dlatego, żeby inni programiści mogli szybko je poznać i szybko zacząć je używać.

Komentarze eliminujemy:

1. Nazwami opisowymi f., zmiennych i makr.
2. Tworzymy nową f. pomocniczą z nazwą opisową. Po prostu wycinamy ten fragment kodu (który chcieliśmy skomentować) i tworzymy nową f. W normalnej sytuacji spadek wydajności jest niezauważalny na dzisiejszych prockach (po za tym można tą f. zrobić wstawianą/inline co powoduje, że koszt wywołania będzie zerowy).
3. Tworzymy komunikat wypisujący treść komentarza. Na marginesie: szczególnie często stosuję to w skryptach.

**Okazuje się, że komentarze należy umieszczać wyłącznie w plikach C++, bo umieszczanie ich w H++ prowadzi do degeneracji zalet plików nagłówkowych (przejrzystość konieczną do szybkiego ich analizowania).**

**W przypadku szablonów należy rozdzielić kod deklaracji f. i jej definicję umieścić poniżej deklaracji kl. i tam dodawać komentarze.**

# Dodatek 1: Mały sabotaż

Mały sabotaż to nowoczesna odmiana dywersji Szarych Szeregów praktykowana współcześnie przez Etycznych Krakerów zatrudnianych przez tajną policję. Mały sabotaż obejmuje m.in.:

1. Publikowanie opasłych książek i monografii (np. monografie prof. Jana Pająka). Zmusza to do pisania własnych skryptów i instrukcji (takich jak ta broszura);
2. Psucie prog. konsoli (brak potrzebnych opcji, np. sed i brak przetwarzania blokowego, blokowanie użycia w skryptach, np. pdfgrep). Zmusza to do pisania własnych narzędzi konsoli;
3. Brak polskiej dok. (np. większość dok. sys. Ubuntu 20.04). Zmusza to do ogłupiającego tłumaczenia w myślach;
4. Psucie prog. okienkowych przez brak przestrzegania zasady „zero-conf” czyli zmuszanie do ogłupiającego, wielokrotnego powtarzania tych samych czynności (np. Qt Creator). Zmusza to do prog. własnych prog. (np. edytor tekstu Tekstprofan);
5. Psucie składni i organizacji kodu w j. programowania. (np. brak dziedziczenia operatorów w C++). Zmusza to do programowania makr oraz parserów kodu źródłowego i generatorów kodu;
6. Psucie interfejsów bibl. (API) (np. kl. QFile oraz QProcess z Qt). Zmusza to to prog. własnych

normalnych kl. opakowujących (np. bibl. energo-protekcja);

7. Kasowanie pam. (np. o tym co b. chciałem zapamiętać czytając dany podręcznik). Zmusza to do programowania prog. w stylu SuperMemo (np. Turborety).

# Dodatek 2. Wzorce proj. - jak na nie patrzeć?

**Programistyczne wzorce projektowe to robota tzw. Bandy Czworga z 1993r. Można powiedzieć, że częściowo to też podpucha z SZAP, bo jak się czyta zestawienie tych wzorców to wiele z nich jest bliźniaczo podobnych do siebie.**

Podam tu jakie wzorce są zdublowane i jakie na prawdę należy sobie przyswoić.

## Wzorce

1. Stan

Robi: Przechowuje aktualny stan logiki, kontroluje przejścia stanów i informuje o zmianie stanu zainteresowane kl. i wtyczki.

Za pomocą: Zwykły enum i zmienna tego typu w kl. logiki.

2. Prototyp

Robi: Jest to wirt. f. kopiuj() pełniąca w kl. rolę wirt. konstruktora.

Za pomocą: Zwykła f. wirt.

3. Singleton

Robi: Jedyny w aplikacji obiekt świadczący określone usługi.

Za pomocą: Jest on tworzony na żądanie tuż przed pierwszym użyciem.

4. Budowniczy

Zamiast: Fabryka abstrakcyjna, Metoda wytwórcza.

Robi: Buduje obiekty za pomocą f. cząstkowych, lub przez łańcuch zobowiązań.

Za pomocą: Abstrakcyjnego interfejsu.

5. Kompozyt

Robi: Dostarcza interfejs abstrakcyjny dla podobnych o.

Za pomocą: Zwykłe dziedziczenie.

6. Iterator-Wizytator

Robi: Umożliwia iterację po kolekcji, której elementy odwiedza wizytator.

