13. Jak pisać projekty?

1 Tworzenie programu

Po pierwsze, celem programisty jest napisanie kodu, który będzie spełniał postawione założenia, tzn. będzie robił prawidłowo to co jest oczekiwane. Następnie ważne jest sprawdzenie i uwzględnienie przypadków skrajnych lub szczególnych, aby uniknąć ewentualnych błędów. Jeżeli jest to istotne, w dalszej kolejności można próbować zwiększyć wydajność zaimplementowanego sposobu rozwiązania problemu.

W trakcie pisania kodu należy zwracać uwagę aby był on w odpowiednim stopniu czytelny. Gdy program (lub częściej jego fragment) zostanie napisany i działa tak jak od niego oczekujemy, należy dokonać jego *refaktoryzacji*. Jest to tym bardziej istotne, im dłuższy jest kod lub jeżeli jest on współtworzony przez kilka osób.

Refaktoryzacja to proces wprowadzania zmian w kodzie programu, który zasadniczo nie zmienia jego funkcjonalności, lecz służy utrzymaniu wysokiej jakości organizacji kodu projektu. Refaktoryzacja powinna m.in. zwiększać czytelności kodu, ograniczać redundancję (powtarzające się fragmenty kodu), prowadzić do lepszego zorganizowania struktury projektu.

2 Czysty kod

Po pierwsze, konieczne jest przestrzeganie wybranego schematu wcięć w kodzie. Każdą linię kodu zagnieżdżoną w bloku pomiędzy nawiasami klamrowymi powinna poprzedzać liczba tabulacji (lub np. czterech spacji w zależności od konwencji) o jeden większa niż dla instrukcji nadrzędnej.

Istotne jest także zachowanie logicznych odstępów pionowych, np. brak pustej linii między instrukcjami powinien oznaczać, że zgrupowane instrukcje są ze sobą w pewien sposób powiązane.

Nie:

```
int foo = 0;

while(bar)
{
if(bar % 2)
foo += 2;
baz(3);

}
```

Tak:

```
int foo = 0;

while (bar) {
  if (bar % 2) {
    foo += 2;
  }

  baz(3);
}
```

Z powyższego wynika, że zmienne powinny z reguły być tworzone przed miejscem pierwszego ich użycia, zamiast zbiorczo w jednym miejscu na początku programu.

Dla zwiększenia czytelności warto stosować spacje między operandami w warunkach lub podczas przypisywania wartości zmiennym. Warto również ograniczyć długość wiersza oraz unikać bardzo długich instrukcji (na przykład wypisując coś na ekran).

Nie:

```
int foo=0, temp, i = 0;

bar( foo );

for(i=0;i<=n;i++){
  temp=bar(i);

  foo +=baz(temp); }
```

Tak:

```
int foo = 0;

bar(foo);

for (int i = 0; i <= n; i++) {
  int temp = bar(i);
  foo += baz(temp);
}
```

Dobry programista powinien starannie dobierać nazwy dla tworzonych zmiennych i funkcji. To głównie nazwy decydują o poziomie czytelności kodu! Opisowe nazwy znacznie przyspieszą orientację w kodzie. Istotne jest stosowanie jednolitego nazewnictwa w obrębie całego programu, np.:

```
#define UPPERCASE 7     // dla stalych
int snake_case = 0;     // dla nazw zmiennych i funkcji
struct CamelCase;       // dla nowych typow
```

Jedną z najstarszych zasad programowania jest unikanie tzw. magicznych liczb. Są to liczby, które powinny być stałymi. Na przykład liczba 3600 może zostać zamieniona stałą o nazwie `SECONDS_PER_HOUR`, a liczba PI w programie powinna rzeczywiście występować pod taką nazwą zamiast wielokrotnego używania wartości 3.14.

