POLSKIE
TOWARZYSTWO
INFORMATYCZNE

ODDZIAŁ GÓRNOŚLĄSKI

WERYFIKACJA I ATESTACJA OPROGRAMOWANIA

OPRACOWAŁ:

JACEK IRLIK

KATOWICE, WRZESIEŃ 1989

1. Wprowadzenie

W miarę wzrostu znaczenia zastosowań systemów informatycznych oraz wzrostu stopnia ich złożoności coraz poważniejszym - ze względu na ewentualne konsekwencje - staje się problem gwarancji, że systemy te funkcjonować będą w sposób zgodny z określonymi, wynikającymi z celu konkretnych zastosowań wymaganiami. Sposób funkcjonowania systemu wyznacza jego oprogramowanie. Zasadniczą więc rolę w udzieleniu wspomnianej gwarancji ma proces zwany procesem weyfikacji i atestacji oprogramowania (software verification & validation).

Znaczenie procesu weryfikacji i atestacji w układzie wszystkich procesów związanych z wytwarzaniem oprogramowania widzi się ostatnio coraz szerzej. Jeżeli poprzednio (por. (1)) proces weryfikacji i atestacji miał udzielaać odpowiedzi na pytania:

- czy projekt oprogramowania przewiduje, a ostateczna jego implementacja zawiera, wszystkie wymagane funkcje oraz czy implementacja ta jest prawidłowa,
- czy wykonana implementacja danego projektu zapewnia wymagana dokładność obliczeń, wymagane czasowe parametry przetwarzania oraz czy będzie funkcjonować w ramach założonych zasobów,

to obecnie istnieją tendencje, aby proces ten umożliwiał również rozstrzygnięcie kwestii:

- czy system może zostać wytworzony w ramach zaplanowanych środków oraz w zaplanowanym terminie,
- czy system będzie po instalacji podatny na ewentualne

modyfikacje lub rozbudowę,

- czy działanie systemu będzie wystarczająco wydajne w środowisku operacyjnym użytkownika.

Wzrost znaczenia procesu weryfikacji i atestacji oraz jego zadań odzwierciedla się w tendencji, aby:

- proces ten stał się procesem integralnie związanym z innymi procesami w pełnym cyklu rozwoju oprogramowania,
- proces ten stał się procesem organizacyjnie sformalizowanym, obejmującym precyzyjnie zdefiniowane i zaplanowane czynności,
- określone czynności tego procesu były wykonywane przez zespoły wyspecjalizowane, niezależne od zespołów realizujących zadania w procesach wytwarzania oprogramowania.

Uwzględniając powyższe tendencje nie można kształtować modelu procesu weryfikacji i atestacji w oderwaniu od modelu pozostałych procesów zachodzących w pełnym cyklu rozwoju oprogramowania.

W niniejszej pracy układem odniesienia będzie model procesów zachodzących w cyklu życia oprogramowania, zaproponowany przez IEEE (2).

Model ten określa procesy, które można wyróżnić w pełnym cyklu życia oprogramowania oraz grupuje je w sfery:

- procesów zarządzania,
- procesów rozwoju,
- procesów integracyjnych.

Proces weryfikacji i atestacji jest jednym z procesów sfery integracyjnej.

Wspomniane procesy można obserwować w kolejnych fazach rozwoju

oprogramowania. Układ występujących w określonym przedsięwzięciu faz oraz ich znaczenie zależy w znacznym stopniu od jego konkretów. Można jednak - z uwzględnieniem wagi każdej z faz - wyróżnić zawsze:

- fazę koncepcji (concept phase),
- fazę określania wymagań (requirements phase),
- fazę projektowania (design phase),
- fazę implementacji (implementation phase),
- fazę testowania (test phase),
- fazę instalowania (installation & checkout phase)

oraz fazę eksploatacji i pielęgnacji (operation & maintenance phase), w której ponownie - w sposób iteracyjny - pdgrywaja rolę procesy występujące pierwotnie w fazach poprzednich.

Wykonywanie zadań w procesach rozwoju oraz ich dokumentowanie prowadzi do wytworzenia kolejnych stadiów rozwojowych powstającego oprogramowania oraz ich dokumentacji.

Podstawowe znaczenie dla procesu weryfikacji i atestacji mają wśród rodzajów tworzonej dokumentacji:

- specyfikacja wymagań (software requirements specification,SRS),
- dokumentacja projektowa (software design description, SDD),
- dokumentacja kodu źródłowego (source code documentation).

Proces weryfikacji i atestacji obejmuje zadania polegające na badaniu toku procesów rozwoju oraz ocenie wyników kolejnych faz (dokumentacja powstająca w kolejnych fazach oraz kod wynikowy) - dokonywanych według ustalonych dla danego przedsięwzięcia kryteriów.

W literaturze formułowane są rozmaite definicje zadań weryfikacji i atestacji oprogramowania. Zgodnie z przyjętym przez nas

modelowym ujęciem cyklu życia oprogramowania określenie tych zadań jest integralna częścia modelu oraz związane jest z ustalonymi kryteriami oceny wyników. Prowadzi to do nastepujacych definicji.

Przez weryfikację (verification) rozumieć będziemy czynności pozwalające na stwierdzenie, że wynik kolejnej fazy rozwoju oprogramowania jest prawidłowym odzwierciedleniem wyniku fazy poprzedniej.

Przez atestację (validation) rozumieć będziemy natomiast czynności pozwalające na stwierdzenie, że oprogramowanie wynikowe spełnia wymagania zawarte w jego pierwotnej specyfikacji (SRS).

Definicje takie przyjęte zostały m.in. w standardzie dotyczącym wymagań dla planów weryfikacji i atestacji (3) (o których w p.5. niniejszej pracy).

Proces weryfikacji i atestacji oprogramowania wymaga stosowania określonych dla danego przedsięwzięcia metod, technik oraz narzędzi. Powinien on zostać zaplanowany i zorganizowany w sposób uwzględniający właściwą dla tego przedsięwzięcia integrację z procesem rozwoju oprogramowania.

Zagadnieniom tym poświęcona jest niniejsza praca.

2. Znaczenie dokumentacji w procesie weryfikacji i atestacji oprogramowania

2.1. Uwagi ogólne

Jednym z podstawowych wymogów prowadzenia rozwoju oprogramowania w sposób systematyczny oraz efektywnego procesu jego weryfikacji i atestacji jest staranne dokumentowanie wyników wykonywanych

zadań oraz zachodzących zdarzeń - w pełnym cyklu życia oprogramowania.

