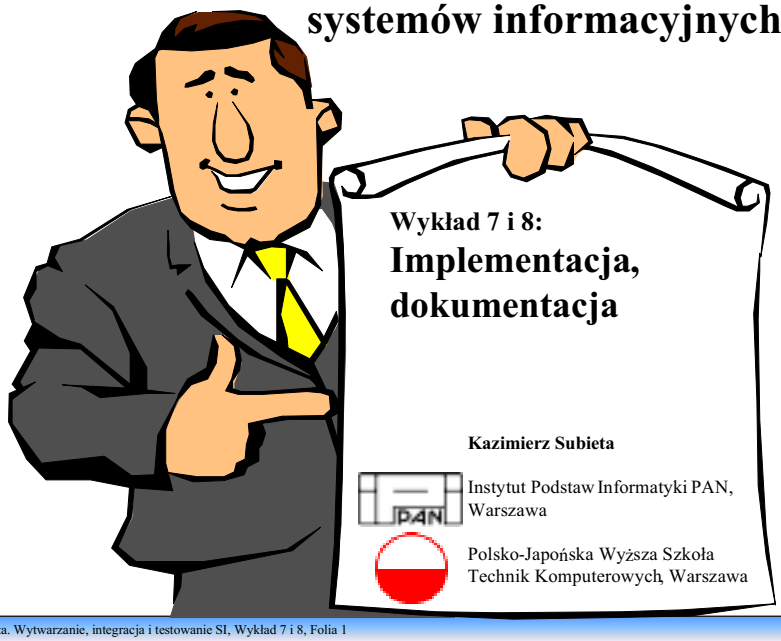
Wytwarzanie, integracja i testowanie systemów informacyjnych



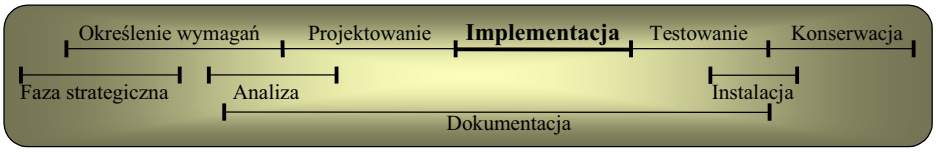
Wykład 7 i 8:
Implementacja, dokumentacja

Kazimierz Subieta

Instytut Podstaw Informatyki PAN, Warszawa

Polsko-Japońska Wyższa Szkoła Technik Komputerowych, Warszawa

Faza implementacji



Faza implementacji uległa w ostatnich latach znaczącej automatyzacji wynikającej ze stosowania:

- Języków wysokiego poziomu
- Gotowych elementów
- Narzędzi szybkiego wytwarzania aplikacji - RAD (*Rapid Application Development*)
- Generatorów kodu

Generatory kodu są składowymi narzędzi CASE, które na podstawie opisu projektu automatycznie tworzą kod programu (lub jego szkielet).

Programowanie niezawodnego oprogramowania

Znaczenie niezawodności:

- Rosnące oczekiwania klientów wynikające m.in. z wysokiej niezawodności sprzętu. Niemniej nadal niezawodność oprogramowania znacznie ustępuje niezawodności sprzętu. Jest to prawdowpodobnie nieuchronne ze względu na znacznie mniejszą powtarzalność oprogramowania i stopień jego złożoności.

- Potencjalnie duże koszty błędnych wykonań, wysokie straty finansowe wynikające z błędnego działania funkcji oprogramowania, nawet zagrożenie dla życia.

- Nieprzewidywalność efektów oraz trudność usunięcia błędów w oprogramowaniu. Często pojawia się konieczność znalezienia kompromisu pomiędzy efektywnością i niezawodnością. Łatwiej jednak pokonać problemy zbyt małej efektywności niż zbyt małej niezawodności.

Zwiększenie niezawodnośći oprogramowania można uzyskać dzięki:
- unikaniu błędów
- tolerancji błędów

Unikanie błędów

Pełne uniknięcie błędów w oprogramowaniu nie jest możliwe.

