1. Wstęp

> Programowanie jest sztuką konstruktywną. W jaki sposób należy uczyć się konstruktywnej i twórczej działalności? Jedną z metod jest wyabstrahowanie na podstawie wielu przykładów, zespołu elementarnych zasad tworzenia i przekazanie ich w systematyczny sposób. Jednakże programowanie jest dziedziną wielce zróżnicowaną, często wymagającą złożonej działalności intelektualnej. Przypuszczenie, że kiedykolwiek można by jego naukę skondensować w postaci ścisłych recept wydaje się niesłuszne.
>
> Wirth, 1989

> Nie istnieją żadne absolutne reguły stylu programowania tak samo jak stylu pisania. ponieważ programowanie jest częściowo sztuką, a częściowo nauką.
>
> McCracken, Salmon, 1987

2. Czym jest programowanie?

Programowaniem nazywamy ciąg wszystkich czynności związanych z rozwiązywaniem problemów przy użyciu komputera.

Programowanie, wbrew częstej opinii, nie jest samym kodowaniem, a więc procesem pisania instrukcji w jakimś języku wyższego poziomu, zrozumiałym przez komputer. Kodowanie poprzedzone jest przez, ogromną często, ilość przygotowań. Są to:

1. dokładne zdefiniowanie zadania
2. wyjaśnianie i usunięcie nieokreśloności w sformułowaniu problemu
3. wybór sposobu rozwiązania problemu
4. naszkicowanie rozwiązania w zrozumiałym zapisie – schemat rozwiązania
5. kodowanie (proces, tak naprawdę, mało twórczy) –' mechaniczny przekład rozwiązania problemu na poprawne gramatycznie zdania pewnego szczególnego języka (np. PASCALA).
6. wyłapanie i usunięcie błędów z napisanego programu – odpluskwianie
7. napisanie i uzupełnienie dokumentacji programu
8. dokładne sprawdzenie (wytestowanie) programu

Programowanie komputerów jest wyjątkowo złożonym zadaniem, przebiegającym w wielu oddzielnych fazach, z których każda jest tak samo ważna i każda w jednakowym stopniu przyczynia się do rozwiązania problemu

2.1. Faza definiowania problemu

Częsty błąd programistów polega na tym, że nie całkiem dokładnie definiują zadania programistyczne. Niejasności, które powstają w pierwszej fazie, fazie w formułowania problemu, przenoszą się na cały proces prac programistycznych.

2.2. Szkic rozwiązania

Z bardzo niewielkimi wyjątkami naprawdę prostych zadań programistycznych, program składa się najczęściej z wielu oddzielonych ale powiązanych ze sobę zadań. Przykładowo, program obsługi skomplikowanej listy płac może być zbudowany z wielu części, które kolejno sprawdzają dane, porządkują i łączą różnego rodzaju spisy, obliczają i drukują czeki płatnicze, drukują rapporty itp. Ważne jest by precyzyjnie określić zadania dla każdej z tych części programu oraz podać związki między tymi częściami. Zapewnia to, że wykonanie każdej z tych części z osobna doprowadzi do prawidłowo pracującej całości.

Tablica 1: JĘZYKI PROGRAMOWANIA I ICH CECHY

FORTRAN	1957	num. mat., stat., fiz.
ALGOL	1960	num.
COBOL	1960	biznes
LISP	1961	listy, symb.
SNOBOL	1962	łańcuchy, edytory, bibl.
BASIC	1965	interaktywny, szkolny
PL1	1965	złożony, uniwers.
APL	1967	operatory
Pascal	1971	og., nauka prog.
ADA	1980	([1])
Modula		
C, C++	po 1972	og.
ASM	zawsze	szczeg.
Mathematica	po 1959	mat., og., symb.
Maple		
Macsyma		M^3!
HTML(?)		
Perl		
SQL		

2.3. Wybór i reprezentacja algorytmu

Właśnie zostały określone różne zadania i podzadania potrzebne do rozwiązania problemu. Wiemy już jakie informacje muszą zostać dostarczone. Nie wiemy jednak jak program ma rozwiązać postawione zadanie. **Algorytm** jest szczególnym sposobem rozwiązania problemu. Może to byś sposób już znany, dostępny w literaturze, bibliotekach algorytmów lub algorytm opracowany przez programistę. Z pewnych powodów dobrze jest zapisać rozwiązanie w pewnym schemacie, a nie od razu w jakimś istniejącym języku programowania komputera. Jest to pośredni krok, któremu poświęcimy więcej czasu w późniejszych częściach wykładu. Zapis schematyczny jest niezależny ani od typu komputera ani od jakiegokolwiek konkretnego języka programowania.