Organizacja całokształtu tych procesów powinna w jednoznaczny sposób ustalać:

- podlegające dokumentowaniu czynności i zdarzenia,
- podlegające dokumentowaniu wyniki otrzymywane w toku procesów,
- zakres treści i formę tworzonej dokumentacji,
- zasady zarządzania dokumentacją:
    - odpowiedzialność za jej przygotowanie,
    - uprawnienia do zatwierdzania,
    - obieg dokumentacji,
    - sposób jej archiwowania i ochrony.

Prowadzenie prawidłowej gospodarki dokumentacją jest jednym z wymogów zapewnienia jakości oprogramowania (8). Wymagania te odnoszą się oczywiście w szczególności również do czynności i zdarzeń mających miejsce w toku procesu weryfikacji i atestacji (por. p.5). Dla samego procesu weryfikacji i atestacji szczególne znaczenie ma natomiast dokumentacja wyników kolejnych faz rozwoju oprogramowania - stanowi ona bowiem podstawowy przedmiot badań w toku tego procesu.

W szczególności w toku tego procesu uwaga koncentruje się na trzech rodzajach dokumentacji:

- specyfikacji wymagań (SRS),
- dokumentacji projektowej (SDD),
- dokumentacji kodu źródłowego.

Niniejszy paragraf poświęcony jest bliższemu omówieniu tych właśnie rodzajów dokumentacji.

2.2. Specyfikacja wymagań (SRS)

Specyfikacja wymagań jest jednym z podstawowych dokumentów w t
wytwarzania oprogramowania. Jest tak dlatego, że z założe
zawiera ona informacje stanowiące ostateczne kryterium oceny, c
wytworzone oprogramowanie spełnia ustalone na początku proce
oczekiwania i czy może tym samym zostać zaakceptowane prz
przyszłego użytkownika. Jest ona więc podstawowym kryteri
atestacji. Ma ona również duże znaczenie prawne. W przypad
oprogramowania wytwarzanego na zamówienie jest podstawą zawarci
kontraktu oraz rozstrzygnięcia, czy zostało ono wykonane zgodni
z jego postanowieniami.

Specyfikacja wymagań dla oprogramowania może stanowić czę
(4)
szerszego dokumentu, specyfikacji wymagań dla systemu (p.
odnoszącej się również do sprzętu oraz fizycznego kontekst
systemu informatycznego. Uwagę naszą ograniczymy do wymagań dl
oprogramowania.

Stawia się na ogół kilka postulatów ogólnych w stosunku do SRS.
Dokumentacja ta powinna być:

1) Jednoznaczna. Oznacza to, że każde z wyspecyfikowanych w nie
wymagań powinno mieć tylko jedą interpretację semantyczną.
Postulat ten jest w istocie równoznaczny z postulatem możliwie
szerokiego stosowania sformalizowanych języków zapisu tych
wymagań.

2) Zupełna. Oznacza to, że:

- żadne wymaganie istotne nie powinno zostać pominięte,
- powinna zostać określona reakcja systemu (wynik przetwarzania,
komunikat, sygnał) na każde przewidywane pobudzenie (dane

wejściowe, komunikat, przerwanie zewnetrzne),

- w stosunku do żadnego z wymagań nie powinny wystepować stwierdzenia typu "zostanie określone później",
- dokumentacja powinna być kompletna redakcyjnie, tzn. zawierać wszystkie elementy, do których występują w niej odwołania - np. rysunki, tablice t t.p.,

3) Spójna. Oznacza to, że nie wystapią rozbieżności lub sprzeczności pomiędzy jakimikolwiek wyspecyfikowanymi w niej wymaganiami.

4) Weryfikowalna. Oznacza to, że dla każdego z wyspecyfikowanych wymagań istnieje i jest określony sposób jednoznacznego rozstrzygnięcia (w skończonym czasie), czy wymaganie to zostało spełnione.

5) Modyfikowalna. Jest to postulat w stosunku do struktury redakcyjnej dokumentacji. Struktura ta powinna umożliwiać łatwe wprowadzanie zmian.

6) Łatwa do prześledzenia. Oznacza to, że dla każdego z wymagań łatwe do ustalenia są: zarówno materiały, na podstawie których zostało ono przyjęte jak i elementy rozwiązań w dalszych fazach rozwoju, które są przez nie kształtowane.

Wyspecyfikowane wymagania powinny dotyczyć:

1) procesów, funkcji i operacji, dla których powinny zostać wyspecyfikowane m.in.:

- algorytmy przetwarzania,
- dane wejściowe i wyjściowe,
- ograniczenia czasowe,
- powiązania z innymi procesami, funkcjami czy operacjami.

2) danych i komunikatów, a dla każdego zestawu danych czy komunikatu powinny zostać określone:

- sposób wprowadzania/wyprowadzania,
- źródło lub przeznaczenie,
- procesy, funkcje lub operacje korzystajace z nich lub generujące je,
- struktura,
- dokładność.

3) charakterystyk oprogramowania takich, jak n.p.:

- formy dialogu z użytkownikiem,
- współpraca z istniejącymi pakietami,
- łatwość adaptacji i zmian określonych wymagań,
- wymogi ochrony danych,
- szczególne wymagania niezawodnościowe

i t.p.

Dokumentacja SRS powinna zostać uzupełniona o opis przebiegu procesu specyfikowania wymagań, przyjętych technik weryfikacji oraz o raport z przeprowadzonych czynności w tym zakresie.

Dokumentacja powinna odwoływać się do przyjętych w procesie specyfikowania wymagań standardów i konwencji oraz do wykorzystanych w toku procesu narzędzi.

Zakres zawartości i forma specyfikacji wymagań dla oprogramowania (SRS) jest przedmiotem standardu IEEE [10].

2.3. Dokumentacja projektowa (SDD)

Dokumentacja projektowa powinna zawierać opis struktury systemu oprogramowania, zbudowanej hierarchicznie i rozbudowanej do takiego stopnia szczegółowości, który umożliwi rozpoczęcie

procesu implementacji (kodowania).

Ogólny opis struktury systemu powinien być przedstawiony w formie diagramu. Struktura tego diagramu powinna uwidoczniać poszczególne moduły kolejnych poziomów hierarchii, zbiory danych oraz zasoby.

Diagram powinien być uzupełniony o:

- opis znaczenia i funkcji poszczególnych modułów,
- powiązania modułów, przepływ danych i sterowania,
- opis dostępu do zasobów,
- zaznaczenie pracy w czasie rzeczywistym.