Za pomocą: Pary f. lub pary kl.

7. Wywołanie Zwrotne

Zamiast: Dekorator, Obserwator

Robi: Dodaje nowe f. do o. kl. w czasie działania prog.

Za pomocą: Rejestruje f. które ma wywoływać w określonych sytuacjach.

**W przypadku wywołań zwrotnych często zdarza się, że trzeba je synchronizować w celu unikania konfliktu z innymi wątkami.**

8. Producent-Konsument

Robi: odbiera dane z wątku głównego, buforuje je i wysyła przez sieć.

Za pomocą: 2 dodatkowe wątki do zapisu i do odczytu z gniazda z użyciem 2 buforów pierścieniowych (tryb full-duplex).

9. Polecenie

Robi: Buforuje napływające rozkazy.

Za pomocą: Zwykła kl.

10. Pamiątka

Robi: Realizuje transakcje w aplikacjach finansowych i historii działań we wszelkich edytorach graficznych i tekstowych.

Za pomocą: Zwykła kl.

11. Metoda szablonowa

Robi: Implementuje alg. niezależnie od danych na których działa.

Za pomocą: Zwykły szablonu f. lub szablon kl.

12. Konwerter

Zamiast: Adapter, Fasada, Most.

Robi: Udostępnia nowy interfejs dla starej funkcjonalności.

Za pomocą: Zwykła kl.

13. Narzędzia-Logika-Okna

Zamiast: Model-Widok-Kontroler (w j. ang. MVC)

Robi: Logika steruje zarówno narzędziami jak i oknami. Dane do edycji w oknach dostarcza logika. Kontrola popr. wprowadzanych danych powinna się odbywać zarówno na poziomie kontrolek okna jak i na poziomie f. pub. w logikach.

Za pomocą: kl. narzędzi, kl. logiki i kl. okien.

## Antywzorce

1. **Pyłek**

Robi: Dostarcza interfejs abstrakcyjny dla podobnych o.

Za pomocą: Zwykłe dziedziczenie z leniwym niszczeniem o. w celu unikania ich ponownego tworzenia.

Dlaczego bezcelowy: Szczegół optymalizacyjny.

2. **Fabryka abstrakcyjna, Metoda wytwórcza**

Robi: To samo co budowniczy.

Za pomocą: Zwykła kl.

Dlaczego bezcelowy: Zbędne powielenie.

3. **Dekorator, Obserwator**

Robi: To samo co wywołanie zwrotne.

Za pomocą: Zwykłe f.

Dlaczego bezcelowy: Zbędne powielenie.

4. **Adapter, Fasada, Most**

Robi: To samo co konwerter.

Za pomocą: Zwykłe kl.

Dlaczego bezcelowy: Zbędne powielenie.

5. **Interpreter**

Robi: Parsuje dok. w określonych j. opisujących zawartość.

Za pomocą: o. kl.

Dlaczego bezcelowy: To program a nie wzorzec.

6. **Model-Widok-Kontroler** (w j. ang. MVC)

Robi: Tworzy spójne trio kl. obsługujących okno prog.

Za pomocą: kl. model zawiera dane edytowane w oknie prog, kl. widok to okno i jego kontrolki, kontroler weryfikuje dane przed wprowadzeniem ich do modelu.

Dlaczego bezcelowy: Zastępuję go wzorcem Narzędzia-Logika-Okna

# Dodatek 2: Zasady używania baz danych SQL

Jak wiadomo całe szaleństwo skryptowego j. SQL, okraszone naukowym bełkotem, wynika jedynie z chęci spowolnienia działania komputerów oraz wymuszania ogłupiających rozw. Dlatego aby temu przeciwdziałać należy:

## W bibl. firmowej należy zakodować kl. TabelaSQL z podst. f.

1. utworzTablice(...);
2. zapis(...); # rekordu
3. jest(licz64 aID); # rekord
4. odczyt(licz64 aID); # rekordu
5. usun(licz64 aID).

## Dla każdej tabeli należy utworzyć odpowiadające jej kl.

```
NazwaProg/Narzedzia/Baza/TabelaDane.h++
NazwaProg/Narzedzia/Baza/TabelaDane.c++
NazwaProg/Narzedzia/Baza/TabelaFunkcje.h++
NazwaProg/Narzedzia/Baza/TabelaFunkcje.c++
```

Gdy tabela ma nazwę Tabela, to kl. TabelaDane ma zmienne odpowiadające polom z tej tabeli. Ta kl. ma f. TabelaDane::zmienne(), która zwraca pary Napis:Wartość w celu aut. tworzenia tabel i aut. serializacji (odczytu i zapisu) tej kl.

