1. Literate programming na małym przykładzie.

Profesor Donald Knuth w 1984 r. zauważył, że istniejące oprogramowanie ma duże braki w dziedzinie dokumentacji. Jeśli w ogóle istniała, była często niekompletna lub niezgodna z kodem, który zdążył się zmienić od czasu jej napisania. Zaproponował więc pisanie dokumentacji równocześnie z oprogramowaniem i to w taki sposób, żeby były ze sobą nierozerwalnie związane.

Brzmi ciekawie, choć narzuca programiście niezwykle ciężki obowiązek – nie dość, że ma programować, to teraz zamiast dodawania skąpych komentarzy musiałby zrozumiale opisywać, co i dlaczego akurat tak programuje. Proces taki może być trudny przede wszystkim ze względu na częsty niedostatek zdolności literackich u programistów, ale też z powodu braku odpowiednich narzędzi.

O ile na pierwszy problem niewiele można poradzić, o tyle drugi prof. Knuth rozwiązał przez skonstruowanie specjalnego języka do opisu programów: WEB. Nazwa wzięła się od pajęczyny, jaką tworzy kod przeplatany dokumentacją, czy raczej dokumentacja z wstawkami kodu. W 1987 r. Silvio Levy zaadaptował ten system do języka C i tak narodził się CWEB.

Język WEB łączy dwa światy: programistyczny służący do właściwego programowania oraz literacki, który dzięki ogromnym możliwościom typograficznym systemu składu TEX[1] pozwala na tworzenie dokładnej i *ładnie wyglądającej* dokumentacji.

Niezwykle przydatną własnością języka CWEB jest możliwość opisywania niewielkich porcji kodu w języku C bez potrzeby zachowywania kolejności, tak jak wymagałby tego kompilator. W efekcie łatwiej jest przy czytaniu zrozumieć ich działanie, a także prościej później modyfikować.

Idea jest prosta, a korzystanie jeszcze prostsze. W jednym pliku (*.w) znajduje się program i dokumentacja. Program CTANGLE[2] wyciąga z niego źródła w C i odpowiednio układa w plik *.c, a CWEAVE[3] przygotowuje plik źródłowy dla kompilatora TEX, który produkuje przyjemne dla oka dokumenty.

Prześledźmy na prostym przykładzie opis takiego programu. Najprostszy możliwy program w CWEB to, jak w każdym języku, „Hello world!”. No dobrze, może ten byłby zbyt prosty – CWEB ma wiele mechanizmów, których nie zauważylibyśmy w tak prostym przykładzie. Napiszmy zatem program do obliczania sumy niezmodyfikowanych punktów funkcyjnych (`UFP`).

---

[1] Co zabawne, język WEB powstał m.in. dlatego, że prof. Knuthowi brakowało odpowiedniego języka do napisania właśnie TEX-a

[2] tangle: plątać

[3] weave: tkać

2. Program.

Opis programu składać się będzie z ponumerowanych paragrafów. Zaczynamy od szkieletu programu w C i powoli będziemy go wypełniać, wyjaśniając po drodze, co robią poszczególne części. Każdy paragraf zawiera odnośniki do sekcji, w których jest użyty oraz cytowany. Najważniejsze części zaczynają się na nowej stronie i są automatycznie dołączane do spisu treści.

Nasz program w C na razie jest bardzo prosty, mieści się w jednym pliku (oczywiście jest możliwe tworzenie programów rozrzuconych po wielu plikach), a jego szkielet wygląda tak:

```
⟨ Pliki nagłówkowe 16 ⟩
int main( )
{
  ⟨ Zmienne lokalne 5 ⟩
  ⟨ Przypisanie wag 8 ⟩
  ⟨ Wypełnij matrycę elementów 9 ⟩
  ⟨ Oblicz i wypisz sumę punktów 4 ⟩
  ⟨ Przeprowadź korektę punktów 6 ⟩
  ⟨ Podaj estymacje kosztów 7 ⟩
  return 0;
}
```

3. Na samym początku zdefiniujemy strukturę, w której będziemy przechowywać dane. Dla uproszczenia wykorzystamy jedną strukturę do przechowywania zarówno wag, jak i liczby elementów wprowadzanych przez użytkownika. Taka konstrukcja ułatwi późniejszą rozbudowę programu o możliwość modyfikacji samych wag.

```
⟨ Definicja struktury wag 3 ⟩ ≡
  struct Waga {
    char nazwa[16];      /* nazwa elementu */
    int proste;      /* wagi */
    int przecietne;
    int zlozone;
    int nproste;      /* liczby elementow (do wypelnienia przez użytkownika) */
    int nprzecietne;
    int nzlozone;
  };
```

Ten kod jest użyty w sekcji 16.

