POLSKIE
TOWARZYSTWO
INFORMATYCZNE

ODDZIAŁ GÓRNOŚLĄSKI

ZALECENIA NORMALIZACYJNE STOSOWANE W OKRESIE ISTNIENIA OPROGRAMOWANIA

Przewodnik problemowy

OPRACOWAŁ:

JANUSZ ZALEWSKI

na prawach rękopisu

KATOWICE, WRZESIEŃ 1989

Spis treści

1. WSPÓŁCZESNE POGLĄDY NA JAKOŚĆ OPROGRAMOWANIA

Coraz szersze stosowanie komputerów w różnych dziedzinach gospodarki spowodowało gwałtowny wzrost zainteresowania jakością oprogramowania - szczególnie w tzw. *zastosowaniach krytycznych*, tj. takich, które stanowią zagrożenie dla życia lub zdrowia, ewentualnie mogą spowodować znaczne straty materialne. Wkrótce też zaczęto określać cechy, jakie powinno mieć oprogramowanie, aby można mówić o jego dobrej jakości. Okazało się wszakże, że znacznie łatwiej te cechy wyliczyć niż je bezpośrednio lub pośrednio zmierzyć, co dało początek całej nowej dziedzinie wiedzy informatycznej, tzw. programometrii. Od opublikowania pierwszych prac na temat różnych aspektów mierzenia jakości oprogramowania minęło już kilkanaście lat. Choć ta ważna dziedzina stanowi punkt wyjścia do rozważań nt. jakości oprogramowania, nie będę się nią zajmował, odsyłając czytelników do innych opracowań, np. [13], pełne jej omówienie wymagałoby bowiem oddzielnego wykładu. Zainteresowanym przedmiotem mogę polecić najpełniejszy, jaki znam, spis literatury [41], którego pewną liczbę egzemplarzy przekazuję *Oddziałowi Śląskiemu* do udostępnienia osobom bardziej zainteresowanym. W niniejszym rozdziale przedstawię tylko sposób podejścia do tego problemu.

1.1 Atrybuty jakości oprogramowania

Ciekawe jest przedstawienie przynajmniej niektórych cech mających wpływ na jakość oprogramowania. W nowym wydaniu słownika inżynierii oprogramowania IEEE [19] wyliczono około 30 takich cech: dyspozycyjność, efektywność, elastyczność, integralność, konserwowalność, krytyczność, modularność, nadmiarowość, niesprzeczność, niezawodność, odporność, ogólność,

poprawność, prostota, przenośność, realizowalność (ang. feasibility), rozszerzalność, spoistość (ang. cohesion), śladowalność (ang. traceability), testowalność, tolerowanie błędów i defektów, użyteczność, wielouzywalność (ang. reusability), współdziałanie (ang. interoperability), wydajność, zabezpieczenie (ang. security), zespolenie (ang. coupling), zgodność, złożoność. Pełne definicje każdego z tych pojęć podano w *Dodatku*.

Inna organizacja mająca szczególny tytuł do formułowania zaleceń dotyczących jakości oprogramowania w zastosowaniach krytycznych, Międzynarodowa Agencja Energii Atomowej, wyliczyła również około 30 cech charakterystycznych, świadczących o jego jakości [16], bardzo zbliżonych do podanych przez IEEE (pełne definicje znajdują się również w *Dodatku*): czytelność, dokładność, efektywność, ergonomiczność, integralność, komunikatywność, konserwowalność, modyfikowalność, niesprzeczność, niezależność od urządzeń, niezawodność, odporność, poprawność, prostota, przenośność, rozliczalność (ang. accountability), rozszerzalność, samoopisywalność (ang. self-descriptiveness), samowystarczalność (ang. self-containedness), strukturalność, śladowalność, testowalność, użyteczność, wielouzywalność, współdziałanie, zdolność obsługi błędów, zrozumiałość, zupełność, zwięzłość.

Oczywiście, cechy te w większości, może nawet wszystkie, są niemierzalne, tzn. niewyrażalne w postaci liczbowej. Aby można je określić ilościowo, trzeba skonstruować specjalne miary, tzw. metryki. Według słownika [19] *metryką* nazywa się ilościową miarę stopnia, w jakim system, składowa lub proces posiada określony atrybut (ang. a quantitative measure of the degree to which a system, component or process possesses a given

attribute). Przez *atrybut* rozumie się, jak w języku potocznym, pewną cechę, definiując ten termin ogólnie jako charakterystykę obiektu (ang. a characteristic of an item), a dokładniej, definiując *atrybut jakości* jako cechę lub charakterystykę wpływającą na jakość obiektu (ang. a feature or characteristic that affects item's quality).

Choć znalezienie takich metryk jest trudne, nie jest przecież niemożliwe. Jako przykład można podać bardzo interesujące skontruowanie dwóch metryk mogących służyć za miary dwóch atrybutów z listy IAEA, tj. modyfikowalności i strukturalności, dla programów napisanych w języku Ada [12].

Modyfikowalność oprogramowania czyli łatwość wprowadzania zmian można powiązać bezpośrednio z *odpornością na zmiany*, wyrażoną jako stosunek liczby odwołań do składowych obiektów złożonych o nielokalnych typach danych, do liczby wszystkich wierszy programu. Znaczenie tej metryki w Adzie odpowiada znanemu faktowi, że rozproszenie informacji utrudnia a skoncentrowanie jej w prywatnych częściach pakietów ułatwia modyfikowanie programu.

Strukturalność można z kolei mierzyć za pomocą stopnia odseparowania poszczególnych pakietów, używając do tego cechy zwanej w Adzie *widocznością pakietu*. Przykładowo, jedną z metryk można zdefiniować jako stosunek liczby jednostek programowych, w których informacja z danego pakietu jest używana, do liczby jednostek w których pakiet może być uczyniony widocznym przez użycie klauzuli with. Znaczy to, że im mniejsza będzie wartość tej metryki, tym większe będzie odseparowanie modułów i lepsza strukturalność programu. W cytowanym artykule [12] podano też inną metrykę widoczności, co świadczy o trudnościach we właściwym doborze tych miar.

1.2 Miary jakości według EWICS

W jednym z nowszych opracowań Komitetu Technicznego nr 7 EWICS [45], w części dotyczącej klasyfikacji atrybutów i miar jakości oprogramowania podano przykłady takiej dekompozycji atrybutów jakości, aby można je mierzyć ilościowo. Przykładowo, *użyteczność* rozumianą jako łączny wysiłek wymagany do nauczenia się systemu, eksploatowania go, przygotowywania danych i interpretowania wyników [16], tj. zasób umiejętności koniecznych do użytkowania systemu, proponuje się określić za pomocą kombinacji czterech cech niższego poziomu, takich jak:

- wymagania podstawowe (ang. entry requirement) opisujące minimalne umiejętności użytkownika, np. fizyczne, jak wzrok, słuch, lub mentalne jak inteligencja, wykształcenie, doświadczenie itp.
- konieczność uczenia się (ang. learning requirement), mierzona przede wszystkim czasem potrzebnym do opanowania i kontroli głównych czynności systemu
- zdolność usługowa (ang. handling ability) mierzona czystą produktywnością w przeciągu ustalonego czasu
- upodobanie (ang. likability) określające, w jakim stopniu użytkownicy lubią korzystać z systemu, co można zmierzyć np. częstością zmian miejsca pracy przez personel lub ankietując pracowników.

Inny ważny atrybut oprogramowania, *konserwowalność* rozłożono jeszcze bardziej szczegółowo na łatwiej mierzalne cechy, wyrażalne bezpośrednio w jednostkach czasu, tzn.:

- czas rozpoznania defektu (automatycznie lub przez człowieka)
- opóźnienie administracyjne (np. do chwili uaktywnienia modułu programowego, lokalizującego defekt)
- czas skompletowania narzędzi (np. dokumentacji, analizatorów programowych, zbioru testów, wyników wzorcowych itp.)

- czas analizy defektu, tj. prześledzenia pracy wstecz, od objawów defektu do jego źródeł
- czas formułowania poprawek, tj. czas potrzebny na przełożenie wniosków o przyczynie defektu na propozycję podjęcia odpowiednich czynności
- czas inspekcji poprawek, rozumiany jako czas niezbędny do sprawdzenia hipotezy, np. ze względu na zupełność i niesprzeczność
- czas wprowadzania poprawek
- czas testowania
- czas oceny wyników testu
- czas przywrócenia poprawnej pracy.

Oczywiście, nie wszystkie te cechy muszą wchodzić w skład miary konserwowalności, ponieważ nie wszystkie zależą od samego oprogramowania - część z nich zależy w dużym stopniu od otoczenia.

1.3 Podejście praktyczne

Jeszcze inne podejście, może nawet bardziej płodne - inżynierskie, zaprezentowano w przewodniku [50], nie poprzestając na czystym wyliczeniu atrybutów jakości oprogramowania lecz proponując ich miary. Atrybuty wysokiego poziomu nazwano *czynnikami* (ang. factors) wpływającymi na jakość, a określające je atrybuty niższego poziomu - *kryteriami*, definiując ich łącznie również około trzydziestu (por. *Dodatek*).

Przykładowo, *niezawodność* jest określona przez tolerowanie błędów, niesprzeczność, dokładność i prostotę, a *konserwowalność* - przez niesprzeczność, prostotę, zwięzłość, modularność i samoopisywalność. Inne atrybuty (czynniki jakości)

są określone podobnie, np.:

o *wielokrotnego użytku* opisuje się za pomocą ogólności, modularności, niezależności i samoopisywalności

o *elastyczność* jest zapewniana przez modularność, ogólność, rozszerzalność i samoopisywalność

o *testowalność* określa się przez prostotę, modularność, sondowalność i samoopisywalność.

Niewiele wszakże mogłoby z takiego wyliczania wynikać, gdyby atrybuty niższego poziomu nie były możliwe do zmierzenia. Przykładowo, najczęściej występująca samoopisywalność może być mierzona następującymi miarami lub ich kombinacją:

1) stosunkiem liczby niepustych komentarzy do całkowitej liczby wierszy w module

2) efektywnością komentarzy, rozumianą jako stosunek liczby modułów spełniających określoną regułę do całkowitej liczby modułów (do przykładowych reguł mogą należeć: posiadanie standardowego nagłówka, jednolite wyróżnienie komentarzy w kodzie, opatrzenie komentarzem każdego przekazania sterowania z podaniem źródła i przeznaczenia, itd.)

3) opisywalnością języka implementacji, rozumianą jak wyżej, przy istnieniu takich reguł, jak np. nadawanie znaczących mnemonicznych nazw zmiennym, stosowanie wcięć tekstu programu, brak nazw identycznych z identyfikatorami słów kluczowych itp.

Podobnie często występujący atrybut - *prostota* - jest określony przez następujące miary lub ich kombinację:

1) dobroć struktury programu, mierzona jak wyżej, przy uwzględnieniu takich reguł, jak: brak dublowania funkcji, niezależność modułów, brak danych globalnych, podanie w opisie modułu jego ograniczeń, jednowejściowość i jednowyjściowość modułu itp.

2) strukturalność programu, rozumiana jako użycie strukturalnego języka programowania lub preprocesora

3) brak złożoności (mierzonej miarą Halsteada)

4) jasność kodu, dzieląca się na kilka jeszcze bardziej szczegółowych miar, np.

a) nasycenie operacjami logicznymi, rozumiane jako stosunek liczby negatywnych lub złożonych wyrażeń logicznych do całkowitej liczby instrukcji wykonywalnych

b) liczba skoków do wnętrza i na zewnątrz pętli (względem całkowitej liczby pętli)

c) względna liczba pętli z modyfikacją indeksu (tj. zmiennej sterującej)

d) względna liczba instrukcji etykietowanych

e) maksymalny poziom zagnieżdżania

f) względna liczba rozgałęzień lub skoków

g) gęstość zmiennych, tj. stosunek liczby zmiennych do całkowitej liczby instrukcji wykonywalnych.

Rzadziej występujące atrybuty, jak tolerowanie błędów, sondowalność, zwięzłość, są również mierzalne w podobny sposób. Nie trudno jednak zauważyć, że wiele z tych atrybutów jest dość trudno mierzalnych, a miary innych mają małą wiarygodność.

1.4 Strategie postępowania

Jak widać, bezpośrednie i pośrednie wyrażanie jakości oprogramowania w sposób ilościowy nie jest łatwe. Dlatego dobrą jakość produktów próbuje się zapewnić głównie drogą narzucenia pewnych rygorów na proces wytwarzania oprogramowania. Z dzisiejszej perspektywy widoczne są w zasadzie dwie drogi prowadzące do tego celu.

Ponieważ do niedawna przez programowanie rozumiano

wyłącznie kodowanie, *pierwsze wysiłki* w kierunku polepszenia jakości oprogramowania skierowano w stronę języków programowania. Ta tendencja doprowadziła niewątpliwie do rozwoju języków, tj. udoskonalenia ich konstrukcji w celu polepszenia jakości programów, czego wyrazem są niewątpliwie takie języki, jak Ada czy Modula 2, oraz - co równie ważne - do unowocześnienia metod (szczególnie metod formalnych) i narzędzi programowania (włącznie z tworzeniem tzw. środowisk programistycznych). Ten obszar wiedzy informatycznej jest dziś bardzo rozległy - o czym świadczy liczba stosowanych i wciąż powstających, nowych języków, metod i narzędzi, i nie będzie tu omawiany.

Równocześnie jednak zaczęto zdawać sobie sprawę, że procesu programowania nie należy ograniczać do kodowania lecz można i trzeba utożsamiać go z wytwarzaniem produktu programowego, w sposób analogiczny do technologii wytwarzania wyrobów w innych dziedzinach przemysłu - z podziałem na poszczególne etapy, fazy lub operacje. Podział procesu produkcji oprogramowania na szereg wyodrębnionych i dobrze rozróżnialnych faz, ze ściśle określonymi przejściami, ma służyć właśnie ulepszeniu końcowego produktu pod względem jakości, wskutek właściwego przestrzegania zasad produkcyjnych. Choć możliwych i stosowanych jest wiele modeli tzw. cyklu produkcyjnego oprogramowania (ang. software development cycle), szczególną popularność zdobywają zasady wypracowane przez komitet techniczny ds. inżynierii oprogramowania Towarzystwa Komputerowego amerykańskiej organizacji inżynierów elektryków i elektroników (IEEE Computer Society, Technical Committee on Software Engineering), ogłoszone w większości w postaci różnego rodzaju dokumentów normalizacyjnych.

1.5 Okres istnienia oprogramowania według norm IEEE

Zbiór norm amerykańskich dotyczących inżynierii oprogramowania, opracowanych przez IEEE Computer Society, Technical Committee on Software Engineering, składa się z kilkunastu dokumentów.

Według definicji podanej w [19] *okres istnienia oprogramowania* (ang. software life cycle) jest to odcinek czasu od powzięcia decyzji o wytworzeniu produktu programowego do momentu, gdy ten produkt przestanie być dostępny. Jest oczywiste, że tak długi odcinek czasu musi dzielić się na krótsze tzw. fazy. Podział całego okresu istnienia na fazy może wyglądać bardzo różnie - zależy to od rodzaju produktu. Przykładowo, w tzw. *modelu kaskadowym* (ang. waterfall model), według normy [19] może on dzielić się na analizę wymagań, projektowanie wstępne i szczegółowe, kodowanie, integrowanie modułów, testowanie, dokumentowanie, dostawę, eksploatację, konserwowanie i wycofanie. W bardziej szczegółowym podziale, przygotowywanym w postaci normy [35], dzieli się cały okres istnienia na procesy, wyróżniając kilka grup procesów:

- procesy zarządzania przedsięwzięciem
- procesy przedprodukcyjne
- procesy produkcyjne
- procesy poprodukcyjne
- procesy integracyjne.