Można znacznie zmniejszyć prawdopodobieństwo wystąpienia błędu stosując następujące zalecenia:

- Unikanie **niebezpiecznych technik** (np. programowanie poprzez wskaźniki)
- Stosowanie zasady **ograniczonego dostępu** (reguły zakresu, hermetyzacja, podział pamięci, itd.)
- Zastosowanie języków z **mocną kontrolą typów** i kompilatorów sprawdzających zgodność typów
- Stosowanie języków o **wyższym poziomie abstrakcji**
- Dokładne i konsekwentne specyfikowanie **interfejsów pomiędzy modułami** oprogramowania
- Szczególna uwaga na **sytuacje skrajne** (puste zbiory, pętle z zerową ilością obiegów, wartości zerowe, niezainicjowane zmienne, itd.)
- Wykorzystanie **gotowych komponentów** (np. gotowych bibliotek procedur lub klas) raczej niż pisanie nowych (ponowne użycie, *reuse*)
- Minimalizacja różnic pomiędzy modelem pojęciowym i modelem implementacyjnym

Niebezpieczne techniki (1)

- **Instrukcja *go to*** (skocz do) prowadząca do pogramów, których działanie jest trudne do zrozumienia.
- **Stosowanie liczb ze zmiennym przecinkiem**, których dokładność jest ograniczona i może być przyczyną nieoczekiwanych błędów.
- **Wskaźniki i arytmetyka wskaźników**: technika wyjatkowo niebezpieczna, dająca możliwość dowolnej penetracji całej pamięci operacyjnej i dowolnych nieoczekiwanych zmian w tej pamięci.
- **Obliczenia równoległe**. Prowadzą do złożonych zależniości czasowych i tzw. pogoni (zależności wyniku od losowego faktu, który z procesów szybciej dojdzie do pewnego punktu w obliczeniach). Bardzo trudne do testowania. Modne ***wątki*** są bardzo niebezpieczne i określane są jako *zatrute jabłko*.
- **Przerwania i wyjątki**. Technika ta wprowadza pewien rodzaj równoległości, powoduje problemy j/w. Dodatkowo, ryzyko zawieszenia programu.

Niebezpieczne techniki (2)

- **Rekursja**. Trudna do zrozumienia, utrudnia śledzenie programu, może losowo powodować przepełnienie stosu wołań (*call stack*)
- **Dynamiczna alokacja pamięci** bez zapewnienia automatycznego mechanizmu odzyskiwania nieużytków (*garbage collection*). Powoduje "wyciekanie pamięci" i w efekcie, coraz wolniejsze działanie i zawieszenie programu.
- Procedury i funkcje, które realizują wyraźnie odmienne zadania w zależności od parametrów lub stanu zewnętrznych zmiennych.
- **Niewyspecyfikowane, nieoczeekiwane efekty uboczne** funkcji i procedur.
- **Złożone wyrażenia bez form nawiasowych**: korzystanie z priorytetu operatorów, który zwykle jest trudny do skontrolowania przez programistę.
- **Akcje na danych przez wiele procesów** bez zapewnienia mechanizmu synchronizacji (blokowania, transakcji).

Niektóre z tych technik są bardzo przydatne, trzeba je jednak stosować ostrożnie.

Zasada ograniczonego dostępu

Zasada bezpieczeństwa: dostęp do czegokolwiek powinien być ograniczony tylko do tego, co jest niezbędne. Wszystko, co ***może być*** ukryte, ***powinno być*** być ukryte.

Programista nie powinien mieć ***żadnej*** możliwości operowanie na tej części oprogramowania lub zasobów komputera, które nie dotyczą tego, co aktualnie robi.

Perspektywa (prawa dostępu) programisty powinny być ograniczone tylko do tego kawałka, który aktualnie programuje.

Typowe języki programowania niestety nie w pełni realizują te zasady.
Dotyczy to również systemów operacyjnych takich jak DOS lub Windows.
Trzeba powiedzieć, że jest w tym zakresie wykonano krok w tył w stosunku np.
do takich systemów operacyjnych jak George 3.