4. Właściwy opis programu zaczniemy od drugiej części. Załóżmy, że mamy już wprowadzone potrzebne dane i przystępujemy do obliczeń. Wzór na ważoną sumę punktów funkcyjnych jest banalny:

$$\texttt{UFP} = \sum_{i=1}^{\texttt{NPT}} element_i \cdot waga_i$$

Przekłada się on na następujący kod:

```
⟨ Oblicz i wypisz sumę punktów 4 ⟩ ≡
  UFP = 0;
  for (i = 0; i < NPT; i++) {
    UFP += wagi[i].proste * wagi[i].nproste;
    UFP += wagi[i].przecietne * wagi[i].nprzecietne;
    UFP += wagi[i].zlozone * wagi[i].nzlozone;
  }
  printf("UFP (Unadjusted Function Points): %d\n",UFP);
```

Ten kod jest użyty w sekcji 2.

5. Nie zapominamy o deklaracji tych zmiennych.

⟨ Zmienne lokalne 5 ⟩ ≡
```
int UFP, FP;
```
Zobacz też w sekcjach 11 i 17.

Ten kod jest użyty w sekcji 2.

6. Aby program nadawał się do estymowania kosztów oprogramowania, należałoby zmodyfikować[4] UFP o wagi dla czternastu czynników technicznych (Degree of Influence, DI) według następującego wzoru:

$$\mathtt{FP} = (0.65 + 0.01 \sum_{k=1}^{14} \mathtt{DI}_k) \cdot \mathtt{UFP}$$

gdzie czynniki techniczne $\mathtt{DI}_k \in 0 \ldots 5$.

Dla uproszczenia przykładu przyjmiemy, że czynniki techniczne nie wpłynęły na skomplikowanie programu, czyli FP = UFP.

⟨ Przeprowadź korektę punktów 6 ⟩ ≡
```
FP = UFP;      /* przy rozbudowywaniu programu wprowadzić korektę DI */
printf("FP (Function Points): %d\n", FP);
⟨ Separator 15 ⟩
```
Ten kod jest użyty w sekcji 2.

7. Skoro mamy już obliczone FP, to możemy podać różne estymacje czasu pisania oprogramowania w osobomiesiącach.

Albrecht i Gaffney	$E = -13.39 + 0.0545 \cdot \mathtt{FP}$
Kemerer	$E = 60.62 \cdot 7.728 \cdot 10^{-8} \cdot \mathtt{FP}^3$
Matson, Barnett i Mellichamp	$E = 585.7 + 15.12 \cdot \mathtt{FP}$

⟨ Podaj estymacje kosztów 7 ⟩ ≡
```
printf("Estymacje kosztów wg:\n");
printf("%-30sE = %7.2f\n", "Albrecht i Gaffney", -13.39 + 0.0545 * FP);
printf("%-30sE = %7.2f\n", "Kemerer", 60.62 * 7.728 * 0.00000001 * FP * FP * FP);
printf("%-30sE = %7.2f\n", "Matson, Barnett i Mellichamp", 585.7 + 15.12 * FP);
```
Ten kod jest użyty w sekcji 2.

8. A teraz te mniej ciekawe części. Definiujemy elementy i przypisujemy im wagi, przy okazji zerując liczby elementów, które zostaną wprowadzone przy użyciu kodu z ⟨ Wypełnij matrycę elementów 9 ⟩.

⟨ Przypisanie wag 8 ⟩ ≡
```
struct Waga wagi[NPT] = {
{"wejścia", 3, 4, 6, 0, 0, 0},
{"wyjścia", 4, 5, 7, 0, 0, 0},
{"zapytania", 3, 4, 6, 0, 0, 0},
{"pliki główne", 7, 10, 15, 0, 0, 0},
{"interfejsy", 5, 7, 10, 0, 0, 0}
};
```
Ten kod jest użyty w sekcji 2.

[4] Mechanizm zdefiniowany w ⟨ Wypełnij matrycę elementów 9 ⟩ w zasadzie już jest gotowy, trzeba go tylko lekko zmodyfikować, żeby był funkcją operującą na tablicach podawanych w argumencie.