Procesy zarządzania i procesy integracyjne trwają przez całe przedsięwzięcie, natomiast właściwe procesy produkcyjne następują po sobie kolejno w czasie. Każdy z procesów jest realizowany przez wykonanie szeregu ściśle określonych czynności (ang. activities).

Grupa *zarządzania przedsięwzięciem* składa się z trzech procesów: zainicjowania przedsięwzięcia (ang. project initiation), nadzorowania i sterowania przedsięwzięciem (ang. project monitoring and control), zarządzania jakością oprogramowania (ang. software quality management process). Przykładowo, do czynności wykonywanych w tym ostatnim procesie zalicza się kolejno: planowanie zapewnienia jakości oprogramowania, opracowanie miar jakości, zarządzanie kontrolą jakości i określenie potrzeb polepszenia jakości.

Procesy przedprodukcyjne podzielono na dwa rodzaje: opracowanie koncepcji (ang. concept exploration) i rozdział pracy (ang. system allocation process). Na grupę procesów bezpośrednio *produkcyjnych* składają się procesy: specyfikowania wymagań (and. requirements specification), projektowania (ang. design) i implementowania (ang. implementation). *Procesy poprodukcyjne* obejmują: instalowanie, eksploatację i serwis, wycofanie z użycia i konserwowanie.

Grupa *procesów integracyjnych* obejmuje cztery procesy: weryfikację i atestację, zarządzanie konfiguracją, wytwarzanie dokumentacji oraz szkolenie. Poniżej omówię przykładowe czynności niektórych procesów lub przykładowe procesy każdej z grup.

2. PRZYKŁAD CZYNNOŚCI PROCESU ZARZĄDZANIA PRZEDSIĘWZIĘCIEM

Jednym z trzech procesów służących zarządzaniu przedsięwzięciem (ang. project management), zidentyfikowanym w normie [35], jest proces kontroli jakości. Jest on wykonywany przez cały okres istnienia oprogramowania - nie tylko podczas jego wytwarzania. Jedna z pierwszych norm kontroli jakości oprogramowania, norma amerykańska [20], dotyczyła zasadniczej czynności - zapewnienia jakości. Obecnie, zmierzają ku końcowi prace nad aktualizacją tej normy. Jej zalecenia dotyczą przede wszystkim tzw. *oprogramowania krytycznego*, przez co rozumie się oprogramowanie, którego awarie mogą zagrozić bezpieczeństwu lub spowodować znaczne straty materialne lub społeczne (ang. software whose failures would impact safety or cause large financial or social losses).

2.1 Plan kontroli jakości oprogramowania

Zgodnie z tą normą, dokument zwany *planem zapewnienia jakości oprogramowania* (ang. software quality assurance plan) powinien obejmować piętnaście rozdziałów w następującej kolejności, według tytułów:

1) Cel (określający cel i zakres planu dla poszczególnych modułów oraz ich przeznaczenie)

2) Wykaz dokumentów (na które powołano się w planie)

3) Zarządzanie (rozumiane jako struktura organizacyjna, podział zadań i podział odpowiedzialności za ich realizację)

4) Dokumentacja (obejmująca co najmniej minimalny zbiór dokumentów niezbędnych do zapewnienia jakości, jak specyfikacja wymagań, opis projektu, plan weryfikacji i atestacji, sprawozdanie z weryfikacji i atestacji, dokumentacja użytkowa, plan

zarządzania konfiguracją)

5) Normy, praktyki, konwencje i metryki (stosowane do zapewniania jakości oprogramowania - dokładniej chodzi o standardy dokumentowania, określania struktury logicznej, kodowania, komentowania, wykonywania przeglądów i lustracji oraz testowania)

6) Przeglądy i lustracje (ang. reviews and audits), które powinny obejmować co najmniej następujące czynności: przegląd wymagań (ang. software requirements review, SRR), wstępny przegląd projektu (ang. preliminary design review, PDR), krytyczny przegląd projektu (ang. critical design review, CDR), przegląd planu weryfikacji i atestacji (ang. software verification and validation plan review, SVVPR), lustracja funkcjonalna (ang. functional audit), lustracja fizyczna (ang. physical audit), lustracja produkcyjna (ang. in-process audit), przeglądy zewnętrzne (ang. managerial reviews), przegląd planu zarządzania konfiguracją (ang. software configuration management plan review, SCMPR), przegląd końcowy (ang. post-mortem review)

7) Testy (dodatkowe, tzn. nie objęte planem weryfikacji i atestacji)

8) Zaistniałe problemy i działania korekcyjne (określające zasady postępowania w wypadku wykrycia defektów)

9) Narzędzia, techniki i metodyki (stosowane z myślą o zapewnieniu jakości oprogramowania)

10) Kontrola kodu (przed nieupoważnionym użyciem podczas całego okresu istnienia oprogramowania)

11) Kontrola nośników (przed nieupoważnionym dostępem)

12) Kontrola dostawców

13) Gromadzenie, utrzymywanie i przechowywanie dokumentów (służących zapewnieniu jakości oprogramowania)

14) Szkolenie (konieczne dla zepewnienia jakości oprogramowania)

15) Analiza ryzyka.

Ważną częścią tej normy jest zdefiniowanie pięciu metryk, opartych na normach [24,25], związanych z jakością oprogramowania:

- metryka rozgałęzień
- metryka punktów decyzyjnych
- metryka dziedzinowa
- metryka komunikatów błędów
- metryka wykonania wymagań.

Przykładowo, przez *metrykę rozgałęzień* (ang. branch metric) rozumie się stosunek liczby modułów, w których każde rozgałęzienie zostało wykonane przynajmniej raz, do całkowitej liczby modułów. W tym samym duchu są utrzymane pozostałe definicje, np. *metryka dziedzinowa* (ang. domain metric) jest to stosunek liczby modułów, w których poprawnie przetworzono przynajmniej jeden zestaw ważnych i jeden zestaw nieważnych danych wejściowych każdej klasy (np. komunikatów zewnętrznych, poleceń operatorskich, danych lokalnych itp.), do całkowitej liczby modułów.

2.2 Zalecenia dotyczące stosowania planu

Ważnym dokumentem uzupełniającym tę normę są zalecenia dotyczące jej użycia [26]. Każdy z rozdziałów planu zapewnienia jakości jest tu omówiony dokładniej wraz z dodatkowymi wyjaśnieniami. Przykładowo, dokumentacja punktu 4 może być rozszerzona o plan wytwarzania oprogramowania (ang. software development plan), podręcznik norm i zasad (ang. standards and procedures manual), plan zarządzania przedsięwzięciem (ang. software project management plan) i podręcznik konserwacji

oprogramowania (ang. software maintenance manual), a także dodatkowe dokumenty, jak wymagania użytkowe, specyfikacja sprzężeń wewnętrznych i zewnętrznych, instrukcja eksploatacji i instrukcja instalowania, podręcznik i plan szkolenia. Zalecana zawartość każdego z tych dokumentów jest oczywiście omówiona mniej lub bardziej szczegółowo, zależnie od jego wagi dla zapewnienia jakości. W punkcie 5 z kolei wyjaśniono rozróżnienie *konwencji*, jako standardowych wymagań dotyczących produktu programowego, od *praktyk*, rozumianych jako standardowe wymagania narzucone na proces wytwórczy oprogramowania. Praktyki i konwencje uznane za konieczne do stosowania objęte są normami.

Najobszerniejsze jednak, bo najważniejsze dla zapewnienia jakości, są zawarte w tym rozdziale wyjaśnienia dotyczące punktu 6 planu - przeglądy i lustracje. Każdy element oprogramowania i każdy dokument ujęty w procesie wytwórczym, służący zapewnieniu jakości oprogramowania, musi przejść przez wieloraką kontrolę. Szczególnie ważne są dwa rodzaje tej kontroli:

o *przeglądy*, które polegają na badaniu integralności projektu, wykrywaniu usterek i wprowadzaniu niezbędnych zmian, mają więc znaczenie techniczne

o *lustracje*, które stanowią podstawę do podejmowania decyzji, a więc mają znaczenie dla szczebli kierowniczych.

Dla wszystkich rodzajów przeglądów i lustracji opisano ich przebieg, podając mniej lub bardziej szczegółowe zasady postępowania, np. zalecając podczas przeglądu specyfikacji wymagań sprawdzenie jej m.in. ze względu na następujące cechy:

- śladowalność i zupełność
- stosowność wymagań, algorytmów i równań
- poprawność opisów logicznych
- zgodność sprzężeń zewnętrznych

- dostosowanie opisów do sprzęgu człowiek-maszyna
- niesprzeczność użytych symboli i specyfikacji sprzężeń
- dostępność stałych i tabel do wykonywania obliczeń
- testowalność wymagań
- stosowność i zupełność wymagań weryfikacji i odbioru.

Spośród wszystkich rodzajów przeglądów, jedynie przegląd zewnętrzny musi być powierzony niezależnej jednostce organizacyjnej lub oddzielnemu kontrahentowi. Trzy rodzaje lustracji różnią się natomiast celami i momentami ich przeprowadzenia. *Lustracja funkcjonalna* jest wykonywana przed dostawą oprogramowania i ma na celu sprawdzenie, czy spełnione zostały wszystkie wymagania zawarte w specyfikacji. *Lustracja fizyczna* jest przeprowadzana w celu sprawdzenia, że oprogramowanie i jego dokumentacja są wewnętrznie niesprzeczne i gotowe do dostawy. *Lustracja produkcyjna* polega na wykonaniu kontroli implementacji, tj. zgodności kodu z dokumentacją, projektem i wymaganiami, w celu oceny postępów procesu produkcyjnego.

Oprócz szczegółowych wyjaśnień dotyczących zawartości planu zapewnienia jakości oprogramowania, w trzech odrębnych rozdziałach zaleceń [26] opisano też zasady realizacji tego planu, jego oceny oraz modyfikowania.

3. STUDIUM REALIZOWALNOŚCI - PRZYKŁAD CZYNNOŚCI PROCESU PRZEDPRODUKCYJNEGO

W pierwszym z procesów przedprodukcyjnych, w normie [35] zidentyfikowano pięć czynności:

- określenie warunków i potrzeb
- sformułowanie potencjalnych rozwiązań
- przeprowadzenie studium realizowalności
- zaplanowanie przeniesienia (jeżeli ma zastosowanie)
- sprecyzowanie i sfinalizowanie warunków i potrzeb.

Choć dla żadnej z tych czynności nie opracowano szczegółowej normy, jak na przykład dla zapewnienia jakości, można przytoczyć interesującą propozycję jednej z ciekawszych i chyba nieodzownych czynności, jaką jest *studium realizowalności* (ang. feasibility study). Zaproponowano, aby punktem wyjścia do przeprowadzenia studium realizowalności były wyniki innych czynności tego procesu, a więc:

- wstępne określenie potrzeb
- zidentyfikowanie możliwych rozwiązań
- przewidywane ograniczenia i korzyści.

W prowadzeniu studium zaleca się, co jest naturalne, intensywne korzystanie z technik modelowania i makietowania (ang. prototyping). Produktem tej czynności powinny być zalecenia dotyczące wykonania planowanego przedsięwzięcia, np. określające, czy jest możliwe jego wykonanie, a jeśli tak, to czy oprogramowanie należy wykonać czy zakupić itd. Z nielicznej literatury dotyczącej tej czynności zwraca uwagę raport [46], ponieważ precyzuje pewne wymagania normalizacyjne narzucone na przeprowadzanie studium realizowalności. Tematowi temu warto poświęcić trochę uwagi także z tego powodu, że jak sądzę, nie stykamy się z tego rodzaju problemami w krajowej praktyce

programistycznej.

Studium realizowalności może być wykonywane zarówno wewnątrz organizacji planującej zastosowanie lub wytworzenie oprogramowania, jak i na zewnątrz niej, tj. przez oddzielnego kontrahenta. Jednakże, zlecenie przeprowadzenia takiego studium wykonawcy zewnętrznemu jest bardziej celowe o tyle, że wtedy łatwiej ustalić, czego oczekuje się jako wyniku studium. Punktem wyjścia powinno więc być jednoznaczne stwierdzenie, na jakie pytanie lub pytania zleceniodawca oczekuje odpowiedzi, tj. sformułowanie problemu.

W zasadzie studium realizowalności należy wykonać przed rozpoczęciem lub na samym początku przedsięwzięcia programistycznego. Skutkiem przeprowadzenia takiego studium powinno być jednoznaczne stwierdzenie lub wskazanie, czy należy podjąć się realizacji danego przedsięwzięcia, a w wypadku pozytywnej odpowiedzi - także wskazówki dotyczące jego ewentualnego przebiegu. Przy podejmowaniu rozstrzygającej decyzji należy uwzględnić główne kryteria, takie jak maksymalne nakłady, horyzont czasowy, liczba wykonawców, kwalifikacje personelu, możliwości technologiczne, warunki środowiskowe itp.

Ponieważ studium polega w zasadzie na wykonaniu trzech zespołów operacji, tj. zebraniu informacji, przeanalizowaniu jej i opracowaniu wniosków, końcowy dokument, zwany *sprawozdaniem z badania realizowalności* (ang. feasibility study report, FSR), powinien zawierać co najmniej:

- wyraźne sformułowanie problemu (pytania, na które przeprowadzane studium ma odpowiedzieć)
- określenie ograniczeń nałożonych na studium, takich jak termin realizacji, dostęp do informacji itp., a więc także na wiarygodność wyników studium

- zalecenia dotyczące rodzaju informacji, na których oparto studium
- różne opcje rozwiązania problemu
- sugestie dotyczące sposobu i czasu implementacji poszczególnych opcji
- oszacowanie kosztów poszczególnych opcji rozwiązania, zarówno w procesie produkcyjnym jak i związanych z późniejszą konserwacją
- wybór najkorzystniejszej opcji rozwiązania według jasno sformułowanego kryterium.

Warto podkreślić, że oprócz fachowości wykonawców istotny wpływ na wiarygodność wyników studium ma czas jego realizacji (tzn. czy nie jest za krótki) oraz dostęp do informacji (tzn. czy jest pełny). Często zdarza się bowiem, że przyszły użytkownik, wytwórca czy zleceniodawca, w chwili wykonywania studium realizowalności nie jest jeszcze w pełni świadomy całego zakresu przedsięwzięcia programistycznego i wszystkich cech oprogramowania - dlatego nie jest w stanie udzielić wystarczających informacji; może to podważyć wiarygodność wyników studium.

Na tej podstawie, autorzy omawianego dokumentu [46] zawarli w nim zalecenia dotyczące treści, złożonego z dziewięciu rozdziałów, sprawozdania z badania realizowalności. Wydaje się, że ich zawartość może w chwili obecnej stanowić punkt wyjścia do opracowania analogicznych lecz chyba ważniejszych zaleceń dla przygotowywania warunków umowy (ang. terms of reference) dotyczącej studium, opartych na trzech podstawowych pytaniach:

o Co należy zrobić wykonując studium realizowalności ?

o Jak to zrobić ?

o Jak zweryfikować otrzymane wyniki ?

4. PRZYKŁADY PROCESÓW PRODUKCYJNYCH

Każdy z trzech procesów produkcyjnych, tj. dotyczących ścisłego wytwarzania oprogramowania, jak formułowanie wymagań, projektowanie i implementowanie, ma kluczowe znaczenie dla jakości ostatecznego produktu. Wydaje się, że zasady implementacji, obejmujące takie czynności jak wytworzenie kodu źródłowego (kodowanie), opracowanie danych testowych, wygenerowanie kodu wynikowego, tworzenie dokumentacji eksploatacyjnej, planowanie i wykonywanie integrowania modułów, są najlepiej poznane i najwszechstronniej stosowane. Poniżej, zwrócę więc uwagę przede wszystkim na początkowe procesy produkcyjne: specyfikowanie wymagań i projektowanie.