Nie:

```
for (int i=0;i<d;i++)
{
  cin >> t;

  if (t > y) s += t - y;
  else if (t < x) s += x - t;
  else s = 0;
}
```

Tak:

```
for (int i = 0; i < liczbaRzutow; i++) {
  wczytaj(rzut);

  if (czyTrafiono(rzut))
    punktyKarne = 0;
  else
    punktyKarne = policzPunktyKarne(rzut);
}
```

Komentarze również ułatwiają zrozumienie kodu, jednak należy je stosować tylko wówczas gdy nazwy zmiennych i funkcji wydają się w tym niewystarczające.

Nie:

```
// parzysta to zwieksz
if (i % 2 == 0) {
  i++; //zwiekszenie
}
```

Tak:

```
if (czyParzysta(i)) {
  i++;
}
```

Komentarz powinien odpowiadać na pytanie *dlaczego to robię?*, a nie *co robię?*, dlatego powyższe komentarze są niewłaściwe. Natomiast komentarz opisujący ogólny algorytm działania skomplikowanej funkcji może być czasem niezbędny.

Nie należy się również bać stosowania funkcji — wprowadzenie funkcji daje często możliwość rezygnacji z nadmiarowego komentarza i zwiększa czytelność kodu poprzez użycie odpowied-

niej nazwy funkcji. Jeżeli jakieś obliczenia w programie wykonywane są kilkakrotnie, to jest to przesłanka do tego aby utworzyć odpowiedzialną funkcję zamiast powielać kod.

3 Pisanie programu

Poniższe wskazówki pomogą w pisaniu dłuższego programu:

- Rozpoczynaj od implementacji najbardziej ogólnych funkcjonalności. Np. projekt dotyczy gry albo bazy danych? Zacznij od zaimplementowania pustego menu dla gracza lub użytkownika.
- Pisz program przyrostowo. Nie jest dobrym pomysłem dokonanie pierwszej kompilacji programu po wytworzeniu dużej ilości kodu. Kompiluj program często, upewniając się, że każdy napisany fragment kodu (funkcja, możliwa do przetestowania część funkcjonalności) kompiluje się i (jeżeli to możliwe) działa poprawnie.
- Po zaimplementowaniu funkcjonalności (funkcji), poświęć chwilę na jej refaktoryzowanie. Upewnij się, czy nazwy zmiennych są wystarczające opisowe, czy sposób rozwiązania problemu jest najprostszy itd., i ew. dokonaj poprawek.
- Dbaj o odpowiednie nazywanie zmiennych i funkcji. Opisowe nazwy znacznie uproszczą późniejsze szukanie błędów oraz ułatwią czytanie kodu po krótkiej przerwie w jego pisaniu.
- Nie bój się funkcji! (wręcz twórz ich jak najwięcej). Funkcje ułatwiają przyrostowe pisanie programu, pozwalają na uniknięcie kopiowania kodu i często zastępują komentarze.
- Komentarze powinny bardziej odpowiadać na pytanie *dlaczego*, a nie *co* robi dany fragment kodu. To co robi fragment programu powinno być oczywiste na podstawie samego kodu i użytych nazw zmiennych i funkcji.
- Funkcje odpowiadające za podobne funkcjonalności wydziel do osobnych plików. Zwiększy to czytelność plików źródłowych.

4 Zadanie — refaktoryzacja

1. Studenci dopierają się w pary.

2. Student z mniejszym numerem indeksu oblicza liczbę według wzoru: `(numer indeksu % 4) + 1` i wybiera odpowiadające jej zadanie z poniższych: `ppr16`, `ppr17`, `ppr20`, `ppr22`.

Jeżeli zadanie nie jest wykonane przez któregoś ze studenta, wybiera on następne z listy.

3. Studenci w parze wymieniają się swoim rozwiązaniem zadania i samodzielnie refaktoryzują kod zadania (rozwiązanego przez drugą osobę w parze), zgodnie ze wskazówkami podanymi na zajęciach.

4. Studenci w parze wymieniają się doświadczeniem i porównują co zmienili i dlaczego? Co było najmniej zrozumiałe w cudzym kodzie? Jakie funkcje i komentarze zostały dodane, aby zwiększyć czytelność kodu?

5. Trzy ochotnicze pary omawiają swoją refaktoryzację przed grupą.