Zasadę tę realizuje hermetyzacja (znana np. z Modula 2) oraz obiektowość:
- **prywatne** pola, zmienne, metody
- **listy eksportowe**
- **listy importowe** (określające zasoby wykorzystywane z zewnętrznych modułów)

Mocna kontrola typu (1)

Typ jest przypisany do zmiennej, wyrażenia lub innego bytu programistycznego (danej, obiektu, funkcji, procedury, operacji, metody, parametru procedury, modułu, ADT, wyjątku, zdarzenia). Specyfikuje on rodzaj **wartości**, które może przybierać ten byt, lub „zewnętrzne" cechy (interfejs). Typ jest formalnym ograniczeniem narzuconym na **budowę** bytu programistycznego (lub jego specyfikację parametrów i wyniku). Jest to również ograniczenie **kontekstu**, w którym odwołanie do tego bytu może być użyte w programie.

Zasadniczym celem typów jest wspomaganie kontroli poprawności programu. W językach z tzw. mocnym typowaniem (*strong typing*), np. Pascalu i Modula-2, każdy deklarowany byt programistyczny musi być obowiązkowo wyposażony w deklarację typu. Poprzez tę deklarację programista wyraża swoje oczekiwania co do roli tego bytu w programie. Te oczekiwania są następnie sprawdzane. Np. określając typ zmiennej X jako *integer* programista ustala, że ta zmienna ma przechowywać wartości całkowite. Dzięki temu możliwe jest sprawdzenie, czy wszystkie odwołania do tej zmiennej w programie mają kontekst, w jakim może być użyta wartość całkowita.

Mocna typologiczna kontrola poprawności programów okazała się cechą skutecznie eliminującą błędy popełniane przez programistów. Według typowych szacunków, po wyeliminowaniu błędów syntaktycznych programu około 80% pozostałych błędów jest wychwytywane przez mocną kontrolę typu. Niestety, w wielu produktach komercyjnych taka kontrola jest całkowicie zaniedbywana lub występuje w postaci szczątkowej (Smalltalk, SQL).

Mocna kontrola typu (2)

System mocnej statycznej kontroli typu zawiera następujące elementy:

- **Specyfikacja typów wszystkich zmiennych i obiektów** , które występują w programie, np.:
  `typedef TypPrac=struct{ string nazwisko, int zarobek, Dzia³ pracuje_w, int zarobek_netto()};`
  `TypPrac Pracownik;`

- **Specyfikacja sygnatur** wszystkich operatorów, procedur, funkcji, metod, np.
  `boolean sprawdŸ( in TypPrac prac, in TypDzia³ dzial, out int ile_lat_pracuje )`

- **Specyfikacja interfejsów** modułów, klas i innych hermetyzowanych abstrakcji programistycznych.

- **Dla parametrów procedur, funkcji, metod: okre ślenie które z nich są wejściowe** (czyli komunikowane przez wartość, *call-by-value*), a które wyjściowe (czyli komunikowane przez referencję, *call-by-reference*).

- **Określenie reguł wnioskowania o typie** (*type inference rules*) dla wszystkich konstrukcji składniowych języka programowania, szczególnie dla wyrażeń. W procesie analizy gramatycznej następuje ustalenie jaki będzie wynikowy typ każdej konstrukcji, która w tym programie występuje.

W procesie wnioskowania o poprawności typologicznej ustalone są między innymi typy rzeczywistych parametrów aktualnych poszczególnych wywołań procedur, metod, etc. Sprawdzane jest także, czy nie ma odwołań do bytów, które nie są zadeklarowane lub sa niedostępne. Niezgodności są sygnalizowane jako błędy typu.

Tolerancja błędów

Żadna technika nie gwarantuje uzyskania programuw pełni bezbłędnego.
Tolerancja błędów oznacza, że program działa poprawnie, a przynajmniej sensownie także wtedy, kiedy zawiera błędy.

Tolerancja błedów oznacza wykonanie przezprogram następujących zadań.

- Wykrycia błędu.
- Wyjścia zbłędu, tj. poprawnego zakończenie pracy modułu w którym wystąpił błąd.
- Ewentualnej naprawy błędu, tj. zmiany programu tak, aby zlikwidować wykryty błąd.

Istotne jest podanie dokładnej diagnostyki błędu, ewentualnie, z dokładnością do linii kodu źródłowego, w której nastąpił błąd.