4.1 Proces specyfikowania wymagań

W jednej z pierwszych norm [23] dotyczących wytwarzania oprogramowania, obejmującej specyfikowanie wymagań programowych, określono przede wszystkim konieczne cechy takiej specyfikacji:

- jednoznaczność (ang. unambiguity)
- zupełność (ang. completeness)
- sprawdzalność (ang. verifiability)
- niesprzeczność (ang. consistency)
- modyfikowalność (ang. modifiability)
- śladowalność (ang. traceability)
- użyteczność w fazach eksploatacji i konserwacji (ang. usability).

4.1.1 Specyfikowanie wymagań programowych według IEEE

Każda z tych cech jest opisana bardziej szczegółowo. *Jednoznaczność* specyfikacji najłatwiej osiągnąć stosując do specyfikowania język formalny. Tej dziedzinie poświęca się

współcześnie niezmiernie wiele uwagi, jednak większość metod formalnych udaje się stosować jedynie do prostych przypadków (por. [15], [37], [48]), a stopień ich komplikacji na ogół wyklucza użycie w rzeczywistych przedsięwzięciach programistycznych. Do *zupełności* specyfikacji można się zbliżyć eliminując z niej maksymalnie tzw. TBD's (ang. to be determined), tj. fragmenty do późniejszego określenia. *Weryfikowalność* specyfikacji polega na tym, że muszą istnieć sposoby sprawdzenia, czy gotowy produkt spełnia poszczególne wymagania zawarte w specyfikacji. Przykładowo, za nieweryfikowalne należy uznać następujące stwierdzenia:

- produkt powinien mieć dobre sprzężenie użytkowe (nie wiadomo, co znaczy - dobre)
- program nie może wejść w nieskończoną pętlę (niemożliwe może być udowodnienie, że program nie zawiera pętli nieskończonych).

Niesprzeczność specyfikacji jest osiągnięta wtedy, gdy żadne z indywidualnych wymagań nie wchodzą ze sobą w kolizję, co może się zdarzyć, jeśli np.:

- w dwóch różnych miejscach specyfikacji żąda się uzyskania dwóch przeciwstawnych cech tego samego obiektu
- narzuca się niezgodne wymagania na tę samą czynność (np. do wykonania w tym samym czasie)
- dwa lub więcej wymagań opisuje ten sam obiekt, czyniąc to różnymi językami.

Modyfikowalność specyfikacji jest znacznie ułatwiona, jeśli dokument specyfikacyjny ma dobrą organizację, tj. klarowny podział na rozdziały, spis treści, indeks, słownik itp., i nie zawiera nadmiarowości (redundancji). Choć sama nadmiarowość nie jest błędem, łatwo prowadzi do błędów. *Śladowalność* polega na tym, że pochodzenie i przyczyna (a więc ślad, stąd -

śladowalność) każdego wymagania są na tyle jasno określone, że można zanalizować (prześledzić) sposób tworzenia danego wymagania i odtworzyć jego źródło. Dotyczy to zarówno śladów wstecz jak i wprzód, ponieważ specyfikacja jest dokumentem wielokrotnie zmienianym przed skrystalizowaniem się ostatecznej wersji.

Podkreśla się, że dokument ten powinien być opracowany łącznie przez nabywcę (ang. customer) i dostawcę (ang. supplier) oprogramowania, a zawarte w nim wymagania powinny określać skutki działania oprogramowania, a nie sposoby ich osiągnięcia. Wyliczono też główne sposoby specyfikowania wymagań oraz - kilka rodzajów wymagań narzuconych na oprogramowanie, z których najważniejszymi są wymagania:

- funkcjonalne, określające, co oprogramowanie ma robić
- wydajnościowe, np. szybkość, czas reakcji, liczba i rozmiar obsługiwanych plików, liczba jednoczesnych użytkowników itp.
- sprzężeniowe, dotyczące zasad współpracy z użytkownikami, sprzętem, innym oprogramowaniem, systemem komunikacyjnym itd.

Opierając się na tych zasadach w normie [23] podano przykładowy kształt specyfikacji wymagań programowych oraz ich zawartości.

4.1.2 Specyfikowanie wymagań według EWICS

W bardzo blisko związanym z tą normą rozdziale dotyczącym specyfikacji, zawartym w zaleceniach [44], proces specyfikowania nieco rozszerzono. Podobnie, przez specyfikację rozumie się określenie wymagań, jakie powinno spełniać oprogramowanie zdaniem nabywcy, i podobnie zaleca się, aby specyfikacja zawierała jedynie wymagania, a nie proponowała rozwiązań, gdyż mogą one wprowadzać istotne ograniczenia.

Podkreśla się jednak, co jest niesłychanie ważne, że specyfikacja wymagań powinna być jedynym środkiem porozumienia między nabywcą a dostawcą (lub wykonawcą) systemu. Choć uważa się, że specyfikacja rzadko jest dokumentem ustalonym raz na zawsze (jest to oczywiste, ponieważ będąc przedmiotem zlecenia, stanowi przedmiot ciągłych negocjacji między zleceniodawcą a zleceniobiorcą), powinna w pewnym momencie stać się dokumentem zamrożonym. Od tej chwili wszelkie zmiany specyfikacji winny być przedmiotem formalnej procedury zmian (ang. change control procedure).

Bardziej istotne jest jednak rozszerzenie rozumienia specyfikacji polegające na stwierdzeniu, że pełna specyfikacja powinna odnosić się nie tylko do tzw. systemu docelowego (ang. target system), będącego przedmiotem zamówienia czy dostawy, lecz też do warunków produkcji i warunków eksploatacji tego systemu. Z tego powodu, specyfikacja wymagań, oprócz wymagań dotyczących samego oprogramowania czy systemu docelowego, musi zawierać wymagania narzucane na środowisko eksploatacyjne, a także na środowisko produkcyjne, tzn. całe przedsięwzięcie (ang. project). Zasadnicza część zaleceń odpowiada więc wymienionemu założeniu, określając jak formułować wymagania dla systemu docelowego, jego środowiska, oraz - wymagania odnoszące się do całego cyklu produkcyjnego (przedsięwzięcia) i środowiska, w jakim jest realizowany.

Spośród zaleceń dotyczących systemu docelowego, najciekawsze są związane z wymaganiami decydującymi o rzetelności systemu (ponieważ tego dotyczy całość materiału [44]), a więc uwzględniającymi takie cechy, jak niezawodność, bezpieczeństwo i poufność, lecz także - adaptowalność, dyspozycyjność i konserwowalność systemu. Odnośnie pozostałych

trzech aspektów tego modelu specyfikacji, zalecenia dotyczą tylko cech specyficznych, np. dla środowiska eksploatacyjnego szczególnie istotne jest bezpieczeństwo, a dla całego przedsięwzięcia (tzn. cyklu produkcyjnego) i jego środowiska - zagadnienie kontroli jakości. Oczywiście, główną część specyfikacji powinny stanowić tradycyjnie wymagania dotyczące funkcji, wydajności i sprzężeń systemu docelowego - co też wyraźnie podkreślono w zaleceniach.

4.1.3 Normy a techniki i metody

Technologia tworzenia specyfikacji wymagań programowych (i systemowych) i automatyzacja tego procesu jest w stanie intensywnego rozwoju, dlatego też może się wydawać, że normy te są ubogie, nie zawierają istotnych wskazówek i można je znacznie rozwinąć, np. wzbogacając o metody, języki (techniki) i narzędzia specyfikacyjne [38]. Należy jednak pamiętać, że wtedy nie byłyby już normami lecz praktyką zalecaną przez zwolenników tych a nie innych metod, technik lub narzędzi. Pokusa znormalizowania jednej, wybranej metody lub techniki jest tym większa, że wytwórcy oprogramowania intensywnie dążą do posiadania narzędzi do automatycznego przekształcania specyfikacji formalnych na kod. Choć niejedną z tych technik stosuje się w produkcji oprogramowania już od wielu lat, żadne z dotychczas proponowanych podejść nie jest jeszcze na tyle dojrzałe, aby można je uznać za warte zalecania w standardowej postaci. Stosunkowo najbliższa osiągnięcia tego stanu jest metoda VMD (ang. Vienna Development Method), której prace standaryzacyjne już rozpoczęto, lecz jej właśnie brakuje przede wszystkim narzędzi - a one głównie świadczą o dojrzałości metody.

4.2 Proces projektowania

Proces projektowania uważa się za decydujący dla powodzenia całego przedsięwzięcia (tj. wytworzenia produktu o wysokiej jakości), ponieważ polega on na przekształceniu specyfikacji wymagań, określającej *co należy zrobić*, na specyfikację projektową, określającą *jak należy to zrobić*. Na ogół istnieje zgodność co do podziału procesu projektowego na projektowanie ogólne, tj. projektowanie architektury oprogramowania, a więc funkcji i struktury jego składowych, oraz - projektowanie szczegółowe, tj. struktur danych i algorytmów tworzących poszczególne składowe.

4.2.1 Opis projektu według IEEE

Do tych dwóch podstawowych czynności procesu projektowania w normie [35] dodano jeszcze następujące:

- analiza przepływu informacji
- projektowanie sprzężeń
- projektowanie bazy danych
- wybór lub opracowanie algorytmów.

Wynikiem projektowania szczegółowego, opartego m.in. na specyfikacji wymagań, opisie architektury, opisie bazy danych, opisie sprzężeń i opisach algorytmów, powinien być niezwykle ważny dokument - *opis projektu oprogramowania* (ang. software design description). Według definicji [29] opis projektu oprogramowania jest to reprezentacja oprogramowania utworzona w celu ułatwienia analizy, planowania, implementowania i podejmowania decyzji. Opis projektu służy jako środek przekazywania informacji o projekcie oprogramowania.

Zasady i sposoby opisu projektów są objęte samodzielną normą [29], którą naprawdę warto przestudiować dokładniej. Przede wszystkim należy ponownie podkreślić, że ta norma także nie wyróżnia żadnej szczególnej metodyki ani techniki projektowania, pozostawiając ich wybór do decyzji projektantom. Podaje natomiast zalecenia dotyczące zawartości i organizacji dokumentu zwanego opisem projektu oprogramowania. Elementy składowe projektu nazywa się w niej *jednostkami projektowymi* (ang. design entities). Przykładami jednostek projektowych mogą być systemy, podsystemy, zbiory danych, moduły, programy, procesy, podprogramy i procedury. Jednostki projektowe opisuje się za pomocą atrybutów, tj. określonych charakterystyk lub właściwości. W normie zaproponowano opis jednostek projektowych za pomocą dziesięciu atrybutów spełniających następujące kryteria:

- uwzględnienie atrybutu jest konieczne we wszystkich przedsięwzięciach programistycznych
- niepoprawna specyfikacja wartości atrybutu mogłaby spowodować defekt opracowywanego oprogramowania
- atrybut opisuje informację ściśle projektową, a nie informację tylko związaną z projektem.

Tak wybrana grupa dziesięciu atrybutów obejmuje następujące:

1) Identyfikator, tj. jednoznaczna nazwa charakteryzująca jednostkę
2) Typ, tj. określenie rodzaju jednostki
3) Cel (tzn. odpowiedź na pytanie o sens istnienia jednostki)
4) Funkcja, tj. stwierdzenie dotyczące czynności wykonywanych przez jednostkę, a dokładniej transformacji danych wejściowych, aby uzyskać pożądane dane wyjściowe

5) Jednostki składowe, tworzące strukturę rozważanej jednostki

6) Zależności, tj. hierarchiczne związki z innymi jednostkami

7) Zasoby, tzn. zewnętrzne wobec projektu źródła zasilające, niezbędne usługi itp. (np. drukarki, pamięci, biblioteki, procesory itd.)

8) Sprzężenia, tj. opis zasad współpracy z innymi jednostkami

9) Przetwarzanie (tzn. opis metody prowadzącej do wykonania funkcji)

10) Dane (wewnętrzne jednostki).

Zalecono szczegółowy opis atrybutów, który np. dla sprzężenia powinien zawierać m.in. mechanizmy wywoływania jednostki, wykonywania przerwań, przekazywania parametrów i komunikatów, korzystania ze wspólnych danych, bezpośredniego dostępu do danych, składnię komunikatów, formaty danych, zakresy wartości wejściowych i ich znaczenie, kody błędów, format obrazu, język komunikacji itd.

Powyższa informacja w postaci atrybutów może być powiązana w opisie projektu różnymi sposobami. Proponowane zasady polegają na wyborze niektórych wymienionych cech i połączeniu ich w opis wyższego poziomu np.:

- opis dekompozycyjny, uwzględniający identyfikator, typ, cel, funkcję i składowe każdego modułu
- opis współoddziaływań, określający dla wszystkich modułów ich identyfikatory, typy, cele, zależności i zasoby
- opis zewnętrzny, obejmujący identyfikator, typ i sprzężenia każdego modułu
- opis wewnętrzny, na który składają się identyfikatory, przetwarzanie i dane wszystkich modułów.

4.2.2 Zalecenia projektowe EWICS

Nieco inaczej do procesu projektowania podchodzi się w odpowiednim rozdziale zaleceń [45]. Nie należy zresztą temu się dziwić, ponieważ dotyczą one przede wszystkim projektowania systemów z uwzględnieniem bezpieczeństwa. Podstawowa różnica polega tu na zwróceniu większej uwagi nie na sam proces projektowania lecz na czynności przygotowawcze. Sposób postępowania określono w pięciu zasadach projektowych:

1) niezależna analiza bezpieczeństwa, polegająca na sporządzeniu i zbadaniu listy możliwych wypadków, zagrożeń itp., i powiązaniu ich możliwych przyczyn z oprogramowaniem

2) rozdzielenie projektu oprogramowania na część krytyczną dla bezpieczeństwa i niekrytyczną

3) zapewnienie maksymalnie wysokiej niezawodności części krytycznej

4) zapewnienie bezpieczeństwa kosztem niezawodności lub innych atrybutów jakości

5) ciągłe nadzorowanie części krytycznej oprogramowania.

Realizację tych zasad proponuje się w czterech krokach, z których pierwsze dwa nie mają charakteru ściśle projektowego lecz przygotowawczy:

- wszechstronna analiza bezpieczeństwa komputeryzowanej instalacji i jej otoczenia (wynikiem tego kroku jest wyróżnienie celów bezpieczeństwa, rozumianych jako podział wszystkich możliwych stanów na bezpieczne i niebezpieczne, oraz sformułowanie ograniczeń projektowych)
- analiza specyfikacji projektowej pod względem bezpieczeństwa (wynikiem tego kroku jest lista wymagań projektowych, zwiększających bezpieczeństwo).

Krok trzeci, ściśle projektowy, polega na opracowaniu dokumentu

będącego projektem systemu docelowego, na podstawie: specyfikacji wymagań funkcjonalnych, sprzężeniowych i in., oraz ograniczeń i wskazówek sformułowanych w dwóch poprzednich krokach. W tym kroku można oczywiście wprost stosować sposób przygotowywania projektu zalecany w normie [29]. Ostatni krok polega na zweryfikowaniu dokumentów projektowych pod względem zgodności ze specyfikacją oraz z wymaganiami bezpieczeństwa określonymi w dwóch krokach przygotowawczych. Po wprowadzeniu poprawek i ponownym zweryfikowaniu, projektowanie uważa się za zakończone.

Aby ułatwić realizację projektów, opracowano szczegółowy poradnik zalecanych technik projektowania [2], grupując je w pięciu rozdziałach poświęconych kolejno: analizie bezpieczeństwa, unikaniu defektów, wykrywaniu defektów, wykrywaniu uszkodzeń i neutralizowaniu uszkodzeń (ang. failure containment). Każdą technikę scharakteryzowano dość szczegółowo, podając najbardziej istotne informacje w następującej kolejności: cel, opis, warunki użycia, główne wady i podstawowe zalety, powiązanie z innymi metodami, dostępne narzędzia, źródła literaturowe.