Dla każdego z modułów powinny zostać określone m.in.:

- sposób identyfikacji,
- opis zadań modułu,
- sposób uruchomienia,
- dane wejściowe (sposób identyfikacji, typ),
- dane wyjściowe (sposób identyfikacji, typ),
- dane lokalne (identyfikacja, typ),
- wykorzystywane przez moduł podprogramy standardowe,
- wykorzystywane przez moduł zasoby,
- logika przetwarzania, wyrażona w postaci np. schematu blokowego, tablic decyzyjnych, tekstu języka wysokiego poziomu lub pseudojęzyka lub t.p.,
- wymagany czas przetwarzania, jeśli wynika z SRS.

Dla każdego zbioru danych powinny zostać okre$lone:

- opis struktury na poziomie logicznym (bloki, pliki, rekordy, pola - zasady ich identyfikacji, definicje, relacje),
- charakterystyka poziomu fizycznego (sposób odwzorowania

obiektów poziomu logicznego na poziom fizyczny).

Opis funkcjonalny oraz charakterystyka poszczególnych elementów struktury systemu powinna jednoznacznie odwoływać sie do określonych wymagań, wyspecyfikowanych w SRS wyjaśniajac równocześnie, w jaki sposób dany element struktury zwiazany jest i w jaki sposób wynika z danego wymagania.

Dokumentacja projektowa powinna zostać uzupełniona o opis przebiegu procesu projektowania, przyjętych strategii i technik weryfikacji oraz o raport z przeprowadzonych w tym zakresie czynności.

Dokumentacja powinna odwoływać się do przyjętych w procesie projektowania standardów i konwencji oraz wykorzystanych w toku procesu narzędzi.

2.4. Dokumentacja kodu źródłowego

Podstawowym elementem dokumentacji jest listing programu.

Wymagania w odniesieniu do formy dokumentacji dotycza:

- graficznego rozmieszczenia elementów tekstu,
- sposobu komentowania,
- formy odwołań do dokumentacji projektowej.

Wymagania te powinny zostać ustalone w sposób jednolity dla danego przedsięwzięcia.

Dokumentacja kodu źródłowego powinna zostać uzupełniona o opis przebiegu procesu implementacji, przyjętych strategii i technik weryfikacji oraz o raport z przeprowadzonych w tym zakresie czynności.

Dokumentacja powinna odwoływać się do przyjętych standardów i konwencji oraz wykorzystanych w procesie implementacji narzędzi.

2.5. Uwaga

Zespoły uczestniczące w procesie rozwoju powinny realizować niektóre czynności weryfikacyjne, podlegające - jak wspomniano wyżej - właściwemu udokumentowaniu. Mówiąc o procesie weryfikacji i atestacji mamy na myśli czynności wykonywane i dokumentowane przez niezależny zespół.

3. Techniki stosowane w procesie weryfikacji i atestacji oprogramowania

Celem procesu weryfikacji i atestacji oprogramowania jest - jak o tym była mowa - stwierdzenie, że wytworzony produkt zachowuje się w sposób oczekiwany, zgodnie ze specyfikacją SRS.

Prawidłowym z punktu widzenia teorii programowania podejściem do tak postawionego zagadnienia jest dokonanie matematycznego dowodu poprawności programu względem odpowiednio zapisanej specyfikacji. W obecnym stanie rozwoju nauki o programowaniu podejście takie, jak dotychczas, nie może być jednak podstawą skutecznej weryfikacji - w większości przedsięwzięć praktycznych. Wynika to zarówno z trudności w sformułowaniu jednolitego matematycznego zapisu wymagań, jak i ze stopnia złożoności przedmiotu dowodzenia.

W wykształconej praktyce, w procesie weryfikacji i atestacji wykorzystuje się szereg rozmaitych technik, które - nie dając na ogół ścisłego dowodu - uprawdopodobniają jednak, że programy działają w sposób zgodny z wyspecyfikowanymi wymaganiami.

Niektóre ze stosowanych technik polegają na analizowaniu dokumentacji kolejnych faz rozwoju oprogramowania, do

dokumentacji kodu źródłowego włacznie (techniki analityczne), inne polegaja na przeprowadzaniu rozmaitych serii testów kodu wynikowego (techniki testowania). Wszystkie techniki stosowane sa przy założeniu ścisłej dyscypliny, której należy przestrzegać stosujac ustalone dla danego przedsięwzięcia metody, techniki, konwencje, standardy oraz narzędzia - odpowiednie dla kolejnych zadań w procesie rozwoju.

Skuteczność technik testowania zależy ponadto od wyboru określonej strategii testowania, ściśle związanej ze strategia implementacji (p.4 niniejszej pracy). Paragraf niniejszy poświęcony jest krótkiemu przeglądowi omawianych w literaturze technik.

3.1. Techniki analityczne

3.1.1. Analiza dokumentacji (inspection)

Technika ta polega na dokonaniu szczegółowego przeglądu dokumentacji w celu:

- stwierdzenia, że jest ona formalnie poprawna oraz, że nie pominięto w niej żadnego elementu,
- stwierdzenia, że w prawidłowy sposób odzwierciedla ona rozwiązania, które wynikają z dokumentacji fazy poprzedniej.

Analizie dokumentacji może podlegać: specyfikacja wymagań (SRS), dokumentacja projektowa (SDD) oraz dokumentacja kodu źródłowego. Przy dokonywaniu tak określonej analizy dokumentacji należy posługiwać się listami kompletności (check-lists), które powinny zostać opracowane dla każdego z rodzajów dokumentacji - przed przystąpieniem do przedsięwzięcia opracowania oprogramowania. Przykłady list kompletności podane są w przewodniku dotyczącym rozwoju komputerowych systemów krytycznych opracowanym przez TC7

(4)
EWICS .

3.1.2. Prześledzenie (walktrough)

Technika ta polega na tym, że zespół ludzi uczestniczących w weryfikacji dokonuje wyboru kilku reprezentatywnych zestawów danych wejściowych, ustala odpowiadające określonym danym wyniki, a następnie - na podstawie dokumentacji - śledzi przewidywany tok przetwarzania.

Prześledzenie jest techniką, którą można stosować w odniesieniu do specyfikacji wymagań (SRS), dokumentacji projektowej (SDD), a także w odniesieniu do tekstu kodu źródłowego. Prześledzenie dokonane dla określonego zestawu danych na podstawie dokumentacji badanej powinno zostać skonfrontowane z wynikami prześledzenia dokonanego dla tego samego zestawu danych na podstawie dokumentacji fazy poprzedniej.

3.1.3. Analiza programu (program analysis)

Analiza programu polega na odtworzeniu funkcji programu na podstawie analizy sekwencji rozkazów języka poziomu maszynowego lub instrukcji języka wyższego poziomu. Analizę programu można przeprowadzić na podstawie dokumentacji projektowej (SDD) lub na podstawie dokumentacji kodu źródłowego.