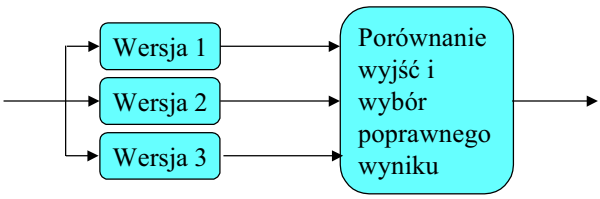
Istnieją dwa główne sposoby automatycznego wykrywania błędów:

- sprawdzanie warunków poprawności danych (tzw. asercje). Sposób polega na wprowadzaniu dodatkowych warunków na wartości wyliczanych danych, które są sprawdzane przez dodatkowe fragmenty kodu

- porównywanie wyników różnych wersji modułu.

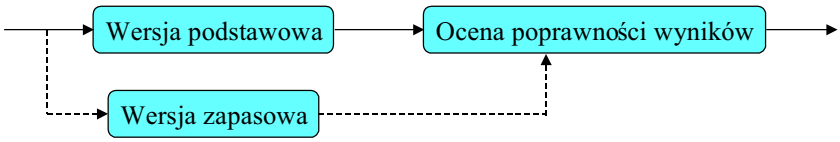
Porównywanie wyników różnych wersji modułu

Programowanie N-wersyjne: ten sam moduł zaprogramowany przez niezależne zespoły programistów. Wersje modułu działają równolegle, wyniki są porównywane:



Istotne są narzuty na czas wykonania.

Programowanie z modułem zapasowym:



Po wykryciu błędnego wyniku uruchamia się wersję zapasową.

Transakcja: jednostka działalności systemu

Transakcja umożliwia powrót do stanu sprzed rozpoczęcia jej działania po wystąpieniu dowolnego błędu. Jest to podstawowa technika zwiększenia niezawodności oprogramowania działającego na bazie danych.
Jednocześnie, transakcje zapewniają spójność wielodostępu do bazy danych.

Własności transakcji: **ACID**

A **Atomowość** (*atomicity*) - w ramach jednej transakcji wykonują się albo wszystkie operacje, albo żadna

C **Spójność** (*consistency*) - o ile transakcja zastała bazę danych w spójnym stanie, po jej zakończeniu stan jest również spójny. (W międzyczasie stan może być chwilowo niespójny)

I **Izolacja** (*isolation*) - transakcja nie wie nic o innych transakcjach i nie musi uwzględniać ich działania. Czynności wykonane przez daną transakcję są niewidoczne dla innych transakcji aż do jej zakończenia.

D **Trwałośc** (*durability*) - po zakończeniu transakcji jej skutki sa na trwale zapamiętane (na dysku) i nie mogą być odwrócone przez zdarzenia losowe (np. wyłączenie prądu)

Typowe środowiska implementacyjne - języki proceduralne

Jest to tradycyjne i wciąż popularne środowisko implementacji.

Procesy i moduły wysokiego poziomu mogą odpowiadać całym aplikacjom.

Grupy procedur i funkcji - odpowiadają poszczególnym funkcjom systemu.

Składy/zbiorniki danych w projekcie odpowiadają strukturom danych języka lub struktorom przechowywanym na pliku

Języki proceduralne dają niewielkie możliwości ograniczenia dostępu do danych. Poszczególne składowe struktur są dostępne wtedy, gdy dostępna jest cała struktura.

Istnieją języki o znacznie bardziej rozbudowanych możliwościach ograniczania dostępu do danych, np. Ada, Modula-2.

Istnieje możliwość programowania w stylu obiektowym, ale brakuje udogodnień takich jak klasy, dziedziczenia, metod wirtualnych.

Typowe środowiska implementacyjne - języki obiektowe

Bardzo przydatne jako srodowisko implementacji projektu obiektowego, gdyż odwzorowanie pomiędzy modelem projektowym i implementacyjnym jest proste.

Z reguły jednak są niewystarczające w przypadku duzych zbiorów przetwarzanych danych i wymagają współpracy z bazą danych.

Większość języków obiektowych to języki hybrydowe, powstające w wyniku dołożenia cech obiektowości do języków proceduralnych. Najbardziej klasycznym przypadkiem takiego rozwiązania jest C++.