2.4. Kodowanie

Po bezbłędnym (prawie niemożliwe) sformułowaniu problemu, zorganizowaniu rozwiązania, naszkicowaniu krok po kroku co powinno być wykonane, można rozważyć rozpoczęcie procesu kodowania.

Wybór języka programowania będzie prawdopodobnie zależał od następujących rzeczy:

— natura problemu
— dostępność języków programowania na danej maszynie
— ograniczenia związane z konkretną instalacją komputera

Niektóre języki programowania są językami bardziej uniwersalnymi, pewne z nich są zbudowane do bardziej specjalnych celów. Wiele języków jest szeroko dostępnych, ale niektóre mogą być dostępne tylko na pewnych typach komputerów.

2.5. Faza sprawdzania – testowanie

Otrzymanie newet konkretnych wyników obliczeń nie kończy pracy. Musimy być ich pewni. Należy sprawdzić, że wyniki jakie daje nasz program są dobre w każdym wypadku, nawet takim, który nie był sprawdzony jawnie. Należe w tym celu

wybrać takie dane, dla których potrafimy przewidzieć wyniki i dokonać wszelkich możliwych testów programu.

2.6. Dokumentacja

Proces dokumentowania programu jest zajęciem ciągłym. Dokumentację programu tworzą wszystkie materiały z etapów poprzednich, a więc
— szczegółowy opis problemu
— przedstawienie algorytmiczne zadania
— program jako taki
Dokumentacja powinna zawierać zarówno informacje **dla użytkownika** jak i **dla programistów**, którzy ewentualnie w przyszłości będą dokonywać zmian w kodzie programu (bywa tak, że jest ta sama osoba!).

2.7. Konserwacja i dostosowywanie

Program nie jest czymś stałym. Jest to narzędzie, które należy poprawiać (błędy w różnych fazach jego powstawania), dostosowywać do aktualnej sytuacji: różnice w konfiguracji sprzętu komputerowego, różne urządzenia wejścia-wyjścia, różnice w realizacji języków programowania dla różnych maszyn, itp.

Starsze realizacje (wersje) programów, które są uaktualniane, należy uzupełnić o nowe informacje w ich dokumentacji. Każda interwencja w gotowy program powinna zakończyć się dokładną informacją na ten temat.

Tylko dobrze zorganizowane programy, z przejrzystą o nich dokumentacją, mają szanse na przetrwanie.

2.8. Podsumowanie

Z tego co zostało powiedziane wynika, że programowanie nie jest prostym, liniowym procesem w wyniku którego zawsze powstaje dobry program. Wiele kroków podczas jego realizacji pokrywa się, inne są ciągle powtarzane. Przykładem może być proces odpluskwiania czyli proces wychwytywania i usuwania wszelkiego typu błędów, które zostały popełnione na różnych etapach programowania (po angielsku debuging), proces poprawiania dokumentacji, itp.

Szybko można nauczyć się kodowania, czyli przepisywania programu w języku zrozumiałym dla komputera, ale być dobrym programistą oznacza o wiele więcej. Jest to bowiem zrozumienie i stosowanie reguł z całego widma zagadnień związanych z programowaniem. Wymaga to aktywnej nauki i długiej, często wieloletniej, praktyki.

2.9. Czas

Można w przybliżeniu podać przybliżony czas (w procentach) potrzebny na wykonanie poszczególnych kroków, opisanych wyżej.