W przypadku wiąkszych programów przeprowadzenie ich analizy jest przedsięwzięciem wymagającym wspomagania poprzez stosowanie odpowiednich programów.

3.1.4. Wykonanie symboliczne (symbolic execution)

Wykonanie symboliczne polega na skonstruowaniu wyrażenia algebraicznego, które stanowi reprezentację funkcji programu, w drodze konsekwentnego zastąpienia lewych stron we wszystkich

instrukcjach podstawienia przez prawe oraz poprzez skonstruowanie odpowiednich wyrażeń logicznych dla wszystkich instrukcji warunkowych i pętli.

Wykonanie symboliczne można przeprowadzić na podstawie specyfikacji wymagań (SRS) lub na podstawie dokumentacji projektowej (SDD).

Wykorzystanie tej techniki wymaga stosowania odpowiednio sformalizowanych języków specyfikacji wymagań lub zapisu projektu. Stosowanie tej techniki weryfikacji wymaga wspomagania nawet w przypadku średniej wielkości programów. Wspomaganie takie polega na stosowaniu programów konstruujących wspomniane wyrażenia na podstawie opisu programu.

3.1.5. Dowodzenie poprawności programu (program proving)

(5)

Dowodzenie poprawności programu polega na wprowadzeniu do tekstu programu wyrażeń logicznych (assertions), które pełnią rolę warunków "pre" i "post" dla określonych ścieżek logicznych. Należy udowodnić że: jeżeli warunki "pre" są spełnione przed wykonaniem programu oraz jeżeli program się wykona to będą spełnione warunki "post". Należy ponadto udowodnić, że program się wykona (np. nie zapętli). W procesie dowodzenia wykorzystywane są reguły logiczne, które opisują semantykę języka programowania.

Podstawą dla przeprowadzenia dowodu może być opis programu zawarty w dokumentacji projektowej (SDD) lub kod źródłowy.

Przeprowadzenie dowodu nawet dla niewielkich programów jest przedsięwzięciem trudnym. W rozmaitych ośrodkach opracowywane są programy wspomagające proces dowodzenia.

Dowodzenie poprawności programu, podobnie jak wykonanie symboliczne, sa technikami weryfikujacymi prawidłowość wyniku danej fazy w matematycznym rozumieniu tego słowa. Jak już wspomniano, ich zastosowanie jest jednak ograniczone. Można natomiast stosować je w dowolnym przedsięwzięciu w stosunku do niektórych elementów struktury (modułów) tworzonego oprogramowania.

3.1.6. Makietowanie (prototyping)

Technika makietowania polega na zasymulowaniu niektórych aspektów tworzonego oprogramowania na podstawie specyfikacji wymagań. Pozwala ona tym samym np. na zbadanie spójności i kompletności specyfikacji. Pozwala też na prototypowe zademonstrowenie wybranych cech przyszłego oprogramowania, co umożliwia przeprowadzenie częściowej atestacji oprogramowania w bardzo wczesnej fazie jego rozwoju.

Technika makietowania wymaga stosowania sformalizowanych języków specyfikacji wymagań, co umożliwia wykorzystanie programów wspomagajacych tworzenie makiet.

3.2. Techniki testowania

Powszechnie znane jest stwierdzenie, że "testy mogą jedynie wykryć obecność błędów w programie - nigdy jednak nie wykażą, że program błędów nie zawiera". Nie zmienia to faktu, że testowanie jest szeroku akceptowane jako podstawowa technika weryfikacji.

Jak wskazują dotychczasowe doświadczenia oraz rezultaty prac teoretycznych - testowanie programów jest niestety słabo podatne na algorytmizację postępowania i stanowi w dużym stopniu przedmiot swoistej sztuki. Omawiane w niniejszej pracy techniki i

strategie stanowią więc w większym stopniu zestaw wskazówek pozwalających na usystematyzowanie postępowania, nie będąc metodami, które wskazałyby na przykład, jak wygenerować konkretne zestawy danych testowych umożliwiających przetestowanie z określonym poziomem prawdopodobieństwa braku błędów.

Również wymieniane w pracy narzędzia są i mogą być narzędziami jedynie wspomagającymi proces testowania, a nie narzędziami automatyzującymi ten proces.

Techniki testowania można podzielić zasadniczo na dwie klasy. Jedne z nich nastawione są na badanie funkcjonowania programu, którego struktura jest dla testującego niewidoczna, a więc programu widzianego jak "czarna skrzynka" (black-box testing). Pozostałe nastawione są z kolei na badanie funkcjonowania struktury programu. Obraz tej struktury - będąc widocznym dla testującego - pozwala na testowanie programu widzianego, jak "szklana skrzynka" (glass-box testing).

3.2.1. Testowanie czarnej skrzynki

Techniki tej klasy polegają na testowaniu funkcjonalnych aspektów programu wynikowego w oparciu o zestawy danych przygotowanych na podstawie specyfikacji wymagań w celu stwierdzenia, że funkcjonujący program wymagania te spełnia. Testowanie czarnej skrzynki jest więc podstawową techniką atestacji.

Zestawy danych testowych powinny być tak wybrane, aby przebadać wszystkie wynikające ze specyfikacji wymagań aspekty zachowania się programu, wymagane w niej procesy, funcje, operacje, dane, komunikaty oraz charakterystyki.

Podstawową kwestią przy zastosowaniu tego rodzaju techniki jest sposób dokonania wyboru odpowiednich zestawów danych testowych.

W jednym z podejść należy dokonać podziału wszystkich hipotetycznych zestawów danych wejściowych na klasy zestawów równoważnych ze względu na wybrane aspekty działania programu oraz ustalić po kilka reprezentatywnych zestawów danych dla każdej z klas (equivalence partitioning).

Inne podejście zakłada, że po dokonaniu podobnego podziału jak w poprzednim podejściu, przygotowywane zestawy danych powinny zostać wybrane tak, aby testować program na granicach klas lub dla danych przyjmujących minimalne i maksymalne wartości w każdej z nich (boundary-value analysis).

W jeszcze jednym z podejść należy wzspecyfikować warunki logiczne dotyczące danych wejściowych, warunki logiczne dotyczące danych wyjściowych, a następnie skonstruować graf będący obrazem relacji pomiędzy wyspecyfikowanymi warunkami. Graf taki stanowi wówczas podstawę dokonania systematycznego wyboru zestawów danych w taki sposób, aby przebadać możliwie wszystkie relacje wejście/wyjście (cause-effect graphing).