W rozdziale dotyczącym analizy bezpieczeństwa umieszczono te metody, które zaleca się stosować w krokach przygotowawczych projektowania, do zidentyfikowania zagrożeń i ich źródeł. Do takich metod należą m.in.: diagramy przyczyn i skutków (ang. cause-consequence diagrams), drzewa zdarzeń i drzewa defektów (ang. event-tree analysis, fault-tree analysis), karty gry w Go (ang. Go charts) i in. Spośród metod unikania defektów wymieniono przede wszystkim metody formalne, jak rachunek CCS i CSP (ang. calculus of communicating systems, communicating sequential processes), sieci Petriego, logika temporalna, metoda

VDM lub Z, oraz - metody systematyczne (metoda Jacksona, MASCOT - ang. modular approach to software construction, operation and test) i programowanie strukturalne. Jako jedną z metod do wykrywania defektów omówiono makietowanie (ang. prototyping), a do wykrywania uszkodzeń - asercje. Spośród głównych metod neutralizowania uszkodzeń opisano m.in. programowanie defensywne, tolerowanie defektów, wprowadzanie zróżnicowania (ang. diversity) i nadmiarowości (ang. redundancy).

4.3 Kodowanie - przykład czynności procesu implementowania

Jedną z najważniejszych czynności implementacyjnych, oprócz kilku innych, wymienionych na wstępie tego artykułu, jest kodowanie. Wbrew pozorom jednak, choć jest to bez wątpienia najstarsza i najlepiej poznana czynność wykonywana przez programistów, wcale niełatwo znaleźć sensowne standardy czy zalecenia dotyczące jej przebiegu. Jednym z ciekawszych, choć bardzo ograniczonym, bo mającym zastosowanie tylko do Fortranu 77, jest jeden z rozdziałów raportu [8]. Celem tych zaleceń jest, jak podają autorzy, uzyskanie maksymalnej przenośności, niezawodności i zrozumiałości wytworzonego kodu, a także - jego użyteczności i testowalności. Jest oczywiste, że nie wystarczy do tego norma samego języka, gdyż nieumiejętne stosowanie niektórych konstrukcji może być bardzo niebezpieczne. Konieczne są więc zalecenia dotyczące stylu programowania i taką rolę odgrywa ten fragment wymienionego raportu.

Zalecenia ogólne, jak np. to, że każdy moduł wymaga dokładnego opisu w postaci szczegółowego komentarza w nagłówku, lub że wszystkie zmienne powinny być zadeklarowane i zdefiniowane w programie głównym i w modułach, gdzie są używane itp., nie są zbyt konstruktywne. O wiele bardziej interesujące

są szczegółowe wskazówki dotyczące kolejno:

- budowy modułów, włącznie z określeniem ich maksymalnego rozmiaru, organizacji programu głównego, funkcji, podprogramów i bloków, komentarzy, zasad stosowania odstępów i wcięć, wierszy kontynuacji
- użycia struktur sterujących (wśród nich instrukcji GOTO), włącznie z numerowaniem wierszy, instrukcją CONTINUE, powrotami z modułów i obsługą błędów
- tworzenia zmiennych i struktur danych, włącznie z ich typizacją, inicjowaniem, oraz używaniem zmiennych wspólnych i lokalnych
- wykorzystania wejścia-wyjścia i kontroli błędów.

Warto też dodać, że oprócz kodu wynikiem tej czynności powinien być raport nt. użytych narzędzi i technik (podlegający zresztą zatwierdzeniu przez zleceniodawcę).

5. PROCESY POPRODUKCYJNE

Cztery procesy poprodukcyjne, wyróżnione w normie [35] to: instalowanie oprogramowania, eksploatowanie i serwis, wycofanie oprogramowania z użycia, oraz konserwacja. Każdy z tych procesów dzieli się na szereg czynności, które dla porządku wyliczono poniżej:

o *instalowanie oprogramowania*, na co składa się
- planowanie instalacji
- dystrybucja oprogramowania
- instalacja
- zakładanie bazy danych
- kontrola oprogramowania w środowisku eksploatacyjnym
- prowadzenie aktualizacji

o *eksploatowanie i serwis*, które obejmuje
- eksploatację systemu
- zapewnienie pomocy technicznej i konsultacji
- utrzymanie kroniki wezwań serwisu

o *wycofanie z użycia*, które dzieli się na
- powiadomienie użytkowników
- prowadzenie równoległej eksploatacji (dwóch systemów)
- zakończenie eksploatacji

o *konserwacja*, polegająca na ponownym zastosowaniu całego cyklu produkcyjnego począwszy od fazy formułowania koncepcji.

Jak widać, procesy poprodukcyjne nie są specjalnie ciekawe dla wykonawców oprogramowania (ang. developers), za wyjątkiem procesu konserwacji, który z chwilą podjęcia formalnej decyzji o wprowadzeniu zmian przekształca się w nowy cykl produkcyjny. Nie jest to jednakże typowy punkt widzenia na konserwowanie oprogramowania. Inne podejścia zaprezentowano, na przykład, w [40,45].

5.1 Konserwacja oprogramowania według NBS

W bardzo ciekawym raporcie Krajowego Urzędu Normalizacyjnego USA (National Bureau of Standards - NBS) [40] omówiono techniczne i organizacyjne aspekty konserwacji oprogramowania oraz przedstawiono propozycje i zalecenia prowadzące do udoskonalenia tego procesu, a co za tym idzie - do polepszenia jakości oprogramowania. Przez *konserwację oprogramowania* rozumie się wszystkie te czynności wymagane do utrzymania oprogramowania w eksploatacji, które są wykonywane po odbiorze oprogramowania i wprowadzeniu go do eksploatacji. Konserwacja oprogramowania wiąże się więc z dokonywaniem zmian w pierwotnie przyjętym oprogramowaniu. Zmiany te mogą polegać na poprawianiu, wstawianiu, usuwaniu, rozszerzaniu i wzbogacaniu oprogramowania.

Z przyczyn technicznych wyróżniono 3 rodzaje konserwacji (podział ten jest stosowany już od kilkunastu lat):

- *konserwacja korekcyjna*, polegająca na wprowadzaniu zmian wynikających z zauważonych defektów oprogramowania, powodujących błędy w działaniu
- *konserwacja adaptacyjna*, związana ze zmianami w środowisku, w którym oprogramowanie jest eksploatowane
- *konserwacja perfekcyjna*, dotycząca wszystkich zmian spowodowanych zwiększonymi wymaganiami użytkowników.

Dodatkowo, wspomniano też o tzw.:

- *konserwacji prewencyjnej*, określając ją jako zapobiegawcze udoskonalanie oprogramowania w celu zoptymalizowania go oraz wyeliminowania zauważonych braków i wad.

Przyczyny wymagające stosowania konserwacji korekcyjnej podzielono na defekty projektowe, logiczne i defekty kodowania. Jako przykładowe źródła zmian wymagających konserwacji adaptacyjnej wyliczono: regulacje prawne, konfigurację sprzętu, formaty danych i struktury plików, oprogramowanie podstawowe (system operacyjny, kompilatory, programy usługowe). Do przyczyn powodujących konieczność prowadzenia konserwacji perfekcyjnej zaliczono głównie zmiany w specyfikacji, np. wzrost wymagań wydajnościowych itp.

W celu głębszego zrozumienia procesu konserwacji wyróżniono kilkanaście kolejnych kroków, upraszczających jego prowadzenie:

- określenie potrzeby zmian
- dostarczenie żądania zmian (tj. formalnego dokumentu)
- analiza wymagań (zawartych w powyższym dokumencie)
- przyjęcie lub odrzucenie żądania zmian
- opracowanie harmonogramu prac
- analiza projektu
- przegląd projektu
- zmiana i uruchamianie kodu
- przegląd proponowanych zmian kodu
- testowanie
- uaktualnienie dokumentacji
- lustracja
- odbiór przez użytkownika
- poinstalacyjny przegląd zmian i ich wpływu na cały system
- zakończenie prac.

Nie ulega wątpliwości, że przebieg tego procesu nie musi być liniowy, tzn. w podanej kolejności; dopuszczalne są iteracje i powtarzanie określonych czynności.

Podane dalej zasady zalecane do stosowania w procesie konserwacji podzielono na 4 grupy:

1) ogólne reguły kontroli procesów konserwacji korekcyjnej, adaptacyjnej i perfekcyjnej

zalecenia techniczne, obejmujące:

zasady tworzenia kodu źródłowego, jak użycie jednego języka wysokiego poziomu, konwencje kodowania, strukturalizacja i modularyzacja programu, standardowe definiowanie danych, obszerne komentowanie kodu, unikanie rozszerzeń kompilatora,

- zasady dokumentowania programów
- zalecane techniki kodowania i dokonywania przeglądów, jak kodowanie zstępujące lub wstępujące (ang. top-down, bottom-up), przeglądy wzajemne (ang. peer reviews), inspekcje, zespół głównego programisty
- zasady kontrolowania zmian obejmujące żądanie zmian, lustrację kodu (ang. code audit, rozumianą jako procedurę służącą do określenia, w jakim stopniu kod spełnia normy i praktyki kodowania oraz specyfikację projektową) oraz przegląd i przyjęcie zmian
- zasady testowania

3) narzędzia szczególnie użyteczne w konserwacji oprogramowania, np. analizator odwołań (ang. cross referencer), komparator, programy diagnostyczne, biblioteki usługowe i dokumentacyjne, interakcyjne programy uruchomieniowe, generatory danych testowych i wiele innych programów

4) zasady zarządzania, obejmujące określenie celu konserwacji, strategii osiągnięcia tego celu i obsady kluczowych stanowisk prowadzących do jego realizacji.

5.2 Konserwacja oprogramowania według EWICS

Jeden z rozdziałów najnowszych zaleceń komitetu ds. niezawodności i bezpieczeństwa EWICS (ang. European Workshop on Industrial Computer Systems) dotyczy konserwowania i modyfikowania systemów komputerowych z uwzględnieniem

bezpieczeństwa [45,47]. Muszą one więc stanowić wzmocnienie wszelkich dotychczas znanych norm i praktyk dotyczących konserwowania oprogramowania, gdyż będzie ono przeznaczone do zastosowań szczególnie krytycznych. Tak też jest w rzeczywistości, co widać już po kształcie dokumentu, który przedstawia zalecenia w sposób możliwie sformalizowany. Duży stopień precyzji zaleceń wydaje się szczególnie istotny w konserwacji przede wszystkim z tego względu, że systemy stosowane w zagadnieniach krytycznych na ogół nie mogą być w pełni przetestowane przed wprowadzeniem do eksploatacji.

Jest charakterystyczne, że w wymienionym dokumencie przyjęto także znany już, objęty nawet normami terminologicznymi, podział procesu konserwacji na konserwację prewencyjną, korekcyjną, perfekcyjną i adaptacyjną. Wyodrębniono przy tym cztery elementy składowe procesu konserwacji, od których zależy zapewnienie jego powodzenia. Głównym wyróżnionym elementem jest uwzględnienie aspektów bezpieczeństwa, dokonywane trójpoziomowo przez szczeble kierownicze użytkownika, wewnętrzną komórkę ds. bezpieczeństwa systemów i podobny organ zewnętrzny. Przykładowo, użytkownicy mogą przeprowadzić konserwację prewencyjną i korekcyjną samodzielnie, ale w przypadku konserwacji perfekcyjnej i adaptacyjnej nie mogą tego zrobić bez odwołania się do komórki wewnętrznej lub zewnętrznego organu ds. bezpieczeństwa systemów. Drugim elementem mającym wpływ na powodzenie konserwacji jest dobór technik zarządzania, np. dotyczących ustalania celów i priorytetów, stosowania norm i zaleceń, dokumentowania procesu konserwacji, przeprowadzania lustracji i przeglądów. Pozostałe elementy to dobór narzędzi oraz doświadczenie personelu technicznego.

Techniczne podejście do procesu konserwacji oparto na formalizmie opisującym cały proces jako cyklicznie wykonywany ciąg czynności, rozpoczynający się od żądania zmian a kończący wydaniem sprawozdania z konserwacji, co przedstawia się w postaci diagramu przepływu danych (ang. data flow diagram). Na najwyższym poziomie abstrakcji (rys. 1) wyróżniono tylko dwie główne czynności, wykonywane w każdym cyklu konserwacyjnym:

- *właściwa konserwacja*
- *ponowna atestacja i zarządzanie konfiguracją*, polegające na ponownym przetestowaniu systemu i zebraniu danych o wykonanych czynnościach oraz zmianach dokonanych w konfiguracji systemu.

Trzecim, oprócz tych czynności, niezwykle ważnym ogniwem procesu konserwacji jest *baza danych*, zawierająca wszystkie dane niezbędne do wykonania konserwacji, szczegóły konfiguracji (pierwotnej i zmienionej) oraz zapisy historii systemu i procesu.

Tak rozumiany proces konserwacji opisano w zaleceniach oddzielnie dla każdego rodzaju konserwacji: prewencyjnej, korekcyjnej oraz - łącznie - perfekcyjnej i adaptacyjnej, rozróżniając dodatkowo trzy poziomy szczegółowości. Opis na pierwszym poziomie, przedstawiony na rys. 2-4, zawiera tylko część istotnych informacji, mających ogólny charakter. Przykładowo, inicjowanie konserwacji prewencyjnej następuje na podstawie dyspozycji pochodzącej z bazy danych, zgodnie z uprzednio ustalonym harmonogramem. Zainicjowanie konserwacji korekcyjnej może nastąpić jako konsekwencja przedstawionego przez operatora raportu o zauważonych anomaliach w działaniu obiektu. Konserwacja perfekcyjna i adaptacyjna może być zainic-

jowana przez szczebie kierownicze w postaci żądania badań możliwości modyfikacji. Każdy cykl konserwacji kończy się czynnościami atestacji i zarządzania konfiguracją (rys. 5), mającymi na celu stwierdzenie, że system po konserwacji spełnia wymagania funkcjonalne. Istotą tych czynności jest przetestowanie systemu i uaktualnienie dokumentacji; ich opis podano również na kilku poziomach szczegółowości.

W zaleceniach podkreśla się, że każda proponowana zmiana w eksploatowanym systemie musi być najpierw poddana dokładnej analizie, a następnie zostać zaakceptowana na odpowiednim szczeblu zarządzania. Jeżeli zmiana ma być zrealizowana, to należy ją traktować jako niezależne przedsięwzięcie i stosować sprawdzone metody projektowania i wytwarzania. Na zakończenie należy dokonać ponownej atestacji systemu i zaktualizować jego dokumentację. Podany w zaleceniach precyzyjny słownik danych oraz minispecyfikacje poszczególnych czynności konserwacyjnych czynią te zalecenia bardzo użytecznymi w praktyce.

6. PRZYKŁADY PROCESÓW INTEGRACYJNYCH

Procesy integracyjne są niezbędne do skutecznego zakończenia przedsięwzięcia programistycznego. Dwa z tych procesów odgrywają szczególną rolę - *weryfikacja i atestacja* oraz *zarządzanie konfiguracją* oprogramowania, dlatego poświęcono im, w każdym przypadku, po dwa odzielne dokumenty normalizacyjne, określające zarówno same zalecenia [21,28] jak i sposób korzystania z tych zaleceń [32,33]. Inny równie ważny proces, jakim jest *wytwarzanie dokumentacji*, ma nieco inne oblicza w różnych fazach okresu istnienia oprogramowania, dlatego poświęcono mu kilka oddzielnych norm.