9. Część struktury wag użytkownik musi wypełnić. Wypiszemy bieżącą zawartość tablicy elementów i damy użytkownikowi wybór, dla którego elementu chce poprawić liczby. Przyjmiemy wprowadzone dane i zapiszemy, jeżeli będą poprawne. Proces powtarzamy, dopóki użytkownik nie da znać, że już koniec. Za każdym razem dla czytelności oddzielimy to wynikiem kodu ⟨ Separator 15 ⟩.

```
⟨ Wypełnij matrycę elementów 9 ⟩ ≡
  do {
    ⟨ Wypisz matrycę elementów 10 ⟩
    ⟨ Wczytaj złożoności elementów 12 ⟩
    ⟨ Zapisz wczytane liczby 13 ⟩
  } while (elnum ≠ 0);
  ⟨ Separator 15 ⟩
```

Ten kod jest cytowany w sekcjach 6 i 8.

Ten kod jest użyty w sekcji 2.

10. Pokazywanie liczby elementów w każdej kategorii realizuje poniższy kawałek kodu. Korzystamy z $i+1$, bo tablice w C numeruje się od zera, a ludzie przyzwyczajeni są do numerowania od 1. Ukrywamy prawdziwe indeksy, powiększając o jeden przy wyświetlaniu. To zresztą pozwoli nam na w miarę eleganckie zakończenie wprowadzania danych, gdy użytkownik wpisze 0.

```
⟨ Wypisz matrycę elementów 10 ⟩ ≡
  ⟨ Separator 15 ⟩
  printf("%30s␣|proste|przec.|␣złoż.|\n", "");
  for (i = 0; i < NPT; i++) {
    printf("Element␣nr␣%d:␣%-16s␣|%5d␣|%5d␣|%5d␣|\n", i + 1, wagi[i].nazwa, wagi[i].nproste,
        wagi[i].nprzecietne, wagi[i].nzlozone);
  }
  ⟨ Separator 15 ⟩
```

Ten kod jest użyty w sekcji 9.

11. Zadeklarujmy potrzebne nam za chwilę zmienne tymczasowe; *elnum* to numer elementu, który użytkownik będzie wypełniał, a *elw1*, *elw2* i *elw3* to liczby prostych, przeciętnych i złożonych elementów danej kategorii.

```
⟨ Zmienne lokalne 5 ⟩ +≡
  int elnum = 0, elw1 = 0, elw2 = 0, elw3 = 0;
```

12. Wczytywanie liczby elementów będziemy robić seriami po trzy liczby, tj. wszystkie złożoności wybranego elementu równocześnie. Można zrobić wprowadzanie po jednej liczbie na raz, ale wtedy wpisywanie wszystkich liczb będzie trwało dłużej i łatwiej się pomylić. Wpisywanie po trzy to rozsądny kompromis między łatwością obsługi dla początkującego użytkownika a szybkością wprowadzania danych dla użytkownika zaawansowanego.

```
⟨ Wczytaj złożoności elementów 12 ⟩ ≡
  printf("Podaj␣numer␣elementu␣(0␣by␣skończyć):␣");
  scanf("%d", &elnum);
  if (elnum > 0 ∧ elnum ≤ NPT) {
    printf("Podaj␣liczby␣elementów␣,,%s''␣(proste␣przeciętne␣złożone):␣", wagi[elnum].nazwa);
    scanf("%d␣%d␣%d", &elw1, &elw2, &elw3);
  }
```

Ten kod jest cytowany w sekcji 13.

Ten kod jest użyty w sekcji 9.

13. Zapisujemy wprowadzone przez użytkownika dane po sprawdzeniu ich poprawności. $wagi[elnum-1]$ z powodu $i+1$ jak wyżej. Oczywiście jeśli zmienimy sposób wczytywania określony we ⟨ Wczytaj złożoności elementów 12 ⟩, to musimy też poprawić tutaj.

```
⟨ Zapisz wczytane liczby 13 ⟩ ≡
  if (elnum > 0 ∧ elnum ≤ NPT) {
    ⟨ Sprawdzanie danych wejściowych 14 ⟩
    wagi[elnum − 1].nproste = elw1;
    wagi[elnum − 1].nprzecietne = elw2;
    wagi[elnum − 1].nzlozone = elw3;
  }
```

Ten kod jest użyty w sekcji 9.

14. Sprawdzanie danych wejściowych odłożymy na później.

```
⟨ Sprawdzanie danych wejściowych 14 ⟩ ≡      /* dorobić sprawdzanie danych wejściowych */
```

Ten kod jest użyty w sekcji 13.

15. Separator jest nieskomplikowanym kawałkiem kodu i pozwala na łatwą prezentację podsumowań użycia robionych przez CWEB. Automatyczne aktualizacje numerów sekcji są bardzo wygodne zarówno dla autorów, jak i czytelników dokumentacji.

```
⟨ Separator 15 ⟩ ≡
  printf("-----\n");
```

Ten kod jest cytowany w sekcji 9.