Techniki wspomniane powyżej wymagają dokonania przebiegów testowych na ogół dla znacznej liczby zestawów danych testowych, których właściwy wybór jest zadaniem trudnym. Pozwalają one jednak na gruntowne przetestowanie programu pod kątem wymagań użytkownika.

3.2.2. Testowanie szklanej skrzynki

Techniki tej klasy polegają na testowaniu struktury oprogramowania: modułów, grup modułów oraz całego programu wynikowego - dla zestawów danych wybieranych pod kątem weryfikacji działania poszczególnych jej elementów.

Podstawowym materiałem dla dokonania takiego wyboru jest więc informacja zawarta w dokumentacji projektowej (SDD) oraz dokumentacji kodu źródłowego.

Zasadnicza kwestia w zastosowaniu tego rodzaju technik jest dokonanie wyboru elementów struktury, które maja podlegać weryfikacji.

Hipotetycznie można rozważać możliwości:

- testowania wszystkich permutacji możliwych ścieżek logicznych w programie,
- testowania indywidualnych ścieżek logicznych, przez które może przebiegać sterowanie w programie (module),
- testowania wszystkich bloków sekwencyjnych (ciagów instrukcji pomiędzy punktami rozgałęzień logicznych),
- testowania poszczególnych instrukcji programu.

Pierwsza z możliwości jest w oczywisty sposób nierealna do zastosowania dla dowolnego nietrywialnego programu.

Ostatnia z możliwości - co jest oczywiste- nie może stanowić podstawy jakiegokolwiek uprawdopodobnienia braku b;ędów logicznych w strukturze programu.

Techniki testowania szklanej skrzynki polegają na tym, aby wyboru testowanych elementów struktury dokonywać w sposób odpowiadajacy dwóm pośrednim z wymienionych możliwości.

Jedna z możliwości wyboru polega na tym, aby wykonanie testów spowodowało przejście sterowania przynajmniej raz przez każdą instrukcję i każde rozgałęzienie (decision or branch coverage).

W drugim wariancie wyboru wymaga się, aby wykonanie testów spowodowało przejście sterowania przez każdą z instrukcji rozgałęzienia wielokrotnie, dla wszystkich kombinacji warunków

logicznych w danym rozgałęzieniu (multiple-branch coverage).

Trzeci wreszcie wariant wyboru wymaga, aby sterowanie prześledziło wszystkie wykonalne ścieżki programu. W praktyce zakłada się pogrupowanie ścieżek w klasy (na przykład klasa petli dla określonego przedziału liczby wykonań) i testowanie klas ścieżek.

Podstawową trudnością w stosowaniu technik tej klasy jest dokonywanie wyboru zestawów danych testowych w taki sposób, aby dany zestaw "trafiał" w określone elementy struktury.

Techniki te pozwalają na gruntowne przetestowanie całej struktury programu i zbadanie stopnia pokrycia tej struktury przeprowadzonymi testami zwłaszcza, jeżeli wykorzystuje się odpowiednie dla tego celu oprogramowanie narzędziowe.

Funkcje przykładowego oprogramowania wspomagajacego proces testowania omówione są poniżej.

3.2.3. System automatyzacji testowania

Opis funkcjonalny przykładowego systemu automatyzacji testowania (Automated Verification System) podaje Deutsch (1)

Wymienia on pięć podstawowych funkcji takiego systemu.

1) Analiza statyczna kodu źródłowego i utworzenie bazy danych o jego strukturze logicznej.

2) Generowanie raportów z przeprowadzonej analizy kodu źródłowego, zawierających informacje na temat struktury sterowania i przepływu danych oraz na temat miejsc, w których mogą ujawniać się błędy logiczne.

3) Wyposażenie kodu w dodatkowe sekwencje instrukcji, których zadaniem będzie zbieranie danych o faktach przejścia sterowania

przez określone miejsca programu oraz o wartościach danych w tych miejscach.

4) Generowanie raportów o przeprowadzonych testach oraz o wynikach ich analizy.

5) Generowanie raportów, które wspomagać beda przygotowanie zestawów danych dla kolejnych przebiegów testowych.

System taki może zawierać cztery moduły.

Pierwszy (moduł analizy statycznej) dzieli analizowany program na ciągi instrukcji pomiędzy punktami rozgałęzień logicznych tworząc odpowiadająca mu strukturę grafu. Wywołania podprogramów - z uwzględnieniem ich hierarchii - zostają opisane w formie struktury drzewiastej.

Moduł ten tworzy tym samym strukturę logiczną programu umieszczając ją w bazie danych i wyprowadzając w postaci raportu (funkcje 1 i 2).

Drugi (moduł instrumentacji) działa jako preprocesor, wprowadzając przed procesem kompilowania dodatkowe instrukcje do oryginalnego kodu źródłowego. Wprowadzone instrukcje będą miały za zadanie rejestrować wykonanie określonych elementów programu, rejestrować wartości danych, rejestrować naruszenia przez te wartości warunków założonych przez osobę testującą oraz tworzyć w tym zakresie statystykę (funkcja 3).

Trzeci (moduł analizatora) działa jako postprocesor - po dokonaniu kompilacji programu wyposażonego w instrukcje dodatkowe oraz po wykonaniu przebiegów testowych. Tworzy on raport zawierający dane na temat:

- liczby wykonanych instrukcji i jej stosunku do liczby wszystkich instrukcji,

- liczby wykonanych bloków sekwencyjnych i jej stosunku do liczby wszystkich bloków sekwencyjnych w programie,
- zakresów wartości danych w wykonanych testach,
- zarejestrowanych odchyleń wartości danych od wartości zakładanych lub zarejestrowanych naruszeń założonych warunków

Czwarty (moduł wspomagający przygotowanie danych) przetwarza dane statystyczne o wynikach testów w taki sposób, aby ułatwić: przygotowanie kolejnych serii zestawów danych testowych, przygotowanie instrukcji dodatkowych stanowiących instrumentację tekstu i t.p. (funkcja 5).

Deutsch opisuje działanie komercyjnego systemu automatyzacji testowania, na który składają się dwa podsystemy: ,

- V-IFTRAN,
- RXVP80,

opracowane w General Research Corporation w Kalifornii.

4. Strategie testowania

Całkowitą pewność, że program jest poprawny względem danej specyfikacji może dać - o czym już kilkakrotnie przypominano - jedynie formalnie przeprowadzony dowód. Wspomniano też, że w większości spotykanych w praktyce przypadków dowodu takiego nie można w pełni konsekwentnie przeprowadzić.