6.1 Proces weryfikacji i atestacji według IEEE

Proces weryfikacji i atestacji obejmuje zaplanowanie i wykonanie wszystkich przeglądów, lustracji, testów i ocen, przeprowadzanych w okresie istnienia oprogramowania. Przez *weryfikację* (ang. verification) rozumie się tu proces stwierdzania, czy wynik określonej fazy cyklu produkcyjnego oprogramowania spełnia wymagania ustalone w poprzedniej fazie. *Atestacja* (ang. validation) oznacza natomiast proces oceny oprogramowania na końcu cyklu produkcyjnego, w celu stwierdzenia czy produkt jest zgodny z wymaganiami zawartymi w specyfikacji.

Norma [28], przyjmując punkt widzenia przedstawiony we wcześniejszych dokumentach normalizacyjnych [42], reprezentuje podejście sekwencyjne do procesu weryfikacji i atestacji, zalecając dla cyklu produkcyjnego oprogramowania stosowanie modelu kaskadowego, opisanego w [19], z podziałem na fazy: formułowania koncepcji, specyfikowania wymagań, projektowania, implementowania, testowania, instalowania i kontroli, eksploatacji i konserwacji, połączone nadrzędną fazą

zarządzania, trwającą przez cały cykl produkcyjny. Podstawowy dokument, zwany *planem weryfikacji i atestacji*, powinien według tej normy opisywać dla każdej fazy oddzielnie następujące aspekty procesu weryfikacji i atestacji:

1) Określenie celu planu, zawierające opis przedsięwzięcia programistycznego i produktów finalnych

2) Nazwy wiążących dokumentów (norm, kontraktów itp.)

3) Definicje terminów oraz wyjaśnienia skrótów i notacji użytych w planie

4) Ogólny opis procesu weryfikacji i atestacji, z uwzględnieniem zasad organizacyjnych, harmonogramu prac, niezbędnych zasobów, przydziału zadań oraz zalecanych narzędzi, technik i metodyki

5) Właściwy plan charakteryzujący przebieg procesu zgodnie z podziałem na fazy według modelu kaskadowego - najważniejsza część dokumentu

6) Zasady prowadzenia sprawozdań z przebiegu weryfikacji i atestacji w całym cyklu produkcyjnym

7) Procedury administracyjne niezbędne do zrealizowania planu.

Główna część planu (pkt 5) dotyczy szczegółowego przebiegu procesu i określa następujące istotne składniki, które powinny być opisane dla każdej fazy oddzielnie:

1) Zadania (weryfikacji i atestacji w tej fazie)

2) Metody i kryteria (użyte do realizacji powyższych zadań)

3) Dane wejściowe i produkty wyjściowe

4) Harmonogram (realizacji zadań)

5) Zasoby (niezbędne do realizacji zadań)

6) Założenia i ryzyko (występujące przy realizacji)

7) Podział ról i odpowiedzialności (między wykonawcami zadań).

Dla każdej fazy podano też wykaz wymaganych dokumentów źródłowych i wynikowych (określonych po angielsku jako inputs i

outputs) oraz minimalny zestaw zadań (czynności) weryfikacyjno-atestacyjnych koniecznych do zrealizowania przy wytwarzaniu oprogramowania dla systemów krytycznych. Podstawowe zadania dotyczą analizy *śladowalności* (tj. umiejscowionych w poprzednich fazach przyczyn i źródeł przyjętych rozwiązań), oceny produktów cząstkowych (powstających w każdej fazie), analizy sprzężeń, planowania testów i wykonania różnego rodzaju testów, lustracji i przeglądów.

Nowa norma [33] będąca jeszcze w stadium opracowywania, zawiera zbiór wskazówek dla osób stosujących zalecenia zawarte w dokumencie [28] i rozszerza ten punkt widzenia o podejście czynnościowe, przedstawione w [35], wyrażające cykl produkcyjny oprogramowania przez wiele wzajemnie powiązanych procesów, składających się z bardziej szczegółowych czynności. Najważniejszym skutkiem tego podejścia dla procesu weryfikacji i atestacji jest wyraźne wyodrębnienie poszczególnych czynności testowania, tzn. testowania komponentów, testu integracyjnego, testowania systemu i testu odbiorczego, dla których podano szereg szczegółowych zaleceń.

6.2 Inne zalecenia dla procesu weryfikacji i atestacji

Dwa wcześniejsze zalecenia dla weryfikacji i atestacji oprogramowania, norma amerykańska [42] i dokumenty europejskie [10,11] opublikowane też w zbiorze [44], nie odbiegają zbytnio w treści od norm IEEE, gdyż są oparte na wspólnych założeniach, których podstawą jest istnienie cyklu produkcyjnego oprogramowania z wyraźnie wyodrębnionymi fazami.

Istotną cechą pierwszej normy jest podział wszystkich czynności wykonywanych w okresie istnienia oprogramowania na czynności produkcyjne i czynności weryfikacyjno-atestacyjne,

oraz - wydzielenie i zidentyfikowanie ich wyników. Nie dającą się przecenić konsekwencją tego spostrzeżenia jest wprowadzony już do praktyki programistycznej *całkowity rozdział zespołu produkcyjnego od zespołu weryfikacyjno-atestacyjnego*, który najczęściej jest kontrahentem zewnętrznym. Do podstawowych czynności weryfikacyjno-atestacyjnych zaliczono:

- w fazie specyfikowania wymagań - opracowanie planu weryfikacji i atestacji, początkowa specyfikacja testu, przegląd i analiza wymagań, przegląd i analiza podręczników użytkowania
- w fazie projektowania - przegląd i analiza projektu, zaimplementowanie lub nabycie narzędzi wspomagających testowanie
- w fazie kodowania i testowania - końcowa specyfikacja testu, przegląd i analiza kodu, przetestowanie kodu
- w fazie instalowania - przeprowadzenie testu odbiorczego
- w fazie eksploatacji i konserwacji - ocena oprogramowania, *testowanie regresywne* (ang. regression testing, polegające na wykonaniu testów sprawdzonych i działających poprawnie dla poprzednich wersji kodu, w celu wykrycia - przez porównanie wyników - defektów wprowadzonych podczas modyfikowania), ocena modyfikacji kodu.

Ważne są również zawarte w tej normie sugestie i przykłady dotyczące stosowania odpowiednich technik. Z wymienionych względów normę tę można uznać za pionierską w propagowaniu właściwego spojrzenia na problematykę weryfikowania i atestowania produktów programistycznych.

To samo podejście zaprezentowano w zaleceniach [10,11]. Dotyczą one, odpowiednio, metodyki i technik weryfikowania i atestowania systemów komputerowych w zastosowaniach krytycznych, przede wszystkim pod względem *rzetelności* oprogramowania (ang. dependability). Wskazano tu ponownie, że główną i

niezbywalną zasadą weryfikacji i atestacji jest całkowita niezależność i rozłączność zespołów: wykonawczego oraz weryfikacyjno-atestacyjnego. *Weryfikacja* - w rozumieniu tego dokumentu - polega także na porównaniu zgodności wyniku każdego kolejnego etapu wytwarzania systemu komputerowego z ustaleniami dokonanymi na poprzednim etapie. *Atestacja* natomiast stanowi stwierdzenie zgodności całego systemu przekazywanego do eksploatacji z wymaganiami określonymi w jego specyfikacji. Zalecenia podane w tych rozdziałach stanowią szereg wskazówek co do planowania i stosowania preferowanych technik w kolejnych fazach wytwarzania wyrobu, jak: specyfikowanie, projektowanie, kodowanie, testowanie modułów, integrowanie modułów, integrowanie sprzętu i oprogramowania i atestowanie całego systemu.

6.3 Testowanie modułów (jednostek, komponentów)

Spośród wielu rodzajów testowania, jednej z głównych czynności w procesie weryfikacji i atestacji, wyróżniono testowanie modułów (zwanych inaczej komponentami lub jednostkami programowymi) i poświęcono mu oddzielną normę [27]. Wydzielono w niej dwie fazy testowania, tj. przygotowanie testu i przeprwadzenie testu, oraz osiem czynności składających się na te fazy. *Przygotowanie testu* polega na:

- zaplanowaniu podejścia, wykorzystania zasobów i harmonogramu
- określeniu testowanych właściwości (na podstawie specyfikacji)
- uszczegółowieniu planu
- zaprojektowaniu zestawu testów
- zaimplementowaniu szczegółowego planu i projektu.

Na *przeprowadzenie testowania* składają się następujące czynności:

- wykonanie testów
- kontrola zakończenia testów
- ocena wyników.

Obie fazy i ich czynności przedstawiono w normie w postaci diagramów przepływu danych, wyliczając dokumenty źródłowe i wynikowe, co znacznie ułatwia jej stosowanie.

Testowanie rozumie się tu (uwaga !) jako celowe wprowadzanie defektów do programu w celu spowodowania awarii; mniej ważne jest stwierdzenie poprawnej pracy programu (!), chodzi bowiem przede wszystkim o sprawdzenie działania programu w sytuacjach nieoczekiwanych. Tak rozumiane testowanie jest przeciwieństwem *uruchamiania* (ang. debugging), które jest czynnością polegającą na analizowaniu awarii w celu zlokalizowania i poprawienia defektów w programie. Dwoma podstawowymi dokumentami powstającymi w czasie testowania są: *specyfikacja projektu testu* i *sprawozdanie z testowania*. W normie podkreślono wyraźnie, że nie jest jej celem zalecanie określonych technik lub narzędzi testowania, choć podano najważniejsze publikacje, w których można znaleźć odpowiednie opisy.

6.4 Przeglądy i lustracje

Do innych ważnych czynności wykonywanych w procesie weryfikacji i atestacji oprogramowania należą przeglądy i lustracje, dlatego poświęcono im n ezależną normę [30]. *Przeglądem* (ang. review) nazywa się tu postępowanie, podczas którego produkt programowy przedstawia się zainteresowanym (np. członkom zespołu, kierownictwu, użytkownikom, nabywcom itp.), w celu zebrania opinii lub odbioru, a *lustracją* (ang. audit) - niezależne sprawdzenie produktu programowego w celu oceny jego

zgodności ze specyfikacją, normami, kontraktem lub innymi kryteriami [19].

W normie zidentyfikowano trzy rodzaje przeglądów, przegląd zewnętrzny (ang. managemnent review), przegląd techniczny (ang. technical review) i inspekcję oprogramowania (ang. software inspection), oraz cztery rodzaje lustracji - lustracja systemu jakości (ang. quality systems audit), lustracja funkcjonalna (ang. functional configuration audit), lustracja fizyczna (ang. functional configuration audit) i lustracja produkcyjna (ang. in-process audit). Wszystkie rodzaje przeglądów opisano według następującego schematu:

- cel, streszczenie i założenia
- kryteria zewnętrzne, zasoby i odpowiedzialność
- dokumenty wejściowe, procedury i dokumenty wyjściowe
- kryteria rozpoczęcia i zakończenia
- podatność na lustracje (ang. auditability).

Lustracje opisano podobnie, koncentrując się na elementach wspólnych - głównie procedurach. Przeglądy znane z innych procesów, np. stosowane w zapewnianiu jakości (por. cz. I artykułu), tj. przegląd wymagań, wstępny i krytyczny przegląd projektu, czy przegląd planu weryfikacji i atestacji, potraktowano tu jak specyficzne zadania, które można wykonać posługując się zasadami opracowanymi w tym dokumencie jako zaleceniami rodzajowymi.

6.5 Proces zarządzania konfiguracją

Celem *zarządzania konfiguracją* oprogramowania w cyklu produkcyjnym jest utrzymywanie informacji o aktualnym składzie wytwarzanego oprogramowania i kontrolowanie jego rozwoju. Dzięki temu uzyskuje się możliwość odtworzenia źródeł pewnych decyzji,

przyczyn konkretnych rozwiązań, śledzenia istotnych właściwości oprogramowania, a więc - cechę którą można nazwać *śladowalnością*. Podstawowym dokumentem dla tego procesu jest *plan zarządzania konfiguracją* (ang. software configuration management plan, SCMP), opisany w normie [21]. Wyróżniono w nim cztery zasadnicze czynności wykonywane przy prawidłowym prowadzeniu tego procesu:

- o indentyfikowanie konfiguracji
- o kontrola konfiguracji
- o rejestrowanie stanu konfiguracji
- o wykonywanie przeglądów i lustracji.

W obecnie przygotowywanej aktualizacji tej normy wyróżniono jeszcze dwie czynności:

- o kontrola sprzężeń
- o kontrola podwykonawców i sprzedawców (tzn. poddostawców).

Te dwie ostatnie czynności są szczególnie istotne przy zlecaniu części oprogramowania do wykonania na zewnątrz. Norma zawiera wskazówki nt. przygotowywania planu zarządzania konfiguracją, tzn. określenia kto, co, kiedy i jak ma robić, aby zapewnić należyte utrzymywanie informacji o konfiguracji oprogramowania. Spośród wielu wprowadzonych pojęć warto zwrócić uwagę na dwa pokrewne. Wyróżniony element konfiguracji podlegający kontroli nazywa się *wersją* (ang. version). Przy każdej modyfikacji nadaje się wersji nowy identyfikator. Przez *wydanie* (ang. release) rozumie się natomiast proces przekazania elementu konfiguracji będącego półproduktem (ang. baseline) - innej organizacji lub na inny poziom tej samej organizacji - w celu przyjęcia modyfikacji.

Główną zaletą uzupełniających tę normę zaleceń [32] (por. też [49]) jest szczegółowe wyjaśnienie poszczególnych

czynności. Każdą z nich rozbito na fazy opisując je za pomocą pytań wskazujących aspekty konieczne do rozważenia przy wykonywaniu tych czynności. Do normy [32] włączono również szereg przykładów dodatkowo ilustrujących zakres jej stosowania. Przykłady dotyczą wytwarzania dokumentu SCMP dla czterch różnych zastosowań:

- krytycznego oprogramowania dla systemów wbudowanych
- małego systemu budowanego eksperymentalnie
- organizacji zajmującej się konserwacją oprogramowania
- systemu sterującego linią produkcyjną.

Użyteczność norm dotyczących zarządzania konfiguracją potwierdzono opracowując zalecenia do stosowania ich w systemach oprogramowania dla elektrowni jądrowych [31].

6.6 Proces wytwarzania dokumentacji według norm amerykańskich

Dobre dokumentowanie wszystkich czynności podczas wytwarzania i kon wacji oprogramowania stanowi nieocenione źródło informacji zarówno dla samych wykonawców jak i dla późniejszych użytkowników tego oprogramowania. Należy pamiętać, że ścisłe rozumienie oprogramowania obejmuje zawsze programy komputerowe wraz z dokumentacją. Ze względu na dużą różnorodność procesów wytwórczych trudno wszakże opracować ogólną normę odnoszącą się do wszystkich czynności. Jedno ze starszych zaleceń, jakie znam [43], traktuje proces dokumentowania jako całość, formułując wiele istotnych wskazówek dotyczących zarządzania tym procesem, pomijając przy tym zagadnienia wykorzystywanych technik i narzędzi.

Ponieważ inny rodzaj dokumentacji jest przydatny w samym wytwarzaniu oprogramowania, a inny w jego eksploatacji, dzieli się ją na dokumentację cyklu produkcyjnego i dokumentację

wyrobu, co w przybliżeniu pokrywa się z podziałem na dokumentację produkcyjną i dokumentację użytkową. *Dokumentacja cyklu produkcyjnego* obejmuje dokumenty będące wynikami poszczególnych faz wytwarzania oprogramowania, a *dokumentacja wyrobu* (dokumentacja użytkowa) - wszystkie dokumenty towarzyszące oprogramowaniu w dostawie (oba te zbiory dokumentów nie muszą być i na ogół nigdy nie są całkowicie rozłączne). Większość przykładowych dokumentów produkcyjnych wyliczono w poprzednich rozdziałach tego artykułu, natomiast do dokumentacji wyrobu zalicza się takie dokumenty, jak podręcznik użytkowania, podręcznik operatora, podręcznik konserwacji itd.