Ten kod jest użyty w sekcjach 6, 9 oraz 10.

16. Techniczne drobiazgi na koniec. Definicje i pliki nagłowkowe...

```
#define NPT 5      /* ile mamy elementów */
⟨ Pliki nagłówkowe 16 ⟩ ≡
#include <stdio.h>      /* tu mamy printf( ) i scanf( ) */
#include <stdarg.h>      /* dla scanf( ) */
  ⟨ Definicja struktury wag 3 ⟩
```

Ten kod jest użyty w sekcji 2.

17. ...oraz pozostałe zmienne lokalne. Proszę zauważyć, że część zmiennych lokalnych dla *main*( ) zdefiniowaliśmy w innych miejscach – tam, gdzie miały być za chwilę używane, żeby daleko nie szukać ich definicji. CWEAVE odpowiednio je połączył i skomentował.

```
⟨ Zmienne lokalne 5 ⟩ +≡
  int i;
```

18. Koniec programu.

I to wszystko. Teraz plik źródłowy `ufp.w` traktujemy programem `ctangle`, z którego otrzymujemy `ufp.c`, oraz programem `cweave`, z którego dostajemy `ufp.tex`; pierwszy kompilujemy kompilatorem C, np. `gcc`, a drugi przetwarzamy do DVI kompilatorem TEX-a, np. programem `mex`[5], lub od razu tworzymy PDF przy pomocy `pdfmex`.

Potem tylko podziwiamy, oglądając wynik działania i czytając dokumentację.

Całość utworzyłem na aksonie i wszystkie komendy są dostępne dla wszystkich użytkowników. Pliki źródłowe i wynikowe można obejrzeć pod adresem http://akson.sgh.waw.pl/~chopin/ufp/

Przy oglądaniu wygenerowanego `ufp.c` proszę zwrócić uwagę na obficie występujące dyrektywy **#line**, które w przypadku ewentualnego błędu składniowego umożliwiają kompilatorowi C pokazanie, w którym miejscu w pliku ufp.w (tak! .w, nie .c) znajduje się ten błąd:

```
./ufp.w: In function ‘main’:
./ufp.w:184: error: ‘ifp’ undeclared (first use in this function)
./ufp.w:184: error: (Each undeclared identifier is reported only once
./ufp.w:184: error: for each function it appears in.)
```

co zobaczyłem przy próbie kompilacji programu, gdy zrobiłem literówkę w nazwie zmiennej.

19. Bibliografia.

- *Literate Programming* http://www.literateprogramming.com/
- Donald E. Knuth. *TEX: The Program*, *Computers and Typesetting*, tom B. Addison-Wesley. 1986.
- Donald E. Knuth i Silvio Levy. *The CWEB System of Structured Documentation*. Addison-Wesley. 1993.

[5] mex to zlokalizowana polska wersja tex

20. Indeks.

Dokumentacja nie ma prawa nazywać się dobrą, jeśli nie posiada indeksu! Tutaj indeks jest tworzony automatycznie, a dodatkowo podkreśla numer sekcji, w której zmienna lub funkcja została zdefiniowana. Dodatkowo na kolejnych stronach otrzymujemy indeks nazw sekcji, a także spis treści.

⟨ Definicja struktury wag 3 ⟩ Użyte w sekcji 16.
⟨ Oblicz i wypisz sumę punktów 4 ⟩ Użyte w sekcji 2.
⟨ Pliki nagłówkowe 16 ⟩ Użyte w sekcji 2.
⟨ Podaj estymacje kosztów 7 ⟩ Użyte w sekcji 2.
⟨ Przeprowadź korektę punktów 6 ⟩ Użyte w sekcji 2.
⟨ Przypisanie wag 8 ⟩ Użyte w sekcji 2.
⟨ Separator 15 ⟩ Cytowane w sekcji 9. Użyte w sekcjach 6, 9 oraz 10.
⟨ Sprawdzanie danych wejściowych 14 ⟩ Użyte w sekcji 13.
⟨ Wczytaj złożoności elementów 12 ⟩ Cytowane w sekcji 13. Użyte w sekcji 9.
⟨ Wypełnij matrycę elementów 9 ⟩ Cytowane w sekcjach 6 i 8. Użyte w sekcji 2.
⟨ Wypisz matrycę elementów 10 ⟩ Użyte w sekcji 9.
⟨ Zapisz wczytane liczby 13 ⟩ Użyte w sekcji 9.
⟨ Zmienne lokalne 5, 11, 17 ⟩ Użyte w sekcji 2.

UFP