Uzyskanie programów poprawnych jest w praktyce wynikiem szeregu działań równoczesnych takich, jak:

- dążenie do uzyskania jednoznacznego, sformalizowanego zapisu specyfikacji,
- dążenie do przestrzegania dyscypliny programowania

systematycznego,

- dążenie do stosowania sformalizowanych języków opisu struktury oprogramowania w kolejnych fazach jego tworzenia,
- systematyczne weryfikowanie wyników każdej fazy względem fazy poprzedniej.

Wiarygodność testowania, jako techniki weryfikującej zgodność wyniku fazy implementacji z opracowanym projektem, zależy w znacznym stopniu od przyjętej strategii implementacji i testowania, t.j. od kolejności, w jakiej kodowane są i testowane poszczególne moduły tworzonego programu.

Testowanie programu powinno być zsynchronizowane w swojej strategii z procesem kodowania, dokonywanego w oparciu o szczegółowy projekt programu.

Należy jednak zwrócić uwagę na to, że:

- im wcześniej wykryte zostają błędy w kodowaniu, tym mniej kosztowne jest usuwanie ich oraz ich skutków (co jest oczywiste),
- prawidłowo przeprowadzone testowanie prowadzone pochłania ok. 50% kosztów realizacji przedsięwzięcia (Deutsch [1], s.12).

Wynika stąd wniosek, że przyjęta dla konkretnego projektu strategia kodowania powinna wynikać z odpowiedniej dla tego projektu strategii testowania.

Zanim przedstawimy poszczególne rodzaje strategii, omówimy najpierw dwa różne aspekty testowania: testowanie modułów i testowanie integracji.

Przypomnimy też, że w projekcie oprogramowania odzwierciedlona zostaje hierarchiczna struktura modułów, dla każdego z których określono m.in.:

- specyfikację funkcjonalną,

- powiązania z innymi modułami (przepływ danych i sterowania),

- powiązania z zasobami systemu.

4.1. Testowanie modułów

Celem testowania modułów jest zweryfikowanie, że moduł prawidłowo realizuje funkcje określone w jego specyfikacji funkcjonalnej.

Testowanie modułu obejmuje na ogół dwie grupy kwestii:

- logikę modułu,

- obliczenia modułu.

Celem testowania logiki jest wykrycie potencjalnych błędów takich, jak np.:

- zapętlanie ścieżek,

- błędy decyzji logicznych,

- brak decyzji w przypadku niektórych, mogacych wystąpić kombinacji warunków logicznych spełnianych przez dane,

- brak właściwej reakcji w przypadku pominięcia niektórych danych.

Celem testowania obliczeń jest wykrycie potencjalnych błędów dotyczących np.:

- prawidłowej sekwencji obliczeń,

- dokładności obliczeń,

- właściwego inicjowania algorytmów numerycznych,

- prawidłowego działania oraz sposobu wyprowadzania wyników obliczeń,

- czasu przetwarzania, jeżeli w projekcie założono określone wymagania w tym zakresie.

Ewentualnych błędów należy szukać w przypadkach:

- danych, które objęte są specyfikacją funkcjonalną,

- danych, które są niezgodne ze specyfikacją,

- danych, które leżą na granicy określonej przez specyfikację.

4.2. Testowanie integracji

Celem testowania integracji jest zweryfikowanie, że przetestowane wcześniej moduły prawidłowo współpracują z sobą, zgodnie z zasadami przekazywania danych i sterowania określonymi w projekcie.

W szczególności należy przetestować:

- powiązania modułów (w tym prawidłowe przekazywanie danych i sterowania, porządek, liczbę i typ argumentów w wywołaniach procedur i t.p.),
- występowanie niepożądanych efektów brzegowych,
- sposób zakńczenia obliczeń w przypadku przerwania, jeżeli moduł obsługi przerwań stanowi jeden z integrowanych modułów,
- czynności takie, jak n.p. ustawianie wskaźników globalnych (flag) systemu, inicjowanie wartości zmiennych i t.p.,
- czasy komunikacji między modułami, jeśli założenia takie wynikają z projektu.

4.3. Testowanie fazowe (phase testing) a testowanie przyrostowe (incremental testing)

Testowanie fazowe różni się od testowania przyrostowego sposobem agregowania integrowanych modułów. Różnicę tę wyjaśnimy na przykładzie.

W przypadku, w którym pewna liczba modułów (np. A0,...,An) ma składać się na moduł wyższego poziomu hierarchii (np. A), testowanie fazowe polega na przyjęciu następującej kolejności działań:

1) projektuje się, koduje i testuje kolejno moduły A0,...,An,

2) składa się przetestowane moduły w moduł A, który testowany jest (w aspekcie modułu i integracji) jako całość.

Testowanie przyrostowe w tym samym przypadku polega na przyjęciu kolejności działań:

1) projektuje się, koduje i testuje jeden z modułów spośród A0,...,An,

2) projektuje się, koduje i testuje kolejny spośród modułów A0,...,An,

3) moduł przetestowany w kroku 2. składa się z dotychczas zintegrowanym fragmentem modułu A i poddaje testom w aspekcie integracji,

4) kroki 2. i 3. powtarza się do wyczerpania zbiory A0,...,An.

Niewątpliwą zaletą testowania przyrostowego jest to, że w każdym z kolejnych kroków dodawany jest tylko jeden potencjalnie błędny moduł. Dotyczy to tylko jednak testowania w aspekcie integracji. Testowanie logiki i testowanie obliczeń dla powstających fragmentów jest jednak utrudnione, ze względu na nieokreśloną specyfikację funkcjonalną takich fragmentów.

Testowanie fragmentów modułu A w aspekcie testowania madułów staje się możliwe poprzez zastąpienie nieistniejących jeszcze modułów tzw. zaślepkami (p.4.4.).

4.4. Testowanie TOP-DOWN a testowanie BOTTOM-UP

Dla dużych, złożonych strukturalnie systemów, integrowanie modułów przebiega na ogół "z dołu do góry" (BOTTOM-UP). Możliwa jest jednak kolejność, w ramach której kodowanie i testowanie modułów oraz ich integrowanie przebiega "z góry w dół" (BOTTOM-UP) w strukturze hierarchicznej.

Można prześledzić to na następujacym przykładzie.

Niech strukture pewnego systemu tworzą:

- moduł sterujacy M,
- moduły niższego poziomu hierarchii: A i B,
- moduł sterujący A0 oraz moduły A1,...,An, tworzące wspólnie strukture modułu A,
- moduł sterujący B0 oraz moduły B1,...,Bm, tworzące wspólnie strukture modułu B.