— Definiowanie, analiza wymagań użytkownika: **10%**
— Określenie funkcjonalności (analiza zasobów systemowych, szkic dokumentacji, przewodnik użytkownika (pomaga w całym procesie pisania programu i jego konserwacji), analiza możliwych błędów i opis jak na nie należy reagować, itd: **30%**
— Projektowanie programu (wybór typów danych, algorytmu, podział programu na fragmenty, itd.: **20%**
— Kodowanie lub wykonywanie programu (zapisywanie projektu w języku wyższego poziomu (PASCAL, C++, itd.) **15%**

— Sprawdzanie poprawności (kompilacja, usuwanie błędów, wykonanie obliczeń z danymi, które prowadzą do znanych wyników, z danymi błędnymi, przypadkowymi i rzeczywistymi): **15%**
— Instalacja (umieszczenie programu na komputerze klienta, innym niż ten, na którym został przygotowany, (może to być komputer asystenta, który zalicza ćwiczenia), szkolenie personelu obsługującego program itd.): **10%**
— Konserwacja programu (usługi gwarancyjne, inne błędy, usuwanie problemów związanych z nowym sprzętem itd.) Ocenia się, że stanowi to 50–90% całości czasu poświęconego projektowi.

Powyższe dane zostały zaczerpnięte z książki McCrackena i Salmona.

Proszę zwrócić uwagę **jak mało czasu poświęca się średnio na samo kodowanie**, a więc na to co błędnie interpretujemy często jako programowanie.

2.10. Co jest ważne w procesie programowania?

— Przewodnik użytkownika; komentarze. Jest on potrzebny w całym procesie pisania programu i po jego uruchomieniu.
— Wczesne wykrywanie błędów. Wykrycie błędu po zainstalowaniu programu może kosztować o wiele więcej niż budowa całego programu. Przykładem, z dziedziny nauki, jest teleskop Hubble'a, który naprawiano w przestrzeni kosmicznej, z powodu nieprzewidzianych wcześniej zjawisk.
— Proces budowy programu jest procesem cyklicznym. Zawsze powtarza się wcześniej wykonane kroki aby przejść do następnej fazy.
— Czas tworzenia programów zależy od stopnia komplikacji problemu i często jest długi. Warto czynić pisemne uwagi o postępach w budowaniu programu. Pomaga to szybciej wrócić do problemu i przypomnieć o tym co jeszcze należy zrobić. Pamięć ludzka zawodzi. Dokumentacja jest rzeczą podstawową.

3. Charakterystyka dobrych programów

Oto kilka cech uważanych za takie, które charakteryzują dobry program.
— Poprawność (zgodność z wymogami użytkownika)
— Niezawodność (dobre dane wejściowe -> dobre wyniki)
— Przenośność (Łatwość instalacji na różnych komputerach)
— Łatwość konserwacji (Prosto napisany program łatwo przystosować do różnych warunków pracy)
— Czytelność (Prostota jest jedną z najważniejszych cech dobrych programów)
— Prawidłowe wykorzystanie zasobów (pamięci, dyski, itp), szybkość

3.1. Styl programowania

Rozdział ten wymaga znajomości procesu programowania jak też znajomości języka wyższego poziomu i może być opuszczony przy pierwszym czytaniu. Rozważania są oparte na pracy McCrackena i Salmona z 1987 roku.

Styl programowania jest często indywidualną sprawą programisty, ale powinien też spełniać pewne reguły ogóle. Dotyczą one następujących spraw.
— Nazwy zmiennych w programie.
— Wielkości stałe.
— Liczba zmiennych globalnych, dostępnych wszędzie w programie.
— Deklaracje typów zmiennych.
— Odstępy, puste linie, akapity.
— Inteligentne komentarze.
— Podział na sekcje, podprogramy.

Postaram się kolejno omówić zasygnalizowane problemy.

Nazwy zmiennych w programie. Dobrze jest stosować **rzeczowniki** dla oznaczania nazw wszelkich danych, **czasowniki** dla funkcji i procedur (rozkazy, np. dziel, dodaj itp), **przymiotniki** dla zmiennych logicznych (w PASCALu **boolean**).