Biorąc pod uwagę jakość dokumentacji, zalecono jej klasyfikowanie na czterech poziomach: minimalnym, wewnętrznym, roboczym i publikacyjnym. W zasadzie, jedynie *poziom publikacyjny* dopuszcza rozpowszechnianie dokumentacji poza obrębem macierzystej instytucji, gdyż odnosi się do programów przeznaczonych do powszechnego użycia oraz programów stanowiących przedmiot publikacji naukowych; strona formalna tej dokumentacji musi być zgodna z wymaganiami i normami opracowującej instytucji. *Poziom roboczy* stosuje się do programów używanych przez różne osoby w tej samej instytucji i niekiedy w innych instytucjach; dokumentacja jest sporządzana i rozpowszechniana w postaci maszynopisu z minimalnym opracowaniem edytorskim. *Poziom wewnętrzny* odnosi się do specjalizowanych programów, które nie mają szerszego zastosowania. Ich dokumentacja może ograniczać się do obszernych komentarzy w wydruku programu, aby ułatwiać jego użytkowanie i modyfikowanie; dokumentacja opisowa jest bardzo skąpa. *Poziom minimalny* jest odpowiedni dla pojedynczych programów powstających na przykład w ciągu miesiąca; dokumentacja ogranicza się do wydruku

programu, notatek projektowych, danych testowych i abstraktu programu.

W omawianych zaleceniach zwrócono też uwagę na przyczyny powstawania dokumentacji o złej jakości (niski priorytet dokumentowania, brak wspomagania i niedostatek zasobów, brak planowania, negatywne postawy personelu itp.) i zaproponowano sposoby rozwiązywania odpowiednich problemów przez właściwe zarządzanie procesem wytwarzania dokumentacji, zwracając uwagę m.in. na istnienie norm i dobrych praktyk.

Jedną z norm IEEE dotyczących dokumentowania cyklu produkcyjnego jest norma [22] obejmująca zagadnienia dokumentacji testowania programów. Opisano w niej ozlem dokumentów, które powinny powstawać jako wynik czynności testowania. Są to:

- plan testowania, rozumiany jako opis zakresu, podejścia, zasobów i harmonogramu zamierzonych działań; określający przedmiot i podmiot testu, testowane właściwości i zadania testu oraz (co bardzo ważne) ryzyko towarzyszące testom
- specyfikacja projektu testu, obejmująca opis podejścia zastosowanego do testowania określonej właściwości i identyfikująca określone testy
- specyfikacja testu, określająca dane wejściowe, przewidywane wyniki i warunki wykonania testu
- specyfikacja procedury testowania, tj. opis ciągu działań prowadzących do wykonania testu
- raport przekazania testu (ang. test item transmittal report), zawierający informację identyfikującą oraz miejsce przechowywania testu i jego status
- kronika testu, obejmująca chronologiczny wykaz istotnych informacji o wykonaniach testu
- raport nieprawidłowości (ang. test incident report), opisujący

każde wydarzenie podczas testowania wymagające dalszych badań
- sprawozdanie z testowania, podsumowujące przebieg testowania i uzyskane wyniki wraz z oceną poszczególnych testów.

Dokumentacja produktu, a więc dokumentacja użytkowa, ma zasadniczo inne przeznaczenie niż dokumentacja produkcyjna, dlatego ujęto ją w zupełnie innej normie [34]. Norma ta zawiera zalecenia dotyczące przygotowywania dokumentów mających służyć użytkownikom do instalowania i eksploatowania programów. Za najważniejsze uznano precyzyjne określenie na początku: właściwości samego oprogramowania (tj. jego funkcji, sposobów użycia, a nade wszystko sprzężeń użytkowych), kręgu odbiorców dokumentacji, zestawu wymaganych dokumentów i trybów ich użycia. Podział dokumentacji ze względu na tryb użycia wynika z różnic, jakie dzielą *dokumenty podstawowe* (instruktażowe), z których użytkownik uczy się eksploatacji i obsługi programu, od tzw. *dokumentów referencyjnych*, z których korzysta się w celu przypomnienia nazw poleceń, wyjaśnienia komunikatów, zgłębienia szczegółów technicznych itp. Dla obu rodzajów dokumentów podano zalecane zawartości, nieco różniące się między sobą. Jest charakterystyczne, że ta norma nie opisuje treści żadnego konkretnego dokumentu (por. [51]) lecz dostarcza wzorca do tworzenia dowolnej dokumentacji użytkowej, której rodzaj zależy od stosowanej praktyki, od ustaleń między wytwórcą a nabywcą oprogramowania itd.

6.7 Dokumentowanie systemu według EWICS

W nieco innej formie sformułowano zalecenia dokumentacyjne w książce [44], której pierwszy rozdział zawiera wymagania dotyczące dokumentowania systemów komputerowych nie tylko podczas ich wytwarzania lecz także w pełnym okresie istnienia,

tzn. w fazach eksploatacji i konserwacji. Choć nie obejmuje on tak ważnych przy wytwarzaniu faz, jak studium realizowalności i projektowanie, pomijając też inny istotny aspekt okresu istnienia systemu - zarządzanie konfiguracją, zawiera wiele ciekawych wskazówek dotyczących dokumentowania systemów komputerowych w zastosowanich krytycznych, przy założeniu, że taki system stanowi jedynie część większej instalacji. Dlatego też najważniejsze fragmenty tego rozdziału dotyczą zarządzania dokumentacją, opisu realizacji systemu z uwzględnieniem aspektów użytkowych, wielostronnej oceny systemu i zarządzania całym przedsięwzięciem. Poszczególne tematy, odpowiadające wybranym aspektom okresu istnienia systemu, dzielą się na podtematy (np. zarządzanie dokumentacją dotyczy jej struktury, standardów dokumentacyjnych i jej konserwowania), które obejmują najważniejsze elementy składowe, wyliczone w postaci punktów (dzielących się często na podpunkty) jako elementów tzw. list kontrolnych (ang. checklist). Przykładowo, jeden z punktów opisujących konserwowanie dokumentacji zaleca stosowanie do tego celu następujących narzędzi: edytory tekstów, generatory raportów, generatory spisu treści i indeksów, komparatory tekstów, systemy archiwowania, bazy danych i poczta elektroniczna.

Może się wydawać, że tak liczne sugestie co do uwzględniania wielu szczegółów dokumentowania są raczej barierą niż ułatwieniem w konkretnych sytuacjach. W rzeczywistości jednak, stanowią one jedynie dobrze uporządkowany układ odniesienia dla wykonawców systemu, którzy sami zadecydują, jakie z nich należy wziąć pod uwagę. Na tę dowolność właśnie kładzie się szczególny nacisk w zaleceniach, ponieważ poszczególne systemy komputerowe o najrozmaitszym przeznaczeniu są tak odmienne,

że schematyczne dokumentowanie identycznych aspektów i w jednakowy sposób byłoby pozbawione sensu. Zalecenia, aczkolwiek bardzo szczegółowe, zostawiają jednak duży margines dowolności.

Aby w dokumentowaniu systemu komputerowego uwzględnić jego niezawodność i bezpieczeństwo - cechy trudne do wyodrębnienia i opisania na papierze - autorzy utrzymali ogólny schemat zaleceń, z podziałem na tematy, podtematy i punkty, proponując w poszczególnych miejscach dodatkowe ujęcie tych elementów opisu, które są związane z niezawodnością i bezpieczeństwem systemu. Na przykład, w opisie systemu jako całości występują oddzielne listy kontrolne, złożone z punktów odnoszących się kolejno do zintegrowanego systemu (opis i ograniczenia zapewnienia bezpieczeństwa, obsługa defektów, zasady konserwacji i prowadzenia napraw, ostrzeganie przed niesprawnościami), samego sprzętu (wielopunktowy program zapewnienia niezawodności i odporności na defekty) i samego oprogramowania (osiąganie bezpieczeństwa i zabezpieczanie przed nieuprawnionym dostępem). Niezwykle ważną cechą zaleceń jest dostarczenie niemal pełnej informacji o możliwościach wytwarzania dokumentacji i korzystania z niej. Po zapoznaniu się z tą partią materiału, wykonawca lub użytkownik powinien nabyć ogólnych umiejętności tworzenia dokumentacji w całym okresie istnienia systemu, przy utrzymaniu dużej dowolności w generowaniu rzeczywiście potrzebnych dokumentów, uwzględniających elementy niezawodności i bezpieczeństwa.

7. ZASTOSOWANIE NORM

Zastosowanie omówionych norm i zaleceń można rozumieć dwojako. Po pierwsze, mogą one stanowić *materiał źródłowy do bardziej szczegółowych opracowań normalizacyjnych* w wielu dziedzinach wykorzystania systemów komputerowych. Szczególnie istotna jest ich rola przy wytwarzaniu tzw. oprogramowania krytycznego, np. w systemach wojskowych [6], kosmicznych [9], jądrowych [1,16,18,31] i przemysłowych [44,45]. Bardziej bezpośrednie korzyści przynosi *użycie tych norm w praktycznym kwalifikowaniu programów komputerowych pod względem jakości*.

W najbliższej mi dziedzinie wytwarzania oprogramowania do zastosowań jądrowych, od kilku lat stosuje się kwalifikowanie programów *na poziomie wewnętrznym* (producenta, kontraktu itp.) lub *na poziomie ogólnokrajowym* (licencjonowanie). W jednej z najważniejszych prób badawczych atestacji oprogramowania przeznaczonego dla elektrowni jądrowych [7], cykl produkcyjny prowadzono według reguł inżynierii oprogramowania oddzielnie przez dwa zespoły wykonawców (uzyskując dzięki temu dodatkowo *efekt zróżnicowania*, ang. diversity), posługując się specyfikacją formalną, programowaniem strukturalnym, statycznymi narzędziami testowania i automatycznym komparatorem w *łożu testowym* (ang. testbed). Cały cykl był nadzorowany przez niezależny zespół weryfikacyjno-atestacyjny, który wykonywał przeglądy specyfikacji, rewizje kodu i testy weryfikacyjne. Dzięki przyjęciu takiej metody pracy uzyskano zdaniem autorów przedsięwzięcia następujące pozytywne skutki:

- znaczne zwiększenie prawdopodobieństwa wykrycia defektów z chwilą ich wprowadzenia
- zmniejszenie kłopotów związanych z doborem sprzężeń programowych
- ogólne zwiększenie poprawności oprogramowania
- zmniejszenie względnego kosztu jednego wiersza kodu (mimo komplikacji związanych z większą liczbą wykonawców).

Licencjonowanie systemów komputerowych stosowanych w elektrowniach jądrowych jest wymagane przez ustwodawstwo wielu krajów. Nie jest jednak jeszcze rozwiązana jednoznacznie sprawa metodyki badań prowadzących do wydania licencji, choć wszystkie podejścia są oparte w mniejszym lub większym stopniu na przedstawionym modelu cyklu produkcyjnego. Przykładowo, włoski urząd ENEA opiera się na definicji jakości oprogramowania IEEE [5], co prowadzi do konieczności wyodrębnienia i oceny poszczególnych cech jakości (atrybutów oprogramowania). Uznano, że możliwe jest wydzielenie takich cech dla każdej z określonych klas systemów komputerowych według podziału zalecanego w normie IEC [17]. Ponieważ zastosowania dotyczą systemów bezpieczeństwa, określenie cech następuje na drodze *wstępnej analizy bezpieczeństwa* (ang. preliminary safety analysis), wykonywanej przez wnioskodawcę na podstawie zaleceń wydanych przez ten urząd (ogólne kryteria projektowe). Ze względu na brak dostatecznie dobrych i powszechnie uznanych metryk służących do oceny tych cech, stosuje się procedury weryfikacyjno-atestacyjne oparte na analizie cyklu produkcyjnego, głównie w fazie projektowej.

Znany urząd zachodnioniemiecki TÜV (niem. Technischer Überwachungs-Verein) opracował własną procedurę licencjonowania systemów komputerowych czasu rzeczywistego z uwzględnieniem bezpieczeństwa [14], która objęła oprócz przemysłu jądrowego także naftowy, chemiczny i in. Punktem wyjścia dla tych prac był również podział zastosowań na klasy pod względem ich wagi dla bezpieczeństwa. Dla każdej z klas sformułowano następnie inny zestaw kryteriów jakości (w postaci cech, które powinien posiadać licencjonowany system). Poszczególnym kryteriom (cechom) przyporządkowano odpowiednie listy kontrolne w postaci pytań, służące do sprawdzania jakości podczas całego cyklu produkcyjnego. Kontroli podlegają: specyfikacja wymagań, projekt systemu i zrealizowany system. Ta metodyka jest w dalszym ciągu rozwijana, głównie pod względem doboru automatycznych narzędzi wspomagających, których brak szczególnie się odczuwa. Na podstawie raportu TÜV wnioskodawca może otrzymać licencję, co z kolei (interesujący szczegół) może zostać zaskarżone do sądu. Podobną metodykę stosuje się w innych krajach, np. we Francji [4], w innych dziedzinach zastosowań, np. w transporcie [36], ze względu na szczególne wymagania dotyczące bezpieczeństwa, dążąc nawet do koordynacji na szczeblu międzynarodowym [3,39].

8. LITERATURA

[1] ANS: Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations. ANS Std 7-4.3.2-1982, American Nuclear Society, La Grange Park (IL), July 1982

[2] Bishop P. (Ed.): Safety Assessment and Design of Industrial Computer Systems - Techniques Directory. Elsevier Science Publishers, London (w druku)

[3] Bloomfield R.E., Ehrenberger W.D.: Licensing issues associated with the use of computers in the nuclear industry. Report EUR-11147, Commission of the European Communities, Luxembourg, 1987

[4] Colling B. et al.: Digital Integrated Protection System SPIN Qualification Process. Report IAEA SM-265/58. International Atomic Energy Agency, Vienna, 1984

[5] Di Marco G., Pasquini A.: Licensing of Real Time Systems in Nuclear Power Plants. Pp. 567-568 [52]

[6] DOD: Defense System Software Development. MIL-Std-2167. US Department of Defense, Washington (DC), 1984

[7] EPRI: Validation of Real-Time Software for Nuclear Plant Safety Applications. Report NP-2646, Electric Power Research Institute, Palo Alto (CA), November 1982

[8] EPRI: Software Development and Maintenace Guidelines. Vol. 1. Topical Report EL-3089. Electrical Power Research Institute, Palo Alto (CA), May 1983

[9] ESA: Software Engineering Standards. European Space Agency, Darmstadt, 1983

[10] EWICS TC7: Guidelines for Verification and Validation of Safety-Related Software. Computers and Standards, Vol. 4, pp. 33-41, 1985

[11] EWICS TC7: Techniques for Verification and Validation of Safety-Related Software. Computers and Standards, Vol. 4, pp. 101-112, 1985

[12] Gannon J.D., Katz E.E., Basili V.R.: Metrics for Ada Packages - An Initial Study. Communications of the ACM, Vol. 29, No. 7, pp. 616-623, 1986

[13] Gasik S.: Wprowadzenie do programometrii. Informatyka, str. 9-13, nr 11-12, 1985

[14] Glöe G., Westhäusser R.: Assessment and Licensing of Control Computers in the FRG. Pp. 555-563 [50]

[15] Hoare C.A.R.: An Overview of Some Formal Methods for Program Design. IEEE Computer. Vol. 20, No. 9, pp. 85-91, September 1987

[16] IAEA: Manual on Quality Assurance for Computer Software. Final Version, International Atomic Energy Agency, Vienna, April 1987

[17] IEC: Application of Digital Computers to Nuclear Reactor Instrumentation and Control. IEC 643, International Electrotechnical Commission, Geneva, 1979

[18] IEC: Software for Computers in the Safety Systems of Nuclear Power Stations. IEC 880, International Electrotechnical Commission, Geneva, 1986