Istnieją zwolennicy i przeciwnicy takiego podejścia. Jak się wydaje, jedyną zaletą języków hybrydowych jest znacznie łatwiejsze ich wypromowanie przy uzyciu nie do końca prawdziwych (zwykle naciąganych) twierdzeń o “zgodności”, “kompatybilności” i “przenaszalności”.

Tendencja do tworzenia języków hybrydowych słabnie, czego dowodem jest Java, Eiffel, Visual Age i Smalltalk.

Typowe środowiska implementacyjne - relacyjne bazy danych

W tej chwili są to najlepiej rozwinięte środowiska baz danych, pozwalające na zaimplementowanie systemów bazujących na dużych zbiorach danych.

\+
- wielodostęp
- automatyczna weryfikacji więzów integralności
- prawa dostępu dla poszczególnych użytkowników
- wysoka niezawodność
- rozszerzalność (ograniczona)
- możliwość rozproszenia danych
- dostęp na wysokim poziomie (SQL, ODBC, JDBC)

\-
- skomplikowane odzorowanie modelu pojęciowego
- mała efektywność dla pewnych zadań (kaskadowe złączenia)
- ograniczenia w zakresie typów
- brak hermetyzacji i innych cech obiektowości
- zwiększenie długości kodu, który musi napisać programista

Obiektowe bazy danych

Zaletą modelu obiektowego baz danych jest wyższy poziom abstrakcji, który umożliwia zaprojektowanie i zaprogramowanie tej samej aplikacji w sposób bardziej skuteczny, konsekwentny i jednorodny.

Uproszczenie i usystematyzowanie procesu projektowania i programowania, minimalizacja liczby pojęć, zwiększenie poziomu abstrakcji, zmniejszenie dystansu pomiędzy fazami analizy, projektowania i programowania oraz zwiększeniu nacisku na rolę czynnika ludzkiego.

W stosunku do modelu relacyjnego, obiektowość wprowadza więcej pojęć, które wspomagają procesy myślowe zachodzące przy projektowaniu i implementacji.

Filarami na których opiera się model obiektowy baz danych są pojęcia: złożone obiekty, tożsamość, powiązania, klasy i typy, hierarchia (lub inna struktura) dziedziczenia, metody, komunikaty, hermetyzacja, przesłanianie, późne wiązanie, polimorfizm i trwałość.

Obiektowe bazy danych osiągają dojrzałość, ale nie pokonały jeszcze bariery nieufności powszechnego (dużego) klienta.

Obiektowo-relacyjne bazy danych

Sukces obiektowości w zakresie ideologii i koncepcji spowodował wprowadzenie wielu cech obiektowości, takich jak klasy, metody, dziedziczenie, abstrakcyjne typy danych, do systemów relacyjnych. „Częściowa obiektowość" jest wprowadzona do większości systemów relacyjnych znajdujących się na rynku.

Takie podejście jest określane jako „hybrydowe" lub „obiektowo-relacyjne". Ostatnio karierę robi także termin „uniwersalny serwer" (*universal server*), hasło marketingowe eksponujące możliwość zastosowania systemu do przechowywania i przetwarzania obiektów, relacji, danychmultimedialnych, itd. Podstawą ideologiczną systemów obiektowo-relacyjnych jest zachowanie sprawdzonych technologii relacyjnych (np. SQL) i wprowadzanie na ich wierzchołku innych własności, w tym obiektowych.

Systemy obiektowo-relacyjne, powstające w wyniku ostrożnej ewolucji systemów relacyjnych w kierunku obiektowości, liczą na pozycję systemów relacyjnych na rynku i odwołują się do ich wiernej klienteli.

Środowiska programistyczne programów użytkowych

Przykładem może być Microsoft Office, który można przystosować do różnych zastosowań. Np. cechy Microsoft Excel:

- Zawiera pełny proceduralny język Visula Basic dla Aplikacji
- Obejmuje szeroko rozbudowaną bibliotekę obiektów, udostępniającą praktycznie wszystkie możliwości pakietu.
- Pozwala na nagrywanie makrodefinicji w stylu Visual Basic.
- Posiada możliwość dialogowego projektowania interfejsu użytkownika: projektowanie dialogów, menu i pasków narzędziowych, umieszczanie pól dialogowych na arkuszach, definiowanie reakcji na zdarzenia
- Zawiera debugger ułatwiający uruchamianie programów
- Pozwala na dystrybucję aplikacji bez rozprowadzania kodu źródłowego
- Obejmuje rozbudowane możliwości współpracy ze standardami DLL, DDE, OLE, ODBC

Łatwiejsze opracowanie prototypów (montaż z gotowych elementów).