Testowanie i integrowanie TOP-DOWN odbywać sie będzie w kolejności modułów: M,A0,A1 do An,B0,B1 do Bm.

Testowanie i integrowanie BOTTOM-UP odbywać się będzie natomiast w kolejności A1 do An,A0,B1 do Bm,B0,M.

Jest oczywiste, że zrówno testowanie oraz integrowanie TOP-DOWN, jak i BOTTOM-UP wymaga pewnego "oprzyrządowania".

Integrowanie TOP-DOWN i testowanie kolejnych fragmentów powstajacego systemu wymaga utworzenia zaślepek (stubs). Zadaniem zaślepki jest zastąpienie brakujacego modułu stojacego niżej w hierarchii struktury oraz zasymulowanie jego działania w takim zakresie, w jakim jest to niezbędne dla pełnego testowania integrowanego fragmentu.

Integrowanie i testowanie BOTTOM-UP wymaga utworzenia modułów prowadzących (drivers). Zadaniem modułu prowadzącego jest zasymulowanie działania brakujacego modułu stojacego wyżej w hierarchii struktury i współpracującego, zgodnie z projektem, z modułem testowanym.

Posługiwanie sie modułami prowadzacymi i zaślepkami pozwala na planowanie kolejności kodowania, testowania i integrowania w sposób stosunkowo dowolny, mieszany w stosunku do wymienionych

wyżej przeciwstawnych w stosunku do siebie strategii.
Na przykład testowanie modułu A0 jako pierwszego (we wspomnianej wyżej przykładowej strukturze) wymaga utworzenia modułu prowadzacego, symulujacego moduł M oraz zaślepek, symulujacych moduły A1,...,An.
Planowanie kolejności kodowania, testowania oraz integrowania poszczególnych modułów powinno zakładać zarówno sposób agregowania (fazowy, przyrostowy, mieszany), jaki i kierunek w hierarchii struktury (TOP-DOWN, BOTTOM-UP, mieszany).
W planowaniu tym należy uwzględnić konieczność zaprojektowania oraz zaimplementowania wspomnianego oprzyrządowania.
Oprogramowanie będace tym oprzyrządowaniem, zestawy danych testowych oraz przyjęta sekwencja działań stanowią tzw. procedurę testowania (por.5).

4.5. Testowanie wstążek funkcjonalnych (stimulus-response threads)

Deutsch (1) opisuje strategię rozwoju i testowania oprogramowania stosowaną w Hughes Aircraft Company (USA). Jej podstawowe zasady są następujące.
Punktem wyjścia dla zaplanowania procesu jest specyfikacja wymagań (SRS), na podstawie której opracowany zostaje tzw. diagram weryfikacji systemu (system verification diagram). Diagram ten składa się z kolejnych "wstążek" (threads), które odpowiadają parom "pobudzenie-reakcja" (stimulus-response). Przez pobudzenie należy rozumieć w tym przypadku klasę możliwych "wejść" do systemu (komunikatów, przerwań zewnętrznych, danych), a przez reakcję - odpowiadającą jej klasę oczekiwanych "wyjść"

(komunikatów, sygnałów, danych wynikowych). W ten sposób każda ze wstążek związana jest ściśle z określonymi funkcjami systemu oraz z innymi wyspecyfikowanymi wymaganiami, które sa z nimi związane. Analiza projektu pozwala na dokonanie odwzorowania, w którym każdej ze wstążek odpowiada podgraf w grafie logicznej struktury modułów systemu, obejmujący moduły wykonujace funkcje wstążki. Planowanie kodowania i testowania odbywa się w oparciu o usystematyzowaną kolejność wstążek. Analiza funkcji związanych z każdą z nich pozwala na agregowanie wstążek w tzw. "odcinki" (builds), z których każdy stanowi fragment systemu o określonym, spójnym i wewnętrznie zupełnym zestawie realizowanych przez siebie funkcji.

Zalety opisanej strategii są następujące.

1) Możliwe jest zaplanowanie procesu implementacji i testowania etapami, w każdym z których otrzymuje się spójny i wewnętrznie zupełny funkcjonalnie podzbiór systemu - kolejno do jego pełnej wersji.

2) Testowanie kolejno kodowanych wstążek pozwala na odwołanie się do specyfikacji wymagań (SRS). Możliwe jest więc tym samym etapowe - wcześnie rozpoczynane - dokonywanie atestacji oprogramowania.

5. Organizacja i przebieg procesu weryfikacji i atestacji

Proces weryfikacji i atestacji powinien zostać szczegółowo zaplanowany i zorganizowany. Jednym z podstawowych warunków zapewnienia jakości oprogramowania (8) jest mianowicie sporządzenie -przed przystąpieniem do realizacji przedsięwzięcia- planu weryfikacji i atestacji oprogramowania, w którym powinny

znaleźć odzwierciedlenie zarówno przebieg procesu, jak i jego organizacja.

Plan taki powinien uwzględniać konkretne warunki realizacji przedsięwzięcia, a w szczególności:

- charakter powstającego oprogramowania (oprogramowanie wykonywane na zamówienie czy powielarne dla szerokiego kręgu użytkowników, stopień "krytyczności" i t.d.),
- właściwe dla tego przedsięwzięcia fazy w procesie rozwoju.

Plan powinien określać w swej części ogólnej:

- strukturę organizacyjną jednostki prowadzącej proces weryfikacji i atestacji, ogólny podział zadań, odpowiedzialności i kompetencji,
- wymagania dotyczące zawartości i formy dokumentowania zadań wykonanych w toku procesu oraz zasady zarządzania tą dokumentacją,
- zasady postępowania korekcyjnego w przypadku stwierdzenia nieprawidłowości badanych wyników,
- zasady dokonywania zmian w planie

oraz inne wymagania ogólne w stosunku do organizacji procesu.

Dla każdej z faz rozwoju plan powinien określać:

a) planowane do wykonania zadania w ramach procesu weryfikacji i atestacji,

b) kryteria, metody i techniki weryfikacji wyników danej fazy,

c) materiały, w oparciu o które przeprowadzone będą poszczególne zadania (wejściowe) oraz materiały, które stanowić będą wynik ich

wykonania (wyjściowe),

d) harmonogram realizacji zadań,

e) narzędzia przewidziane do zasosowania przy realizacji poszczególnych zadań,

f) znaczenie poszczególnych zadań w realizacji całego przedsięwzięcia,

g) organizacyjny podział zadań, odpowiedzialności i kompetencji w toku procesu.