Jednolite nazwy są mało czytelne. Najlepiej jest używać nazw, które coś oznaczają, powiadamiając tym samym użytkownika (intencjonalnie) o możliwościach funkcji czy procedury, czy też wskazując sens zmiennych, itd. Nazwy bardzo długie, chociaż mogą być komunikatywne, są mało praktyczne jeśli nie stosujemy wyróżnień takich jak duże litery czy podkreślenie (*underscore*) np. nazwa `dodajdwieliczby` jest mniej czytelna od nazwy `DodajDwieLiczby` lub od nazwy `dodaj_dwie_liczby`.

Wielkości stałe. Z wyjątkiem zera, jedynki i może jeszcze kilku innych stałych, nie należy *wprost* używać stałych w całej treści programu, w jego wnętrzu. Najlepiej zadeklarować (zgłosić) je w części opisowej, specjalnie do tego przeznaczonej, np. za pomocą słowa PASCALa **const**. Zmiana sprzętu często powoduje idące za nią zmiany przybliżonych wartości różnych stałych. Ile czasu zmarnujemy wyszukując w programie miejsca, w których występuje stała π zapisana jako liczba zmiennoprzecinkowa z dokładnością do 9 miejsc znaczących by zamienić ją na liczbę o 12 miejscach znaczących. Nie używajmy gołych stałych!

Powiedzmy, że w programie używamy tablicy, której rozmiar 100 zgłosiliśmy kilkakrotnie pisząc między innymi ciąg znaków `[1..100]`. Załóżmy, że zmienimy wymiar 100 wszędzie, automatycznie, za pomocą edytora tekstu, na 200, a przy okazji, całkiem nieświadomie, zamienimy temperaturę jakiegoś procesu też na 200, ale stopni. Co się wtedy stanie?

Liczba zmiennych globalnych. Należy minimalizować liczbę zmiennych globalnych, dostępnych z całego programu. Zmiany w procedurach czy funkcjach, których jest wiele w programach, są zmianami lokalnymi. Często zapominamy przy nich o zmiennych globalnych. W ten sposób łatwo o błędy.

Deklaracje typów zmiennych. Wszystko, co tylko się da, należy umieszczać w deklaracjach `type`, włączając w to podzakresy, definicje tablic oraz rekordów.

Odstępy, puste linie, akapity. Należy używać odstępów, pustych linii i akapitów. Puste linie (jedna, dwie) POWINNY! oddzielać różne fragmenty programu i podprogramy. Akapitów używamy w instrukcjach złożonych. Należy do dobrego tonu podpatrzenie jak radzą sobie z tym inni. Decydując się na jakiś styl trzymajmy się go zawsze. Nie mieszajmy różnych stylów. Programy tracą wtedy na czytelności. Z czasem praktyka pokaże, że trzymanie się jednego stylu opłaca się. Tymczasem należy się do tego przyzwyczajać.

Inteligentne komentarze. Nie należy pisać komentarzy w miejscach, które są oczywiste i jasne. Przesadne używanie komentarzy może oznaczać, że programista nie bardzo wiedział co czyni. Np. komentarz *{Tutaj pod zmienną* x *podstawiamy 7 }*, uczyniony w miejscu $x := 7$ jest niepoprawne.

Podział na sekcje, podprogramy. Program zbudowany jest z części: funkcje i procedury, z których każda ma konkretne zadanie. Zadanie dzielimy na kawałki i następnie proste, zrozumiałe części programu łączymy razem. Jest to realizacja zasady *dziel i rządź*. Proces ten zwie się też *modularyzacją* (część to moduł). Należy zadbać o spójność modułów. McCracken i Salmon piszą, że jeśli uda się znaleźć słowo określające to co robi dany moduł, to właśnie ten fakt oznacza, że funkcja ta jest spójna, koherentna.

Drugą cechą modularyzacji jest przejrzystość komunikacji (lub transmisji). Polega to na tym, że moduły mogą w prosty sposób komunikować się między sobą, używając przy tym minimalnej liczby zmiennych globalnych.

Powtórzmy.

> There is no absolute rules of programming stylej any more than there can be absolute rules of style for any other kind of writing, since programming is partly an art and partly a science
>
> McCracken, Salmon, 1987