Kluczowe czynniki sukcesu i rezultaty fazy implementacji

Czynniki sukcesu:

Wysoka jakość i wystarczająca szczegółowość projektu
Dobra znajomość środowiska implementacji
Zachowanie standardów
Zwrócenie uwagi na środki eliminacji błędów

Rezultaty:

Poprawiony dokument opisujący wymagania
Poprawiony model analityczny
Poprawiony projekt, który od tej pory stanowi już dokumentację techniczną
Kod składający się z przetestowanych modułów
Raport opisujący testy modułu
Zaprojektowana i dostrojona baza danych
Harmonogram fazy testowania

Narzędzia CASE w fazie implementacji

Programiści mogą bezpośrednio korzystać (przegladać) diagramy i słownik danych korzystając z narzędzia CASE.

Niektóre systemy CASE (lower) dostarczają generatorów kodu, które generują programy lub ich szablony. Typowe elementy kodu:

- skrypty tworzące relacje w bazie danych

- definicje struktur danych

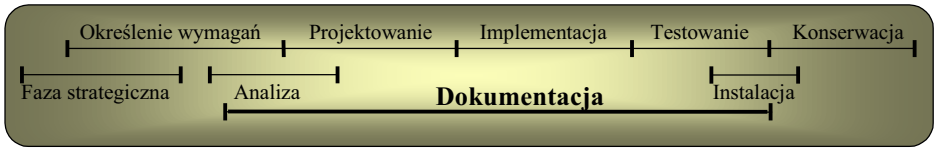
- nagłówki procedur i funkcji

- definicje klas

- nagłówki metod

Kod jest uzupełniany wieloma komentarzami na podstawie informacji ze słownika danych. Niektóre narzędzia CASE umożliwiają sprzęg do narzędzi RAD.

Dokumentacja



Tworzenie dokumentacji jest procesem ciągłym zaczynającym się od określenia wymagań i trwającym aż do fazy instalacji.

Dokumentacja użytkowa:

Użytkownicy końcowy (podręcznik użytkownika)

Administratorzy systemu

Niekiedy potrzebne jest kilka wersji dokumentacji, dla uzytkowników o różnym stopniu zaawansowania.

Podstawowe składowe dokumentacji użytkowej (1)

- **Opis funkcjonalny**: jest to wstępna część dokumentacji, która w zwarty sposób opisuje przeznaczenie i głowne możliwości systemu. Dla osoby rozważającej zakup powinien dostarczyć informacji, czy system spełnia jej potrzeby. Opis jest przydatny dla uzytkownika początkowego, którzy nie znają możliwośći systemu.
  Ta część jest bardzo słabą stroną wielu dokumentacji, gdzie "getting started" zaczyna się od tego, *jak* zainstalować system (tylko nie wiadomo *po co*).

- **Podręcznik użytkownika**: opis głównie przeznaczony dla początkujących. Zawiera:

  • sposoby uruchomienia i kończenia pracy z systemem
  • sposby realizacji najczęściej wykonywanych funkcji
  • metodach obsługi błędów
  • sposobach korzystania z pomocy

Podstawowe składowe dokumentacji użytkowej (2)

- **Prosty przykład korzystania z systemu**.
- **Kompletny opis**: przeznaczony głównie dla doświadczonych użytkowników. Powinien zawierać:

  • Szczegółowy opis wszystkich funkcji systemu
  • Informacje o wszystkich sposobach wywołania tych funkcji
  • Opisy formatów danych
  • Opisy błędów, które mogą pojawić się podczas pracy z systemem
  • Informacje o wszelkich ograniczeniach, np. zakresów danych