Należy zwrócić uwagę, że szczególnie złożone sa zadania weryfikacji i atestacji związane z procesem testowania. Obejmuja one bowiem:

- zaplanowanie właściwej serii testów,
- dokonanie wyboru właściwych zestawów danych testowych,
- zaprojektowanie testów,
- przygotowanie oprzyrządowania dla testów,
- wygenerowanie procedur testowania,
- przeprowadzenie testowania i dokonanie analizy wyników testów.

Wyróżnia sie, ze względu na funkcję w procesie weryfikacji i atestacji, nastepujace cztery rodzaje testowań:

1) Testowanie modułów, które jest sposobem weryfikacji zaimplementowanych modułów (por.4.1).

2) Testowanie integracji, które jest sposobem weryfikacji prawidłowego współdziałania modułów (por.4.2).

3) Testowanie systemu, którego celem jest zweryfikowanie prawidłowego współdziałania wszystkich modułów systemu oraz jego integrację ze środowiskiem sprzętowym.

4) Testowanie odbioru, którego celem jest stwierdzenie, że

funkcjonujący w środowisku operacyjnym użytkownika system oprogramowania spełnia wymagania wyspecyfikowane w SRS (atestacja).

Zadania związane z procesem testowania powinny zostać zaplanowane z maksymalnie dopuszczalnym przez procesy rozwoju wyprzedzeniem.

Już w fazie specyfikowania wymagań powinno zostać zaplanowane testowanie systemu i testowanie odbioru.

Plany testowania modułów i testowania integracji powinny zostać zaplanowane w fazie projektowania, wyznaczając tym samym strategię implementacji.

Przygotowanie danych oraz wygenerowanie procedur dla poszczególnych rodzajów testowania powinno zostać przeprowadzone w fazie implementacji. W fazie tej - zgodnie z przyjętą strategią - rozpoczyna się proces testowania modułów oraz testowania integracji. Testowanie modułów powinno zostać w tej fazie zakończone.

W fazie testowania zostaje zakończone testowanie integracji oraz przeprowadzone testowanie systemu i testowanie odbioru - zgodnie z wygenerowanymi procedurami.

Postać formalna oraz wymagania dotyczące zawartości planu weryfikacji i atestacji oprogramowania stanowią przedmiot standardu IEEE [3].

Przedmiotem standardów IEEE są również wymagania w stosunku do planowania procesu testowania [6] oraz wymagania dotyczące zawartości dokumentacji procesu testowania [9].

6. Znaczenie procesu weryfikacji i atestacji dla jakości oprogramowania

Należy zwrócić uwagę, że nakreślony powyżej model sekwencyjnego przebiegu procesu weryfikacji i atestacji oprogramowania, zsynchronizowanego z procesem jego rozwoju, jest modelem wyidealizowanym i wymaga szeregu zastrzeżeń.

W praktyce, w wyniku określonych czynności podejmowanych zarówno w toku rozwoju jak i weryfikacji zachodzi konieczność wielokrotnego niekiedy powtarzania czynności weryfikowanej fazy, proces ten ma więc charakter iteracyjny, a w jego wyniku powinien powstać produkt w swojej postaci ostatecznej, dla którego:

- istnieje wzajemnie zgodny układ udokumentowanych wyników kolejnych faz rozwoju,
- oprogramowanie wynikowe jest zgodne z definicją zawartą w specyfikacji wymagań (SRS).

Rezultat taki po rozpoczęciu eksploatacji, pomimo jego formalnej poprawności stwierdzonej w wyniku dokonania atestacji, okazuje się jednak często rezultatem nie satysfakcjonującym w pełni jego użytkownika(ków).

Nie powinno to rodzić zdziwienia ani oburzenia.

Po pierwsze, opracowanie odzwierciedlającej praktyczne potrzeby, a równocześnie dostatecznie precyzyjnej specyfikacji wymagań jest zawsze zadaniem trudnym. Ustalona specyfikacja powinna być więc z założenia traktowana jako pierwsze jej przybliżenia.

Po drugie, w yniku iteracji w procesie rozwoju i wynikających z nich kolejnych rewizji dokumentacji poszczególnych faz może zachodzić nawet konieczność modyfikacji specyfikacji wymagań w

stosunku do jej wersji pierwotnej (jeśli np. okazało się w toku rozwoju, że pierwotna wersja nie jest realizowalna).

Po trzecie wreszcie, rozpoczęcie eksploatacji oprogramowania w środowisku użytkownika stwarza w pewnym sensie nową rzeczywistość, kreującą nowe wymagania. Sytuacja taka jest punktem wyjścia dla tworzenia zmodyfikowanej (rozszerzonej) specyfikacji wymagań i rozpoczęcie nowego cyklu rozwoju (por. modelowe ujęcie cyklów rozwoju tzw. A-systemów [7]).

Iterowanie całego cyklu rozwoju można interpretować jako tzw. pętlę jakości. Wynikająca bowiem z tych iteracji ewolucja oprogramowania prowadzi do tego, że w kolejnym obiegu cyklu (nowa wersja produktu) oprogramowanie spełnia w większym stopniu faktyczne wymagania użytkownika(ków).

Systematycznie prowadzony rozwój oprogramowania, niemożliwy bez starannego dokumentowania oraz bez systematycznego weryfikowania i atestacji, jest więc zasadniczym warunkiem zapewnienia jakości oprogramowania. Jakość taka ma szczególne znaczenie w przypadkach, w których eksploatacja wadliwie opracowanego oprogramowania może być przyczyną zagrożeń lub szkód. Zalecenia odzwierciedlone np. we wspomnianych w niniejszej pracy standardach opracowywane były głównie z myślą o takim właśnie oprogramowaniu (oprogramowanie "krytyczne"). Wynikające z nich jednak zasady mają znaczenie ogólne, porządkujące proces tworzenia oprogramowania i pozwalające na podnoszenie jego jakości.

7. Literatura

1. M.S.Deutsch, Software Verification & Validation, Prentice Hall 1982,
2. Software Life Cycle Processes, IEEE Std 1074-1988,
3. Software Verification & Validation Plans, IEEE Std 1012-1986,
4. Dependability of Critical Computer Systems, F.J.Redmill ed., EWICS TC7, Elsevier 1988,
5. Hoare, C.A.R., An Axiomatic Basis ..., CACM 12,ao (1969),
6. Software Unit Testing, IEEE Std 1008-1987,
7. Lehman, M.M.,Belady, L.A., Software Evolution, AP 1985,
8. Software Quality Assurance Plans, IEEE Std 730-1981,
9. Software Test Documentation, IEEE Std 829-1983,
10. Software Requirements Specification, IEEE Std 830-1984.