Natomiast TabelaFunkcje dziedziczy po TabelaSQL i uzupełnia ją o f. wyszukiwania danych w tabeli Tabela.

**Zabronione jest strzelanie SELECT do bazy z dowolnego miejsca w prog. Wszelkie tego typu zadania wykonuje kl. TabelaFunkcje (w przypadku tabeli Tabela).**

## Należy zakodować f. gNormSort

Ta f. powinna normalnie sortować napisy i numery w tych napisach łącznie z przypadkami występowania zer wiodących. Robi się to używając 2 zakresów w obu porównywanych ciągach znaków. Te zakresy przesuwa się od pocz. do końca obu porównywanych ciągów znaków i kolejno porównuje. Do porównania ciągów znaków używa się wtedy f. Unicode. Natomiast przy wykryciu numerów konwertuje się je do liczb 64bit. i porównuje jako l. Użycie f. gNormSort może być takie:

```
SELECT Imię, Nazwisko, WIEK FROM Użytkownik ORDER BY
Imię, Nazwisko COLLATE gNormSort
```

## Należy zakodować f. gWyrReg

Ta f. powinna oferować normalne wyrażenia regularne zamiast „place holders”. Tą f. należy zakodować gdyż nie ma sensu szarpać się z wbudowanym operatorem LIKE polecenia SELECT, bo to całkowicie nienormalne rozw. Użycie f. gWyrReg może być takie:

```
SELECT Imię, Nazwisko, WIEK FROM Użytkownik WHERE
gWyrReg(Nazwisko, 'JAWO.*KI') = 1
```

## Należy zakodować f. gWyszRozmyte

Chodzi o to by pojedyncza literówka nie psuła zapytania. Użycie f. gWyszRozmyte może być takie:

```
SELECT Imię, Nazwisko, WIEK FROM Użytkownik WHERE
gWyszRozmyte(Nazwisko, 'JAWOR-KI') >= 3
```

Wart. zwracana przez f. gWyszRozmyte to stopień/współczynnik dopasowania. W pow. przykładzie zakładam, że stopień dopasowania musi wynosić min. 3 znaki. Zamiast l. całkowitej f. gWyszRozmyte może też zwracać ułamek od 0,0 do 1,0.

## Raporty należy generować lokalnie na serwerze bazodanowym

Podczas pracy z bazą sieciową do realizacji raportów nie należy kopiować wszystkich rekordów przez sieć. Zamiast tego należy zaprogramować demona który będzie działał na tym samym serwerze co motor SQL i który będzie uruchamiał programiki generujące raporty. Ich cechą

będzie komunikacja z motorem SQL na localhost (zamiast przez sieć) i to będzie główny powód ich wydajnego działania. Natomiast wydzielenie raportów do osobnych programików będzie pozwalało na ich łatwe przerywanie (przez ubijanie procesów – jest to chyba jedyna metoda na przerywanie zapytań – bo awaryjne przerwanie zapytania jest zazwyczaj jest blokowane na poziomie C/C++ przez kl. dostępowe w bibl. Qt i VCL).

10 Dodatkowo takie raporty łatwo uruchamiać zdalnie, co jest wygodne przy kodowaniu i testowaniu.

# 7 Dodatek 3: Formatowanie kodu

W trakcie kodowania nie należy dążyć do pedantycznie sformatowanego kodu. Zamiast tego przed commitem należy używać takich prog. jak Artistic Style.

# 8 Licencja

Jest to licencja dotycząca tego dokumentu. Pliki wskazywane przez linki mogą być publikowane na innych licencjach. Zasady licencji:

1. **Zezwolenie na kopiowanie** Zezwala się na niekomercyjne kopiowanie tego dokumentu;
2. **Zezwolenie na udostępnianie** Ten dokument można udostępniać (jednak bezpłatnie);
3. **Zabronione modyfikowanie** Tego dokumentu nie można modyfikować ani skracać ani dodawać czegokolwiek.
4. **Ograniczenia licencji nie dotyczą autora.**

# 9 Bibliografia

# Bibliografia

c++podróż: Bjarne Stroustrup, C++. Podróż po języku dla zaawansowanych, 2023