[19] IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 729-1989

[20] IEEE Standard for Software Quality Assurance Plans. IEEE Std 730-1989

[21] IEEE Standard for Software Configuration Management Plans. IEEE Std 828-1983

[22] IEEE Standard for Software Test Documentation. IEEE Std 829-1983

[23] IEEE Guide to Software Requirements Specification. IEEE Std 830-1984

[24] IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE Std. 982.1-1988

[25] IEEE Guide to Standard Dictionary of Measures to Produce Reliable Software. IEEE Std. 982.2-1988

[26] IEEE Guide to Software Quality Assurance Planning. IEEE Std 983-1986

[27] IEEE Standard for Software Unit Testing. IEEE Std 1008-1986

[28] IEEE Standard for Software Verification and Validation Plans. IEEE Std 1012-1986

[29] IEEE Recomended Practice for Software Design Descriptions. IEEE Std 1016-1987

[30] IEEE Standard for Software Reviews and Audits. IEEE Std 1028-1988

[31] IEEE Recommended Practice for Application of IEEE Std 828 to Nuclear Power Generating Stations. IEEE Std 1033-1985

[32] IEEE Guide to Software Configuration Management Planning. IEEE Std 1042-1987

[33] IEEE Guide for Software Verification and Validation Plans. IEEE Project 1059

[34] IEEE Standard for Software User Documentation. IEEE Std 1063-1986

[35] IEEE Standard for Software Life Cycle Processes. IEEE Project 1074

[36] Krebs H.: Verification of Safety Related Programs for a Maglev System. Working Paper 520, European Workshop on Industrial Computer Systems, TC 7, 21 July 1986

[37] Lamport L.: Uproszczone specyfikowanie systemów współbieżnych. Informatyka, nr 10, 11-12, 1988; nr 1, 1989

[38] Ludewig J.: Languages, Methods, and Tools for Software Specification. Pp. 225-256 [52]

[39] Malagardis N.: Certification and Validation in Europe. Pp. 545-553 [52]

[40] Martin R.J., Osborne W.M.: Guidance on Software Maintenance. NBS Special Publication 500-106, National Bureau of Standards, Washington (DC), December 1983

[41] MUSE Project: A list of references on metrics. Internal Report. ESPRIT Project No. 1257, Brussels, 1989

[42] NBS: Guideline for Lifecycle Validation, Verification, and Testing of Computer Software. FIPS Publication 101, National Bureau of Standards, Washington (DC), June 1983

[43] Neumann A.J.: Management Guide for Software Documentation. NBS Special Publication 500-87, National Bureau of Standards, Washington (DC), January 1982

[44] Redmill F. (Ed.): Dependability of Critical Computer Systems. Vol. 1. Elsevier Applied Science, London, 1988

[45] Redmill F. (Ed.): Dependability of Critical Computer Systems. Vol. 2. Elsevier Applied Science, London, 1989

[46] Redmill F., Barber D.R.: A Guideline for Conducting a Feasibility Study and Preparing the Feasibility Study Report. Document SD0030, British Telecomm International, Dept. NPD/CA, London, December 1986

[47] Smith I.C.: The Integrity of Computer Based Safety Systems. Pp. 163-173 [52]

[48] Turski W.M.: Kanoniczny krok procesu programowania. Informatyka, nr 8, 9, 1988

[49] Updike-Rumley M.: IEEE Standardization of Configuration Management Issues. Pp. 537-544 [52]

[50] Vincent J., Waters A., Sinclair J.: Software Quality Assurance. Vol. 1. Practice and Implementation. Prentice-Hall, Englewood Cliffs (NJ), 1988

[51] Zalewski J.: Dokumentacja oprogramowania - terminologia. Informatyka, nr 3, 5, 8, 10, 1984

[52] Zalewski J., Ehrenberger W. (Eds.): Hardware and Software for Real Time Process Control., Proc. IFIP/IFAC/EWICS Conf. Warsaw, Poland, 30 May - 1 June 1988, North-Holland, Amsterdam, 1989

9. DODATEK

W niniejszym dodatku zawarto definicje atrybutów jakości oprogramowania według norm IEEE [19], zaleceń IAEA [16] i praktyki inżynierskiej [50].

9.1 Norma IEEE 729

Dyspozycyjność (ang. availability) - stopień w jakim system lub jego komponent jest dostępny i zdolny do pracy, gdy żąda się jego użycia.

Efektywność (ang. efficiency) - stopień wykorzystania zasobów przez system lub jego komponent w celu spełnienia wymaganych funkcji.

Elastyczność (ang. flexibility) - łatwość z jaką można modyfikować system lub jego komponent w celu użycia go w zastosowaniu lub w środowisku innym niż to, do którego był zaprojektowany.

Integralność (ang. integrity) - stopień w jakim system lub jego komponent jest chroniony przed nieupoważnionym dostępem lub zmodyfikowaniem programów komputerowych lub danych.

Konserwowalność (ang. maintainability) - łatwość z jaką system programowy lub jego komponent może być modyfikowany w celu poprawienia defektów, dostosowania do zmian środowiskowych, zwiększenia wydajności lub polepszenia innych parametrów.

Krytyczność (ang. criticality, severity) - stopień w jakim wymaganie, moduł, błąd, defekt, awaria lub inny czynnik wpływa na opracowanie lub działanie systemu.

Modularność (ang. modularity) - stopień zawartości w systemie lub programie komputerowym dyskretnych komponentów, takich że wymiana jednego komponentu ma minimalny wpływ na pozostałe.

Nadmiarowość (ang. redundancy) - obecność dodatkowych komponentów w systemie, przeznaczonych do wykonywania tych samych lub podobnych funkcji, w celu zapobiegania uszkodzeniom lub przywrócenia stanu sprzed uszkodzenia.

Niesprzeczność (ang. consistency) - stopień jednolitości i znormalizowania części systemu lub jego komponentu.

Niezawodność (ang. reliability) - zdolność systemu lub jego komponentów do wykonywania wymaganych funkcji przez określony czas, w ustalonych warunkach.

Odporność (ang. robustness) - stopień w jakim system lub jego komponent może działać poprawnie w obecności nieważnych sygnałów wejściowych lub niesprzyjających warunków środowiskowych.

Ogólność (ang. generality) - szerokość zakresu funkcji wykonywanych przez system lub jego komponent.

Poprawność (ang. correctness) - stopień w jakim system programowy lub jego komponent jest wolny od defektów projektowych i defektów kodu.

Prostota (ang. simplicity) - stopień zrozumiałości i jasności projektu lub implementacji systemu lub komponentu.

Przenośność (ang. portability) - łatwość z jaką system lub jego komponenty mogą być przenoszone z jednego sprzętu lub środowiska programowego do innego.

Realizowalność (ang. feasibility) - stopień w jakim przy istniejących ograniczeniach mogą być zrealizowane wymagania, projekt lub plany dotyczące systemu lub jego komponentu.

Rozszerzalność (ang. extendability) - łatwość modyfikacji systemu lub jego komponentów w celu zwiększenia wielkości pamięci lub zdolności funkcjonalnych.

Spoistość (ang. cohesion) - sposób i stopień wzajemnego powiązania zadań realizowanych przez pojedynczy moduł programowy.

Śladowalność (ang. traceability) - stopień w jakim można ustalić związki o charakterze poprzednik-następnik lub nadrzędny-podległy między dwoma lub wieloma produktami procesu wytwórczego.

Testowalność (ang. testability) - stopień w jakim system lub jego komponent ułatwia ustalenie kryteriów testowania i przeprowadzenie testów w celu stwierdzenia, czy spełnione są te kryteria.

Tolerowanie błędów (ang. error tolerance) - zdolność systemu lub jego komponentu do normalnego działania pomimo obecności błędnych sygnałów wejściowych.

Tolerowanie defektów (ang. fault tolerance) - zdolność systemu lub jego komponentu do normalnego działania pomimo istnienia defektów sprzętowych lub programowych.

Użyteczność (ang. usability) - łatwość z jaką użytkownik może nauczyć się obsługi, przygotowywać dane wejściowe i interpretować wyniki działania systemu lub jego komponentów.

Wielokrotne użycie (ang. reusability) - stopień w jakim moduł programowy może być wykorzystany w więcej niż jednym programie komputerowym.

Współdziałanie (ang. interoperability) - zdolność dwóch lub większej liczby składowych lub komponentów do wymiany informacji i jej użycia.

Wydajność (ang. performance) - stopień w jakim system lub jego składowa wykonuje swoje funkcje przy narzuconych ograniczeniach, takich jak szybkość, dokładność, wykorzystanie pamięci.

Zabezpieczenie (ang. security) - ochrona sprzętu komputerowego lub oprogramowania przed przypadkowym lub zamierzonym czynnym, użyciem, zmodyfikowaniem, zniszczeniem lub ujawnieniem.

Zespolenie (ang. coupling) - sposób i stopień współdziałania modułów programowych.

Zgodność (ang. compatibility) - zdolność dwóch lub wielu systemów lub ich komponentów do wymiany informacji.

Złożoność (ang. complexity) - stopień komplikacji lub trudności projektu lub implementacji systemu lub jego komponentu.

9.2 Zalecenia IAEA

Czytelność (ang. readability) - stopień w jakim na podstawie odczytania kodu można wnioskować o jego funkcjach i projekcie, co wyraża się na przykład przez użycie znaczących nazw zmiennych, zaopatrzenie kodu w obszerne komentarze itp.

Dokładność (ang. accuracy) - stopień w jakim wyniki wykonania kodu odpowiadają pierwotnym zamierzeniom.

Efektywność (ang. efficiency) - stopień w jakim kod, wykonując wymagane funkcje, wykorzystuje zasoby. Może to oznaczać konieczność wyboru odpowiednich konstrukcji języka źródłowego w celu uzyskania minimalnej objętości kodu wynikowego, zastosowanie takich algorytmów, które są najszybciej wykonywane, lub - przyjęcie specjalnych metod upakowania informacji w pamięci.

Ergonomiczność (ang. human engineering) - stopień w jakim kod spełnia swoje funkcje nie powodując strat czasu i energii użytkowników oraz nie zmniejszając ich dobrego samopoczucia. W tym celu dane wejściowe i wyniki powinny być zrozumiałe i jednoznaczne, powinny uniemożliwiać mylną interpretację.

Integralność (ang. integrity) - stopień w jakim poddany jest kontroli dostęp do oprogramowania lub danych przez nieupoważnione osoby.

Komunikatywność (ang. communicativeness) - stopień w jakim postać i treść danych wejściowych i wyników działania kodu mogą

być zrozumiałe i przyswojone. Atrybutem tym obejmuje się często także zrozumiałość i przyswajalność standardowych protokołów i sprzężeń służących do połączenia programu z innym niezależnym systemem.

Konserwowalność (ang. maintainability) - stopień w jakim kod jest podatny na aktualizację w celu spełnienia nowych wymagań, usunięcia wad lub przeniesienia do innego lecz podobnego systemu komputerowego.

Modyfikowalność (ang. modifiability) - właściwość projektu lub kodu ułatwiająca wprowadzanie do niego określonych zmian.

Niesprzeczność (ang. consistency) - stopień w jakim program można uważać za jednolity pod względem przyjętej notacji, terminologii, symboliki, komentarzy i technik realizacji.

Niezależność od urządzeń (ang. device independence) - zdolność kodu do uniknięcia skutków wymiany sprzętu komputerowego lub urządzeń zewnętrznych. W tym celu należy zidentyfikować, wyodrębnić i zminimalizować segmenty kodu odnoszące się bezpośrednio do określonych urządzeń.

Niezawodność (ang. reliability) - stopień w jakim kod jest zdolny do nieustannego wykonywania zamierzonych funkcji.

Odporność (ang. robustness) - stopień w jakim kod może kontynuować zamierzone działanie, mimo pogwałcenia niektórych założeń specyfikacji. Znaczy to, że program będzie przyjmował dane wejściowe lub wyniki pośrednie, które przekraczają zakresy, mają inny format lub typ niezgodny z założonym, bez ograniczania swoich funkcji.

Poprawność (ang. correctness) - stopień w jakim program spełnia swoją specyfikację, co rozumie się także jako zdolność programu do uzyskiwania założonych wyników przy założonych danych wejściowych.

Prostota (ang. simplicity) - właściwość oprogramowania polegająca na zaimplementowaniu jego funkcji w możliwie najbardziej zrozumiały sposób. Zwykle wymaga to unikania technik zwiększających złożoność.

Przenośność (ang. portability) - stopień w jakim kod może działać dobrze w konfiguracji komputerowej innej niż ta, dla której jest przeznaczony.

Rozliczalność (ang. accountability) - stopień w jakim można mierzyć użycie zasobów komnputerowych przez moduł lub program. Znaczy to, że krytyczne segmenty kodu mogą być wyposażone w sondy, które mierzą czas wykonania, wykonywanie poszczególnych rozgałęzień itd. Segmenty te powinny być wcielone do kodu podczas kompilacji lub asemblacji warunkowej.

Rozszerzalność (ang. augmentability) - stopień w jakim można rozbudowywać kod pod względem właściwości funkcjonalnych lub zajętości pamięci.

Samoopisywalność (ang. self-descriptiveness) - stopień w jakim wydruk kodu zawiera informacje, na podstawie których czytelnik może określić lub zweryfikować jego cele, założenia, ograniczenia, dane wejściowe, wyniki, komponenty i wersję.

Samowystarczalność (ang. self-containedness) - stopień w jakim kod wykonuje samodzielnie wszystkie swoje jawne i niejawne funkcje, np. inicjowanie, kontrole, diagnostykę itp.

Strukturalność (ang. structuredness) - właściwość kodu polegająca na jego podziale na niezależne części według ściśle określonego wzorca. Wymaga to projektowania programu w sposób uporządkowany i systematyczny, np. metodą zstępującą lub dekompozycji funkcji, oraz kodowania za pomocą dobrze określonych struktur sterujących, np. pętli DO-WHILE i instrukcji warunkowych IF-THEN-ELSE.

Śladowalność (ang. traceability) - atrybut oprogramowania umożliwiający śledzenie jego rozwoju, od specyfikacji wymagań aż do implementacji, w określonym środowisku produkcyjnym i eksploatacyjnym.

Testowalność (ang. testability) - stopień w jakim kod ułatwia tworzenie planu testowania, specyfikacji testów, procedur testujących, i ich realizowanie. W praktyce sprzyja temu modularna budowa programu i dobrze określone sprzężenia, które mogą być testowane niezależnie.

Użyteczność (ang. usability) - wysiłek wymagany do nauczenia się, eksploatowania, przygotowywania danych i interpretowania wyników programu.

Wielokrotnego użycia (ang. reusability) - stopień w jakim program lub jego części mogą być używane w innych zastosowaniach. Właściwość ta wiąże się z zakresem funkcji wykonywanych przez program i ich obudowaniem.

Współdziałanie (ang. interoperability) - wysiłek wymagany do sprzęgnięcia systemu z innym niezależnym systemem.

Zdolność obsługi błędów (ang. error handling capability) - dostosowanie kodu do reagowania na awarie sprzętu lub oprogramowania i błędne polecenia operatora w taki sposób, aby wydajność systemu nie spadała gwałtownie lecz łagodnie.

Zrozumiałość (ang. understandability) - stopień w jakim funkcje kodu są jasne dla czytelnika. W praktyce uzyskuje się to przez niesprzeczne użycie nazw i symboli, samoopisywalność modułów, uproszczenie struktur sterujących i dostosowanie ich do określonych norm itp. Kod nie powinien zawierać ukrytych znaczeń i charakterystyk, które stają się jasne dopiero po dłuższym użyciu.

Zupełność (ang. completeness) - właściwość kodu polegająca na tym, że wykonuje on wszystkie wymagane funkcje w sposób pełny. Wynika stąd, że dostępna jest pełna dokumentacja, a wymagane funkcje są zakodowane zgodnie z projektem.