- **Opis instalacji**. Jest to składowa dokumentacji przeznaczona głównie dla administratora systemu. Powinien zawierać procedury instalacji oraz procedury dostrojenia systemu do środowiska, w którym system ma pracować.
- **Podręcznik administratora systemu**. Możliwość zmian konfiguracji systemu i sposoby udostępniania systemu użytkownikom końcowym.
- **Słownik używanych terminów, indeks**

Jakość dokumentacji, styl pisania

Na jakość wpływają następujące czynniki:

- **Struktura**. Poszczególne podręczniki powinny być w czytelny i zrozumiały sposób podzielone na rozdziały, podrozdziały i sekcje
- **Zachowanie standardów**: Spójna struktura, forma i sposób pisania
- **Sposób (styl) pisania**, wiele elementów omówionych poniżej:
- **Stosowanie formy aktywnej zwracanie się bezpośrednio do czytelnika, np.**

  Na ekranie pojawi się dialog...
  Aby otworzyć plik należy wybrać z menu...

  (w jezyku polskim ta forma może być jednak nienaturalna) ⇨ Na ekranie zobaczysz dialog...
  Jeśli chcesz otworzyć plik, wybierz z menu...

- **Poprawność gramatyczna, ortograficzna i stylistyczna**. Błędy tego rodzaju obniżają zaufanie uzytkownika do projektantów i pośrednio do całego systemu.

Styl pisania (c.d.)

- **Krótkie zdania**, unikanie zanurzeń (nawiasów), wtrąceń, kilku myśli.
- **Krótkie akapity**. Najlepiej zauważalne są informacje na początku i na koncu akapitu. Informacje w środku długich akapitów mają najmniejsze szanse na zauważenie. Najlepiej, jeżeli pierwsze zdanie akapitu jest wprowadzeniem, a ostatnie zakończeniem pewnego tematu.
- **Oszczędność słów**, przejrzystość i precyzja. Poszczególne fakty nalezy wyrażać w jak najkrótszy sposób. W odróżnieniu od tekstów literackich, synonimy i odległe referencje (zaimki) są niepożądane, gdyż mogą powodować niejasność. Lepsze są powtórzenia (mimo, że piękno tekstu może na tym mocno ucierpieć).
- **Precyzyjna definicja używanych terminów**. Mogą pojawić się terminy, kórych znaczenie nie jest powszechnie znane. Mogą pojawić się terminy znane, ale w zawężonym lub specyficznym znaczeniu. Definicje powinny być zebrane w słowniku.

Styl pisania (c.d.)

- **Powtarzanie trudniejszego lub ważniejszego opisu**. Aby uniknąć odległych skoków, elementy opisu mogą być powtórzone w tych miejscach, w których następuje bezpośrednie odwołanie do tego elementu. Nie można jednak z tym przesadzać.
- **Stosowanie tytułów i podtytułów sekcji, wyliczeń i wyróżnień**. Elementy te ożywiają tekst i pozwalają zwrócić uwagę czytelnika na najistotniejsze elementy. Wyliczenia systematyzują i klasyfikują przekazywaną wiedzę.
- **Zrozumiałe odwołania do innych rozdziałów**. Powinno się podawać nie tylko jego numer, ale także charakterystykę zawartości, np:

  W rozdziale 5.1 znajduje się...

  ➡ W rozdziale 5.1 omawiającym zagadnienia ... znajduje się...
- **Zrozumiałe rysunki i inne ilustracje**. Bardzo ożywiają tekst, skupiają uwagę i poprawiają szybkość rozumienia. Rysunki powinny być zaopatrzone w zrozumiałe podpisy.

Podsumowanie fazy dokumentacji

Kluczowe czynniki sukcesu:

Wysoka jakość definicji wymagań, modelu i projektu
Uwzględnienie róznego stopnia zaawansowania użytkowników

Podstawowe rezultaty fazy dokumentacji

Dokumentacja w formie drukowanej i elektronicznej; pomoce on line.
Często: wykrycie błędów i nieścisłości w fazach definicji wymagań, modelu i projekcie

Narzędzia CASE w fazie dokumentacji

Autorzy dokumentacji korzystają ze wszelkich zasobów informacji zgromadzonych w repozytorium narzędzia CASE: słownika danych, generatorów raportów, diagramów. Niektóre narzędzia CASE posiadają specjalne opcje dla przygotowania dokumentacji