Zwięzłość (ang. conciseness) - właściwość kodu polegająca na zrealizowaniu wymaganych funkcji w sposób maksymalnie oszczędny pod względem długości kodu, przy braku elementów zbędnych i nadmiarowych.

9.3 Definicje inżynierskie

Dokładność (ang. accuracy) - osiągnięcie wyganej precyzji obliczeń pośrednich i wyników.

Dostępność (ang. acces control and audit) - możliwość sterowania dostępem i lustrowania dostępu do oprogramowania i danych.

Efektywność (ang. efficiency) - ilość zasobów komputerowych i objętość kodu wymagane przez program do wykonania swoich funkcji.

Elastyczność (ang. flexibility) - miara wysiłku wymaganego do zmodyfikowania eksploatowanego programu.

Integralność (ang. integrity) - stopień w jakim można kontrolować dostęp do oprogramowania lub danych przez nieupoważnione osoby.

Komunikatywność (ang. communicativeness) - stopień przyswajalności danych wejściowych i wyników działania oprogramowania.

Konserwowalność (ang. maintainability) - miara wysiłku wymaganego do umiejscowienia i usunięcia defektu w eksploatowanym programie.

Łatwość obsługi (ang. training) - podatność oprogramowania na opanowanie przez użytkownika.

Modularność (ang. modularity) - posiadanie budowy mającej charakter zestawu maksymalnie niezależnych modułów.

Niesprzeczność (ang. consistency) - jednolitość technik i notacji projektu i implementacji.

Niezależność (ang. <software system, machine> independence) - zdolność oprogramowania do wykonywania swych funkcji bez względu na środowisko programowe lub sprzęt.

Niezawodność (ang. reliability) - stopień wiarygodności z jaką program wykonuje swoje funkcje z wymaganą precyzją.

Ogólność (ang. generality) - miara zakresu wykonywanych funkcji.

Poprawność (ang. correctness) - stopień w jakim program spełnia swoją specyfikację i odpowiada potrzebom użytkownika.

Prostota (ang. simplicity) - zaimplementowanie funkcji w najbardziej zrozumiały sposób.

Przenośność (ang. portability) - miara wysiłku wymaganego do przeniesienia programu z jednej konfiguracji sprzętowej lub środowiska do innej konfiguracji lub środowiska.

Rozszerzalność (ang. expandability) - możliwość zwiększenia pojemności pamięci lub ilości funkcji obliczeniowych.

Samoopisywalność (ang. self-descriptiveness) - wyjaśnianie sposobu zaimplementowania funkcji.

Sondowalność (ang. instrumentation) - możliwość wykonywania pomiarów użycia określonych konstrukcji lub - identyfikowania błędów.

Standardowość (ang. <communication, data> commonality) - stopień wykorzystania standardowych protokołów, sprzężeń i reprezentacji danych.

Szybkość (ang. execution efficiency) - czas wykonywania kodu.

Śladowalność (ang. traceability) - podatność oprogramowania na odtworzenie jego rozwoju od formułowania wymagań do implementacji, w określonym środowisku produkcyjnym i eksploatacyjnym.

Testowalność (ang. testability) - miara wysiłku wymaganego do przetestowania programu w celu upewnienia się, że wykonuje on pożądane funkcje.

Tolerowanie błędów (ang. error tolerance) - ciągłość działania w warunkach odbiegających od nominalnych.

Użyteczność (ang. usability) - miara wysiłku wymaganego do nauczenia się obsługi, eksploatowania, przygotowywania danych wejściowych i interpretowania wyników działania programu.

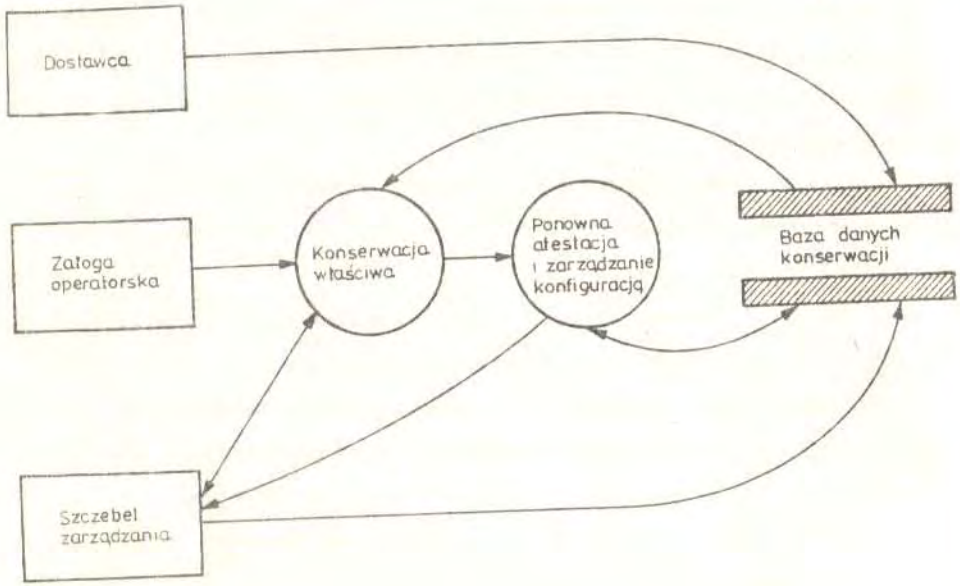
Używalność (ang. operability) - stopień określoności procedur eksploatowania oprogramowania.

Wielokrotnego użycia (ang. reusability) - stopień w jakim program może być używany w innych zastosowaniach związanych z zakresem jego funkcji i obudowaniem.

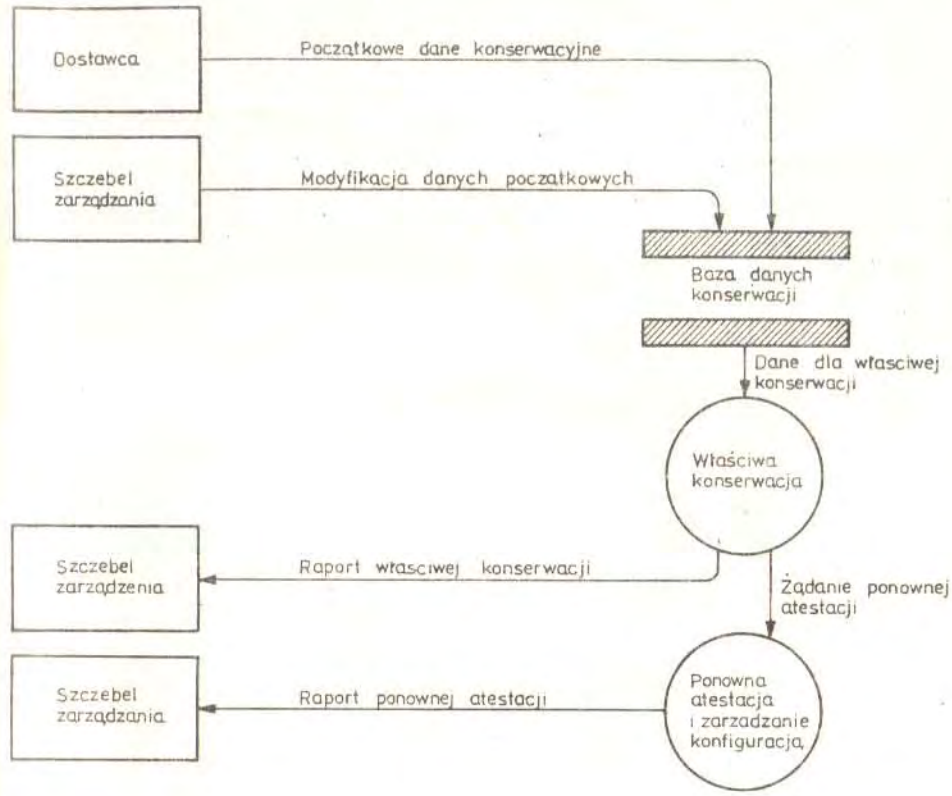
Współdziałanie (ang. interoperability) - miara wysiłku wymaganego do zespolenia jednego systemu z innym.

Zupełność (ang. completeness) - pełne zaimplementowanie wymaganych funkcji.

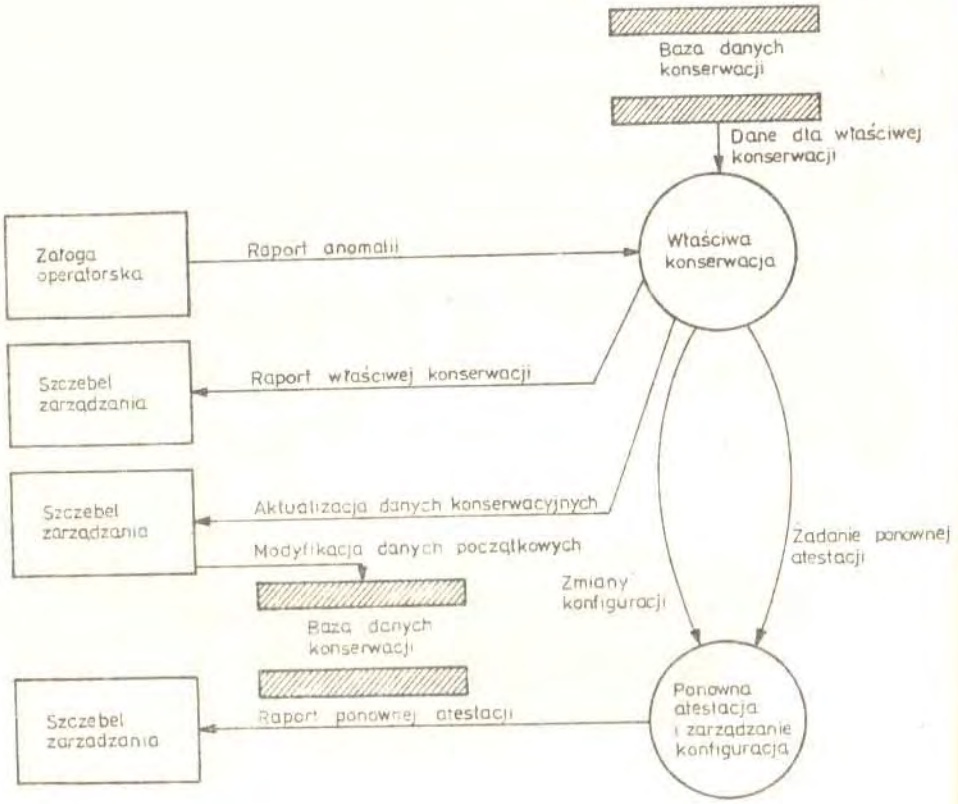
Zwięzłość (ang. conciseness) - zaimplementowanie funkcji przy użyciu minimalnej długości kodu.



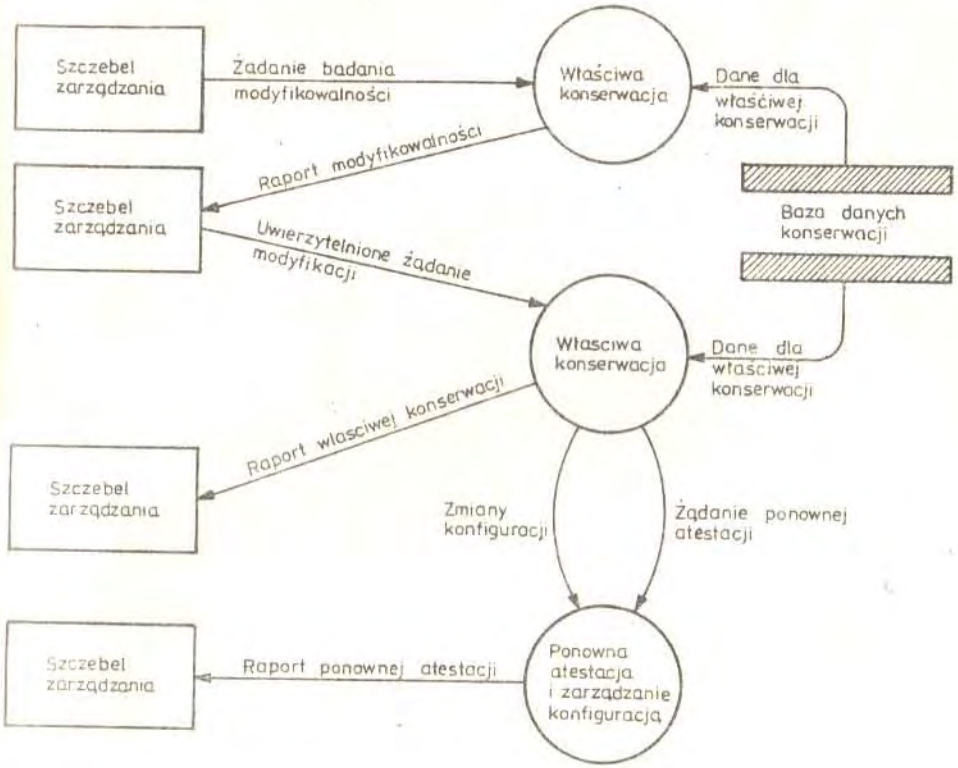
Rys. 1. Zasada procesu konserwacji



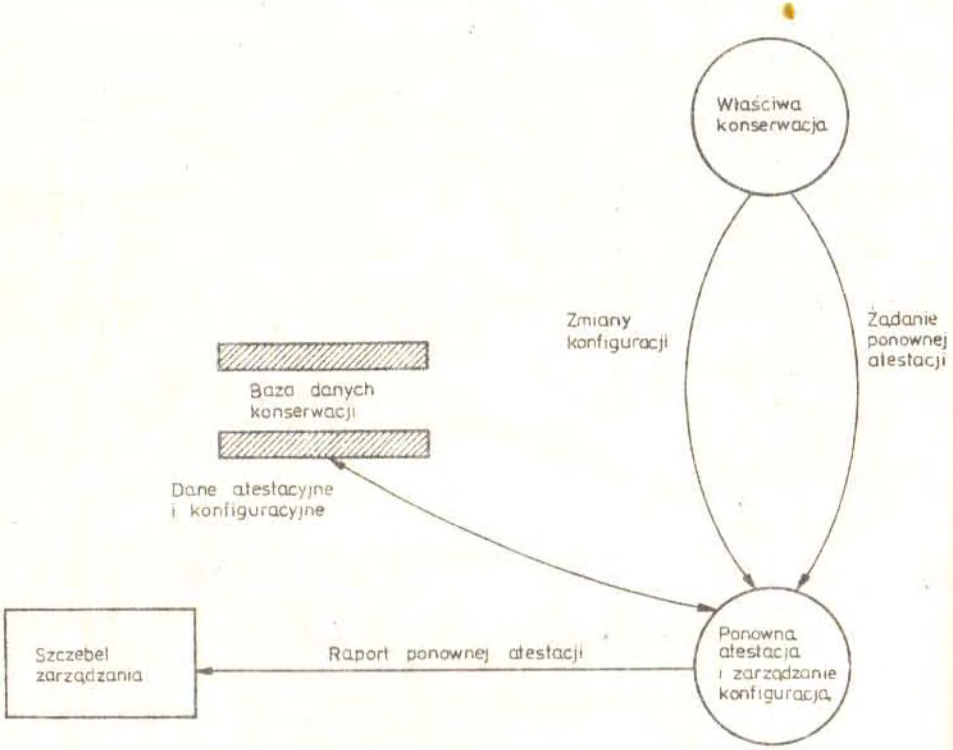
Rys. 2. Konserwacja prewencyjna



Rys. 3. Konserwacja korekcyjna



Rys. 4. Konserwacja perfekcyjna i adaptacyjna



Rys. 5. Ponowna atestacja i zarządzanie konfiguracją